Kylix

Programmer's Choice

PEARSON Education

ADDISON-WESLEY

Elmar Warken

Kylix

Delphi für Linux

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • 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 Produkt werden ohne Rücksicht auf einen
eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Abbildungen und Texten wurde mit größter
Sorgfalt vorgegangen.
rotzdem 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 Produkt 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 01

ISBN 3-8273-1686-3

© 2001 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Christine Rechl, München
itelbild: Acanthus mollis, Akanthus. © Karl Blossfeldt Archiv –
Ann und Jürgen Wilde, Zülpich/VG Bild-Kunst Bonn, 2001.
Lektorat: Christina Gibbs, cgibbs@pearson.de
Korrektorat: Simone Burst, Großberghofen
Herstellung: TYPisch Müller, Arcevia, typmy@freefast.it
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Kösel, Kempten (www.koeselbuch.de)
Printed in Germany

Inhalt

Vorwort **13**

1 Die visuelle Programmierumgebung **19**

1.1 Kylix-Anwendungen sind ereignisorientiert 20
1.1.1 Zu jedem Anlass ein Ereignis 20
1.1.2 Ereignisse in Kylix 22
1.2 Von Formularen und Komponenten 24
1.2.1 Die Fensterhierarchie einer GUI-Anwendung 24
1.2.2 Formulare und Fenster 25
1.2.3 Komponenten 26
1.3 Der Entwicklungszyklus 27
1.3.1 Der Entwicklungszyklus in der Kylix-IDE 27
1.3.2 Übersicht über das Beispielprogramm 28
1.4 Die IDE und die visuellen Werkzeuge 30
1.4.1 Der Aufbau der IDE 30
1.4.2 Hilfe zu IDE und Sprachreferenz 35
1.4.3 Die Komponentenpalette 36
1.4.4 Entwerfen von Formularen 38
1.4.5 Dateiverwaltung 43
1.4.6 Der Objektinspektor 44
1.4.7 Property-Typen 50
1.4.8 Menüs 53
1.4.9 Grafiken 57
1.5 Ereignisse 59
1.5.1 Einführung in die Ereignisbearbeitung 59
1.5.2 Schnellübersicht: Pascal für Ereignisbearbeitungsmethoden 63
1.5.3 Eine Übersicht über die Formular-Unit 65
1.5.4 Die Ereignisse des Beispielprogramms 67
1.5.5 Methoden für die Statuszeile 70
1.5.6 Weiterer Progamm-Ausbau 73
1.5.7 Ereignisverknüpfung: Navigieren, verändern, lösen 77
1.5.8 Drei Blicke hinter die Kulissen 79
1.6 Bearbeiten von Projekten 82
1.6.1 Der Quelltext-Editor 83
1.6.2 Aufbau von Projekten 89
1.6.3 Die Projektverwaltung 91
1.6.4 Objektablage und Komponentenschablonen 95

1.6.5 Der Browser 100
1.6.6 To-Do-Listen 102
1.7 Der Debugger 104
1.7.1 Übersetzungsoptionen für den Debugger 105
1.7.2 Allgemeine Debugger-Fenster 106
1.7.3 Breakpoints 110
1.7.4 Variablen untersuchen 114
1.7.5 Code-Ausführung 117
1.7.6 Assertions 119
1.8 Komponenten kurz vorgestellt 121
1.8.1 Verwendung von Formularen als modale Dialoge 124
1.8.2 Komponenten zur Programmsteuerung 128
1.8.3 Listenelemente und die Sicherung ihres Inhalts 132
1.8.4 Eingabekomponenten 142
1.8.5 Ausgabekomponenten 151
1.8.6 Komponenten zu Gestaltung und Strukturierung 161

Die Sprache Object Pascal 165

2.1 Überblick 166
2.1.1 Object Pascal für Umsteiger 166
2.1.2 Lexikalische Elemente 169
2.1.3 Compileranweisungen 172
2.1.4 Typen und Variablen 175
2.1.5 Konstanten und initialisierte Variablen 176
2.1.6 Gültigkeitsbereiche und lokale Variablen 178
2.1.7 Das Modulkonzept 179
2.2 Objekte und Klassen in Object Pascal 181
2.2.1 Der Aufbau von Objekten 182
2.2.2 Klassen und Instanzen 183
2.2.3 Die Klassendeklaration 183
2.2.4 Zugriff auf die Klassenelemente 184
2.2.5 Properties 186
2.2.6 Zugriffsbeschränkungen 190
2.2.7 Vererbung 191
2.2.8 Vorwärtsdeklaration von Klassen 191
2.3 Der Lebenslauf von Objekten 192
2.3.1 Initialisierung von Objekten: Konstruktoren 192
2.3.2 Aufräumen mit Destruktoren 195
2.3.3 Polymorphie durch virtuelle Methoden 197
2.3.4 TClass – die Klasse der Klassen 205
2.3.5 Klassenmethoden 208
2.3.6 Typinformationen zur Laufzeit 209
2.3.7 Kompatibilität mit Borland Pascal 213
2.4 Typen 213
2.4.1 Einfache Typen 213
2.4.2 Operatoren und Ausdrücke 219
2.4.3 Arrays 222
2.4.4 Die verschiedenen Stringtypen 225

2.4.5 Strukturierte Typen 232
2.4.6 Zeigertypen 234
2.4.7 Typenkompatibilität und Typenumwandlungen 237
2.4.8 Initialisierte Konstanten strukturierter Typen 239
2.5 Anweisungen und Funktionen 239
2.5.1 Pascal-Anweisungen 240
2.5.2 Prozeduren und Funktionen 244
2.5.3 Parametertypen 246
2.5.4 Überladen von Funktionen, Standardparameter 251
2.5.5 Prozedurtypen 253
2.6 Fehlerbehandlung mit Exceptions 254
2.6.1 Verhängung des Ausnahmezustandes 255
2.6.2 Exception-Klassen 255
2.6.3 Schadensbegrenzung mit finally 257
2.6.4 Behandeln der Exceptions 258
2.6.5 Optionen für Exceptions 261
2.6.6 Exceptions im Beispielprogramm 262
2.7 Interfaces 264
2.7.1 Abstrakte Basisklassen versus Interfaces 265
2.7.2 Verwendung eines Interfaces 267
2.7.3 IUnknown, Co-Klassen und andere Begriffe 269
2.7.4 Implementierung eines Interfaces 272
2.7.5 Interface-Mix-In 276
2.8 Funktionsbereiche der Laufzeitbibliothek 281
2.8.1 Dateiverwaltung 281
2.8.2 Dateieingabe und -ausgabe 282
2.8.3 Zeitformat und Zeitfunktionen 285
2.8.4 Formatierungs-/Stringfunktionen 286
2.8.5 Sonstige Funktionen 289

3 Die Komponentenbibliothek CLX 291

3.1 Überblick über die CLX 296
3.1.1 Die grundlegenden Klassen 297
3.1.2 Komponenten 300
3.1.3 Visuelle Komponenten 302
3.1.4 Der Nachrichtenfluss 303
3.2 Die Beziehungen der Komponenten 308
3.2.1 Besitzhierarchie und Komponentenlisten 308
3.2.2 Die Fensterhierarchie 310
3.2.3 Die oberste Komponente: TApplication 312
3.2.4 TScreen 317
3.3 Grundlegende Gemeinsamkeiten von Steuerelementen 319
3.3.1 Grundlegende Eigenschaften 320
3.3.2 Maus- und Tastatureingaben 326
3.3.3 Aktionen beim Bewegen der Maus 329
3.3.4 Anzeigesteuerung 333
3.3.5 Kontrolle über Größe und Position 334
3.3.6 TWidgetControl 336

3.4 Formulare, TScrollingWidget und TScrollBox 337
3.4.1 TScrollingWidget und TScrollBox 337
3.4.2 Die verschiedenen Arten von Formularen 339
3.4.3 Eigenschaften und Ereignisse der Formulare 344
3.4.4 Arbeiten mit mehreren Formularen 346
3.5 Aufbau von Formularen und Verwendung von Dialogen 349
3.5.1 Steuerelemente in Gruppen 349
3.5.2 Gleichzeitige Behandlung mehrerer Komponenten 351
3.5.3 Steuerelemente in Arrays 355
3.5.4 Mehrseitige Dialoge 358
3.5.5 Formulardateien im Textmodus editieren 363
3.5.6 Verwenden der Standarddialoge 366
3.6 Komplexere Steuerelemente 372
3.6.1 Editierfelder und Memos 372
3.6.2 ListViews, IconViews und Bilderlisten 375
3.6.3 Baumanzeige-Komponenten (TreeViews) 390
3.7 Frames und verwandte Techniken 395
3.7.1 Formularvererbung 397
3.7.2 Frames 403

Weitere Klassen der CLX 409

4.1 Grundlegende Datenstrukturen: TList und TStrings 410
4.1.1 TStrings 410
4.1.2 Ableiten einer History-Liste von TStringList 414
4.1.3 TList 416
4.1.4 Andere Container-Klassen 417
4.2 Textbasierte Konfigurationsdateien 417
4.2.1 Dateien im INI-Format 418
4.2.2 Speichern der History-Liste in einer INI-Datei 419
4.2.3 Die TStateSaver-Komponente 421
4.3 Streams und Ablageobjekte 424
4.3.1 Stream-Klassen 424
4.3.2 Der Inhalt von Streams 425
4.3.3 Polymorphe Objekte speichern 426
4.3.4 TReader und TWriter 430
4.3.5 Memory-Streams 432
4.4 Grafikausgabe 434
4.4.1 Die Klasse TCanvas 434
4.4.2 Zeichenwerkzeuge 436
4.4.3 Grafikmethoden 441
4.4.4 Besitzergezeichnete Komponenten 446
4.5 Grafiken als Objekte 449
4.5.1 Die drei TGraphic-Klassen 449
4.5.2 Bitmaps 451
4.5.3 TPicture 452
4.5.4 Bitmaps für eine OwnerDraw-Listbox 455
4.5.5 TDrawing und Metadateien 462

4.6 Menüs 464
4.6.1 Die Unit Menus 465
4.6.2 Dynamische Menüerweiterungen 470
4.6.3 Ein dynamischer Tastenkürzeleditor 478
4.6.4 Befehlslisten mit TActionList 483
4.7 Die Zwischenablage 488
4.7.1 Vordefinierte Zwischenablage-Operationen 488
4.7.2 TClipboard 490
4.7.3 Eigene Ablageformate 493
4.8 Threads 495
4.8.1 Multitasking-Typen 495
4.8.2 Threads in der CLX 503
4.8.3 Mehrere Threads und deren Synchronisation 508
4.8.4 Ein Utility mit dynamischer Thread-Anzahl 521

5 Kylix-Anwendungen in der Linux-Umgebung 527

5.1 Der Kernel und andere C-Bibliotheken 528
5.1.1 Aufruf von Bibliotheks-Funktionen allgemein 529
5.1.2 Deklarationen von externen Funktionen in Object Pascal 532
5.1.3 Aufruf der externen Funktionen 534
5.1.4 Typübersetzung von C nach Object Pascal 535
5.1.5 Ein Mini-Systemmonitor als Beispiel 540
5.1.6 X Window und die XLib 544
5.1.7 Shared Libraries mit Kylix 546
5.2 Konsolenanwendungen 548
5.3 Prozesse und Shells 550
5.3.1 Aufrufen anderer Programme und Tools 550
5.3.2 Eine allgemeine Programmstart-Routine 554
5.3.3 Ausführen von Shell-Skripten 559
5.3.4 Die Umgebung eines Prozesses 563
5.4 Die Anbindung an die Qt-Bibliothek 564
5.4.1 C++ und Object Pascal 564
5.4.2 Die Brücke von Qt zu Object Pascal 565
5.4.3 Das Weltkoordinatensystem von Qt 569
5.5 Ein KDE-Widget als Komponente 577
5.5.1 Ein fest eingebundenes KDE-Widget 577
5.5.2 Laden von Bibliotheken zur Laufzeit 586

6 Dokumentbasierte Anwendungen 589

6.1 Der TreeDesigner 590
6.1.1 Wahl eines Beispielprogramms 591
6.1.2 Spezielle Fähigkeiten 592
6.1.3 Kurzbeschreibung und Bedienung 593
6.1.4 Dateien auf der CD 600
6.2 Das Hauptfenster 603
6.2.1 Der Entwurf von Symbol- und Werkzeugleisten 603
6.2.2 Komponenten für die Zeichenfläche 609

6.2.3 Wichtige Menübefehle 610
6.2.4 Typische Formularaufgaben 613
6.3 Das Grafikdokument 617
6.3.1 Das Dokument-View-Konzept 617
6.3.2 Eine Klasse für das Grafikdokument 626
6.3.3 Eine Klasse für die Grafikobjekte 632
6.3.4 Baumstrukturen 644
6.4 Mausaktionen und Zeichnen 650
6.4.1 Die Methoden im TreeDesigner 651
6.4.2 Shift und Variationen eines Ziehvorgangs 655
6.5 Grafikausgabe und Scrolling 659
6.5.1 Ereignisgesteuerte Grafikausgabe 659
6.5.2 OnPaint und das Zeichnen von Objekten 660
6.5.3 Effiziente Grafikausgabe 666
6.5.4 Scrolling 671
6.5.5 Clipping 676
6.6 Skalierung, virtuelle Koordinatensysteme und Druckerausgabe 678
6.6.1 Window- und Viewport-Transformation 678
6.6.2 Das Koordinatensystem des TreeDesigners 686
6.6.3 Drucken 690
6.7 MDI-Anwendungen 702
6.7.1 MDI im Überblick 702
6.7.2 MDI- und SDI-Versionen des TreeDesigners 704
6.7.3 Verschmelzen von Menüs 707
6.7.4 Verwaltung der MDI-Kindfenster 709
6.7.5 Dynamisches Werkzeugleisten-Management 716
6.7.6 Globale Werkzeugleisten und Mauspaletten 725
6.8 Erweiterung der Benutzerschnittstelle 732
6.8.1 Tastatursteuerung 733
6.8.2 Docking von Fenstern 738
6.8.3 Interaktion mit Grafikobjekten 744
6.8.4 Drag&Drop 746
6.9 XML 753
6.9.1 XML-Grundlagen 754
6.9.2 Ein grafischer XML-Betrachter 764
6.9.3 XML-Dateiformate 771

Datenbankanwendungen 777

7.1 Datenbank-Variationen und Datenzugriff 778
7.1.1 MyBase-Datenbanken 780
7.1.2 Interbase-Datenbanken 783
7.1.3 Portierung von Delphi-Desktop-Datenbanken 788
7.1.4 Datenzugriff mit Verbindungskomponenten 792
7.2 Von der Tabelle zum Browser 797
7.2.1 Das grundlegende Datenbankformular 797
7.2.2 Datensensitive Steuerelemente 801
7.2.3 Elementare Funktionen 804

7.2.4 Ein Browser für MyBase-Tabellen 809
7.2.5 Ein Browser für Interbase-Tabellen 811
7.2.6 Anwenden und Verwerfen von Updates 814
7.3 Programmieren mit Feldern 817
7.3.1 TField und die Fields 817
7.3.2 Persistente Felder und der Felder-Editor 822
7.3.3 Feld-Definitionen 826
7.3.4 Ein Beispielprogramm mit dynamischen Feldern 827
7.3.5 Die Daten des aktuellen Datensatzes 829
7.3.6 Eine Dateidatenbank 832
7.3.7 Zusatzfunktionen für die Anwendung 837
7.4 Sortieren, Suchen und Filtern 840
7.4.1 Methoden zum Suchen, Filtern und für Lesezeichen 841
7.4.2 Sortieren mit Indizes 843
7.4.3 Modusabhängige Filter- und Suchfunktionen 845
7.4.4 Aktualisieren der Dateidatenbank 848
7.4.5 Haupt-/Detailformulare und Sortieren 851
7.5 Eine Beispielanwendung 853
7.5.1 Definition der Datenbank 854
7.5.2 Das Datenmodul 858
7.5.3 Die Formulare 865
7.5.4 Updates und das Änderungsprotokoll 869
7.5.5 Mehrbenutzersimulation 874
7.5.6 Updates per SQL 876
7.5.7 Ein SQL-Monitor 881

8 Komponentenentwicklung 885

8.1 Das Komponentenkonzept von Kylix 886
8.1.1 Das Wesen einer Komponente 886
8.1.2 Überblick über die Komponentenentwicklung 887
8.1.3 Das Package-Konzept 889
8.2 Beispiele und Installation 891
8.2.1 Starthilfe durch den Komponenten-Experten 891
8.2.2 Eine Minimalkomponente 892
8.2.3 Installation von Packages und Komponenten 894
8.2.4 Eine sinnvolle Beispiel-Komponente 898
8.3 Die Schnittstelle zum Benutzer 900
8.3.1 Properties 900
8.3.2 Events 902
8.3.3 Events auslösen 905
8.4 Komponenten intern 906
8.4.1 Ereignisse in den Komponenten 906
8.4.2 Komponenten in der Formulardatei 910
8.4.3 Steuerung der Property-Speicherung 912
8.4.4 Die Schnittstelle zur Kylix-IDE 914
8.5 Erweiterung bestehender Komponenten 917
8.5.1 Verändern bestehender Komponenten 917
8.5.2 Erweiterung von TScrollBox 920

8.5.3 Die automatische History-Kombobox 922
8.5.4 Zusammenfassen mehrerer Komponenten 927
8.6 Entwicklung neuer Steuerelemente 931
8.6.1 Eine Testumgebung aufbauen 931
8.6.2 Die Schnittstelle der neuen Farbpalette 932
8.6.3 Implementierung der Komponente 939
8.6.4 Events mit Eingriffsmöglichkeiten 944
8.6.5 Komponenten- und Property-Editoren 947
8.6.6 Speichern 953
8.6.7 Vordefinierte Aktionen 957
8.7 Formulare als Dialogkomponenten 960
8.7.1 Eine Hülle für das Formular 960
8.7.2 Die automatische Aktivierung 961
8.7.3 Implementierung 963
8.8 Weitere Komponenten zum Buch 966
8.8.1 Anzeige von Verzeichnisstrukturen und Dateilisten 966
8.8.2 Abschalten von ausgewählten Meldungsfenstern 969
8.8.3 Delphi-Kompatibilitäts-Komponenten 970
8.8.4 Sonstige Komponenten 971

Anhang A: Rezeptverzeichnis 975

1 Benutzerschnittstelle: Entwurf 975
2 Formulare zur Laufzeit 975
3 Formulare: Management mehrerer Formulare 976
4 Benutzerschnittstelle: Komponenten 976
5 Features 976
6 Benutzereingabe mit Tastatur und Maus 977
Tastatur und Maus 977
Tastatur 977
Maus 977
7 Grafikausgabe 977
8 Drucken 978
9 Ein- und Ausgabe: Dateien 978
Dateien allgemein 978
Konfigurationsdateien 978
10 Datenbanken – ausgewählte Themen 979
11 Komponenten – ausgewählte Themen 979
12 Programmiertechnik 979
Datenstrukturen 979
OOP 979
Erweitertes Pascal 980
Debugging 980
Programmfluss 980

Anhang B: CLX-Hierarchiegrafiken des TreeDesigners 981

Index 987

Vorwort

Für viele Software-Entwickler, die bisher mit Delphi gearbeitet haben, hatte Linux bis vor kurzem einen beachtlichen Nachteil: Bei allem Reichtum an verschiedenen Programmiersprachen gab es unter Linux kein Entwicklungssystem, das die von Delphi bekannten Vorzüge in sich vereinen konnte. Von den vielen Vorzügen sind es vor allem zwei, die bislang nur einzeln zu haben waren: Die Erzeugung kompilierter, nativer Linux-Anwendungen war bisher hauptsächlich in der Hand von C/C++-Compilern, und das von Delphi bekannte Konzept der »Two-Way-Tools« gab es unter Linux bisher nur bei Borlands JBuilder.

Mit Kylix hat sich die Situation nun grundlegend geändert: Delphi-Benutzer werden sich in der IDE von Kylix sofort heimisch fühlen und Linux-Programmierer, die bisher nicht mit Borland-IDEs gearbeitet haben, erhalten nun Gelegenheit, die klassischen Delphi-Merkmale auf die Probe zu stellen:

- Das Konzept der Two-Way-Tools, bei dem die GUI-Tools für den Entwurf der Programmoberfläche nahtlos mit dem Code-Editor zusammenarbeiten. Der Entwickler kann beim Schreiben des Programmcodes die Oberfläche als gegeben betrachten, muss sich dabei nicht um Ereignisverknüpfungen kümmern und kann sich sehr gut auf das eigentlich Wesentliche konzentrieren: den Code, der die eigentliche Funktion des Programms ausmacht. Die »Zweiseitigkeit« der Two-Way-Tools bedeutet, dass man beliebig zwischen dem Editieren des Codes und dem Oberflächendesign wechseln kann.
- Die zugrunde liegende Programmiersprache Object Pascal, die von ihrer Mächtigkeit her eher mit C++ als mit dem im Namen noch erwähnten und bei einigen Menschen Vorurteile auslösenden »Pascal« vergleichbar ist. Object Pascal unterstützt fast alle objektorientierten Konzepte der Sprache C++ und noch einige darüber hinaus (wie etwa die erst in Java eingeführten Interfaces), ist aber leichter zu erlernen und oft auch leichter zu lesen (wobei dies natürlich auch vom Programmierstil abhängt).
- Eine sowohl vom technischen Design als auch von der praktischen Nutzbarkeit her gesehen hervorragende Klassenbibliothek – unter Kylix wird hier die neue CLX verwendet, die auch zugleich erweiterbare *Komponenten*bibliothek ist, in die Sie

selbst geschriebene Komponenten nahtlos integrieren können. Mit Hilfe der Qt-Bibliothek von Trolltech versteckt die CLX die plattformabhängigen Details der Programmierung und ermöglicht die parallele Entwicklung für Linux und Windows.

- Die Effizienz von Entwicklungsumgebung, Compiler und dessen Endprodukt: Das Starten kleinerer Programme aus der Entwicklungsumgebung heraus geht so schnell, dass man kaum einen im Hintergrund arbeitenden Compiler ahnt. Auch bei größeren Projekten untertrifft der Kylix-Compiler die von C++-Programmen gewohnten Build-Zeiten um Faktoren, und dies, ohne dass man dadurch einen wesentlichen Nachteil bei der Ausführungsgeschwindigkeit in Kauf nehmen muss, denn die ausführbaren Programmdateien enthalten kompilierten und optimierten Maschinencode in einem nativen Linux-Dateiformat.

Ein weiteres gewichtiges Merkmal von Kylix ist die vielfältige Erweiterbarkeit, auf die die IDE ausgelegt ist. Das OpenTools-API der ersten Kylix-Version ist zwar hier noch nicht der Weisheit letzter Schluss, da für Formulare, die in die IDE eingebunden werden sollen, die VCL und damit theoretisch der Formular-Designer von Delphi für Windows verwendet werden muss, aber es ist sicher verständlich, dass Borland in der 1.0-ersion zugunsten des von sehr vielen Entwicklern lang erwarteten Verkaufsstarts einige Merkmale, die für viele Entwickler nicht entscheidend sind, auf eine spätere ersion verschiebt.

Ziele des Buchs und erforderliche Vorkenntnisse

Dieses Buch wendet sich an die Entwickler, die sich intensiver mit Kylix beschäftigen wollen, als das für das Schreiben einfacher dialog- oder datenbankorientierter Anwendungen notwendig ist. So behandelt es auch das Entwickeln von eigenen Komponenten, das Schreiben dokumentorientierter Anwendungen und den Einsatz von Object-Pascal-Eigenschaften, die bei der einfachen Formularprogrammierung noch nicht notwendig sind. Besonderer Wert wurde auch darauf gelegt, die nicht-visuellen eile der CLX zu behandeln, dabei auch einmal hinter die Kulissen zu schauen und die Arbeitsweise der CLX zu durchleuchten wie z. B. beim Versand von Nachrichten (Kapitel 3.1.4) oder bei der Anbindung der CLX an Qt (Kapitel 5.4).

Die Darstellung der Sprache Object Pascal betont besonders die neueren Sprachmerkmale, und da die grundlegenden Spracheigenschaften nur kurz besprochen werden, sollten Sie bereits elementare Programmiererfahrungen besitzen, bevor Sie sich dem ersten Kapitel zuwenden. Diese Erfahrungen können Sie beispielsweise bei der Programmierung mit Java, Visual Basic, Turbo Pascal, Perl, Datenbank-Programmiersprachen, C oder C++ erhalten haben. Das Buch enthält Abschnitte, die speziell für Umsteiger vorgesehen sind, so zum Beispiel das Kapitel 2.1, das einige Object-Pascal-Konzepte einführt, die in Basic- oder in Makrosprachen meistens nicht anzutreffen

sind. Darüber hinaus ist vor allem das erste Kapitel bis einschließlich Kapitel 1.7 so geschrieben, dass Sie auch dann alles verstehen können, wenn Sie Pascal nicht kennen. Dies ändert sich in Kapitel 3, von dem an die Sprache Object Pascal als weitgehend bekannt vorausgesetzt wird.

Das Buch, die Kylix-Ausgaben und Delphi-Versionen

Borland bietet drei verschiedene Ausgaben von Kylix an: Die frei verfügbare Open Edition und die Desktop-Developer-Ausgabe unterscheiden sich im Wesentlichen nur in der Art der Lizenz: Während Sie Erstere kostenlos aus dem Internet beziehen können, damit aber keine kommerzielle Software entwickeln dürfen, ist die Desktop-Developer-Ausgabe für die professionellen Entwickler gedacht, die mit ihren Programmen Geld verdienen wollen. Es liegt auf der Hand, dass dieser Unterschied für dieses Buch keine Rolle spielt, da es hier nur um das »Wie« der Software-Entwicklung geht, das in beiden Ausgaben identisch ist. Schließlich gibt es noch die Server-Developer-Ausgabe, welche den kleineren Ausgaben lediglich die Fähigkeit zur Entwicklung von Web-Server-Modulen voraus hat. Dieser Bereich kann in diesem Buch nur sehr am Rande angesprochen werden, so dass auch dieser Unterschied vernachlässigbar ist und sich dieses Buch für alle Kylix-Ausgaben gleichermaßen eignen sollte.

Zum Zeitpunkt der Drucklegung dieses Buches waren übrigens lediglich die professionellen Ausgaben von Kylix in ihrer fertigen Verkaufsversion verfügbar. Was die Open Edition anbelangt, beruhen also alle Informationen in diesem Buch auf den frühen Ankündigungen von Borland. Sollten sich beim tatsächlichen Erscheinen der Open Edition irgendwelche Besonderheiten ergeben, die für die Beispielprogramme auf der CD-ROM oder für die Aussagen dieses Buches von Bedeutung sind, so werden diese auf der Website dieses Buches nachgereicht (`ewlab.de/kylix/buch-homepage.html`).

Da es in diesem Buch auch um die plattformübergreifende Entwicklung geht, ist auch Delphi für Windows damit angesprochen. Die TreeDesigner-Beispielanwendung etwa demonstriert die parallele Entwicklung für zwei Plattformen, indem sie sich sowohl mit Kylix unter Linux als auch ab Delphi 5 unter Windows übersetzen lässt. Um dies und die Kapitel zur plattformübergreifenden Entwicklung dieses Buches unter Windows nachvollziehen zu können, genügt die Standard-Version von Delphi 5 (für alle früheren Delphi-Versionen liegen auf der CD spezielle Varianten des TreeDesigners vor).

Schließlich reisen bei Erscheinen dieses Buchs bereits die Delphi-6-Pakete um die Welt und auch Delphi 6 wird ab der Professional-Version mit der CLX-Bibliothek ausgeliefert. Die Beispielprogramme aus diesem Buch sollten sich, sofern sie keine speziellen Linux-Eigenschaften verwenden, ohne Änderung mit Delphi 6 Professional unter Windows und für Windows (mit Qt) übersetzen lassen. Wie gut die Portierung wirklich funktioniert, konnte jedoch vor dem Druck dieses Buchs noch nicht ausprobiert wer-

en. Sollte es kleinere Probleme geben, so wird dies wahrscheinlich eher an den Beispielprogrammen liegen (Verwendung von Linux-Besonderheiten) als an Delphi 6. Auch für diesen Fall sei für Updates der Beispielprogramme auf die bereits erwähnte Buch-Website verwiesen.

Kapitel-Übersicht

Im Folgenden finden Sie einen kurzen Überblick über die acht Kapitel des Buchs:

- Kapitel 1 beschreibt die Entwicklungsumgebung (IDE) und den grundlegenden Aufbau sowie die Funktionsweise einer Kylix-Anwendung. Sie können wählen zwischen einem Schnelleinstieg, der direkt bei einem kleinen Beispielprojekt beginnt, und einer ausführlicheren Einführung, die auch Hintergründe erläutert.
- Kapitel 2 widmet sich der Programmiersprache Object Pascal. Obwohl sein Schwerpunkt auf der objektorientierten Programmierung liegt, werden auch die Grundlagen von Pascal an geeigneter Stelle so zusammengefasst, dass sie sich auch Umsteiger aneignen können.
- Kapitel 3 beschäftigt sich intensiv mit der CLX und reicht von der Untersuchung wichtiger CLX-Interna über die Beschreibung von Komponenten bis hin zu konkreten Beispielprogrammen. Besonderer Wert wurde dabei auf allgemeine Klassen wie *TComponent*, *TControl* und *TWidgetControl* gelegt, denn wenn Sie diese kennen, kennen Sie bereits einen großen Teil der Eigenschaften aller speziellen visuellen Komponenten.
- Kapitel 4 setzt die in Kapitel 3 begonnene Behandlung der CLX fort, beschäftigt sich aber nicht mehr in erster Linie mit den visuellen Komponenten, sondern mit nicht-visuellen Klassen wie etwa Listenklassen und Klassen für Menüs und Threads sowie für die Grafikausgabe.
- Kapitel 5 behandelt die Einbindung einer Kylix-Anwendung in das Linux-System auf verschiedenen Ebenen, von der Kernel-Ebene bis hin zu KDE als Beispiel für einen grafischen Desktop: Es beschreibt, wie Sie mit Systemaufrufen die Funktionen des Kernels nutzen, wie Sie C-Bibliotheken in Kylix-Programmen aufrufen können, und wie auch mit C++-Bibliotheken eine Zusammenarbeit möglich ist.
- Kapitel 6 demonstriert anhand der Beispielanwendung *TreeDesigner 3.5* unter anderem die Entwicklung größerer, dokumentbasierter Anwendungen mit moderner Benutzerschnittstelle, Dokument-View-Konzept und mit der Ausgabe geräteunabhängiger Grafik auf Bildschirm und Drucker. Die CD-ROM enthält bereits einige Hierarchiegrafiken der CLX, die Sie mit dem TreeDesigner bearbeiten können.
- Kapitel 7 befasst sich mit Datenbankanwendungen. Es erläutert den Aufbau von Datenbanken, die Komponenten der DataCLX, über die Sie diese ansprechen, die

allgemeine Funktionsweise einer Datenbankanwendung in Kylix, und es zeigt verschiedene Beispiele der Programmierung mit Feldern und Indizes. Umsteiger von Desktop-Datenbanken unter Windows wie dBase und Paradox finden Informationen zur Portierung ihrer Datenbanken nach Interbase.

- Thema von Kapitel 8 ist die Entwicklung eigener Komponenten. Das Kapitel beschreibt die hinter den CLX-Komponenten stehenden Konzepte und demonstriert die Entwicklung eigener Komponenten anhand von nützlichen Beispielen wie einer History-Kombobox, einer Echtfarben-Farbpalette und einem Tastenkürzeleditor für Menüs und Aktionslisten.

Rezeptverzeichnis *R167*

R-Nummern in Überschriften wie die oben gezeigte beziehen sich auf die Nummern des Rezeptverzeichnisses in Anhang A. Dieses Verzeichnis gibt eine Übersicht über viele der in diesem Buch beschriebenen Lösungen und weist dabei eine zum Inhaltsverzeichnis alternative Gliederung auf. Es beschränkt sich auf die nach meiner Einschätzung interessantesten Rezepte und verzichtet auf die wiederholte Nennung von grundlegende Beschreibungen, die schon durch das Inhaltsverzeichnis schnell gefunden werden können. Ich hoffe, dass dieses gegenüber meinem Delphi-5-Buch geänderte Rezept-Auswahlverfahren das Verzeichnis für Sie zu einer nützlicheren Lektüre macht.

Feedback und Homepage

Mich hat es sehr gefreut, die Entstehung von Kylix verfolgen und mein erstes Buch zur Programmierung unter Linux schreiben zu können. Ihr Feedback zu diesem Buch, auch Fragen und Korrekturhinweise, sind unter der Adresse *leserkontakt@ewlab.de* herzlich willkommen. Wie schon mehrfach erwähnt, gibt es zu diesem Buch auch eine Homepage. Die Adresse lautet `ewlab.de/kylix/buch-homepage.html`. Neben den aktualisierten Infos zur Kylix Open Edition und zu Delphi 6 werden dort gegebenenfalls auch Fehlerkorrekturen und Erweiterungen der Beispielprogramme veröffentlicht. Und nun wünsche ich Ihnen nützliche Erkenntnisse aus diesem Buch sowie viel Spaß und viel Erfolg mit Kylix und Linux.

Bonn, im Juni 2001 *Elmar Warken*

1 Die visuelle Programmierumgebung

Dieses Kapitel soll Ihnen nicht nur einen praktischen Einstieg bzw. Umstieg in die Programmierumgebung von Kylix ermöglichen, sondern auch ein fundiertes Grundwissen über die Arbeitsweise der neuen Umgebung und die Funktionsweise von Kylix-Anwendungen vermitteln. Daher wechselt sich die praktische Einführung mit Grundlagenbeschreibungen ab.

Damit Sie mit diesem Kapitel auch einen Schnelleinstieg finden können, wurde das Kapitel so organisiert, dass Sie es auf zwei verschiedenen Routen durchqueren können (das Buch begrüßt die *Two-Way-Tools* von Kylix also mit einem *Two-Way-Tutorial*):

- Die große Route: Sie lesen das Kapitel von Anfang bis Ende durch und überspringen dabei nur wenige Themen, die Sie später nachschlagen wollen. Auf diese Weise werden Ihre praktischen Erfahrungen mit Kylix von umfangreichem Grundlagenwissen untermauert und Sie erhalten gleichzeitig einen guten Überblick über die gesamte Kylix-IDE, ohne dass Sie sich mit jedem einzelnen Menüpunkt beschäftigen müssen.
- Die kleine Route: Sie beginnen sofort mit Kapitel 1.3. Nachdem Sie dieses über den Entwicklungszyklus für eine Kylix-Anwendung informiert hat, finden Sie dort einen kurzen Überblick über die Entwicklung eines Beispielprogramms. Sie besuchen daraufhin die in diesem Überblick beschriebenen Stationen des Kapitels, um die vollständige Entwicklung dieses Programms beobachten oder mitmachen zu können.

Wir wenden uns nun den wichtigen Grundlagen zu, auf denen jede Kylix-Anwendung aufbaut – Willkommen auf der großen Route! Wenn Sie Kylix zum ersten Mal starten, zeigt es Ihnen ein leeres Formular, das Sie durch Hinzufügen von Komponenten zur Oberfläche Ihres Programms ausbauen können. An der Oberfläche dreht sich also alles um Formulare und Komponenten. Für die Entwicklung von Anwendungen ist jedoch die grundsätzliche interne Funktionsweise einer Kylix-Anwendung am wichtigsten, weshalb wir auch genau dort beginnen.

1.1 Kylix-Anwendungen sind ereignisorientiert

Moderne Anwendungen auf grafischen Oberflächen arbeiten ereignisorientiert, das heißt, sie laufen nicht nach einer festgelegten Reihenfolge ohne Eingriffsmöglichkeiten ür den Benutzer ab wie etwa ein einfaches Shell-Skript.

Eine optimal entworfene Kylix-Anwendung tut nichts anderes, als auf Ereignisse (Events) zu reagieren. Diese Ereignisse können vom Betriebssystem, von anderen Anwendungen oder von der Kylix-Anwendung selbst (beispielsweise von der Komponentenbibliothek CLX) erzeugt werden.

Die Ereignisse haben zunächst je nach ihrem Entstehungsort unterschiedliche Erscheinungsbilder: Von der grafischen Oberfläche generierte Ereignisse erreichen die Anwendung beispielsweise in Form von Nachrichten, und Ereignisse der CLX hängen oft mit *virtuellen Methoden* zusammen. Als Verwender von Kylix macht dies für Sie jedoch keinen Unterschied, denn die CLX formt fast alle Nachrichten in gleichartige (aber nicht identische) *Events* um, die Sie im Objektinspektor sehen und bearbeiten können.

Sie kommen zwar normalerweise mit den von der CLX bereitgestellten Events aus, jedoch steht Ihnen unter Kylix auch der Weg zu den ursprünglichen Nachrichten offen. Wenn Sie beispielsweise selbst Komponenten programmieren oder auf Nachrichten des Systems reagieren wollen, für die es im Objektinspektor kein Event gibt, können Sie andere Möglichkeiten der Ereignisbearbeitung nutzen. Eine detaillierte Darstellung des Nachrichtenflusses in einer Kylix-Anwendung finden Sie in Kapitel 3.1.4.

1.1.1 Zu jedem Anlass ein Ereignis

Falls Sie bisher nicht ereignisorientiert programmiert haben und nicht davon überzeugt sind, dass Ereignisse genügen, um Ihre Anwendung zu steuern, stellen Sie sich alle Aktionen vor, die Ihnen bekannte Anwendungen ausführen und die Ihre eigene Anwendung möglicherweise ausführen könnte. Jede dieser Aktionen muss durch gendetwas ausgelöst werden. Dieser Auslöser wird in ereignisgesteuerten Anwendungen als *Ereignis* bezeichnet. Jeder Auslöser bzw. jedes Ereignis sollte in einer der Kategorien zu finden sein, die im Folgenden aufgezählt sind.

Befehlsereignisse: Die wichtigsten Operationen wie das Öffnen und Speichern von Dateien oder das Aufrufen von Dialogboxen werden vom Anwender über Menüs oder Schalter gestartet. Die Qt-Bibliothek, mit der Delphi-Anwendungen arbeiten, erzeugt ein spezielles Signal, wenn der Anwender auf diese Art einen Befehl aufgerufen hat. Die CLX wandelt dieses Signal in ein normales Event um – Events für Befehlsereignisse haben meistens den Namen *OnClick*, der darauf zurückzuführen ist, dass der Benutzer ein Element (Menü, Schalter etc.) angeklickt hat. Es spielt jedoch keine Rolle, ob das Element tatsächlich angeklickt oder mit der Tastatur ausgewählt wurde.

- *Eingabeereignisse*: Weitere Ereignisse sind die Eingaben, die der Benutzer mit der Maus oder der Tastatur vornimmt, doch nicht alle dieser Eingaben erreichen Ihre Anwendung. So wird z.B. die gesamte Menüsteuerung per Tastatur im Falle einer Qt-Anwendung wieder von Qt übernommen. Sobald der Benutzer einen Menüpunkt aufruft, erhält die Anwendung eine Befehlsnachricht des oben genannten Typs. Eingabeereignisse, die Sie vielleicht direkt bearbeiten möchten, sind Aktionen der Maus in einem Zeichenbereich Ihrer Anwendung. Zu den Mausereignissen gehören nicht nur Klicks, sondern jede kleine Bewegung der Maus ist ein eigenes Ereignis, das Sie bearbeiten können. Auch Tastatureingaben in den Arbeitsbereich Ihres Fensters bzw. Formulars gehören zu den Eingabeereignissen.
- *Fensterereignisse*: Um beim Öffnen und Schließen des Fensters (bzw. beim Starten und Beenden des Programms) bestimmte Initialisierungs- oder Freigabeaktionen durchführen zu können, müssen Sie die zum Öffnen und Schließen gehörenden Ereignisse abfangen.
- *Timer-Ereignisse*: Soll Ihr Programm beispielsweise in regelmäßigen Abständen automatische Sicherheitskopien anlegen, kann es hierzu sicher keine Eingaben des Benutzers als auslösendes Ereignis verwenden. Für derartige Zwecke finden Sie in der Komponentenpalette eine Timer-Komponente, deren *OnTimer*-Events in frei wählbaren konstanten Intervallen generiert werden.
- *Andere Systemereignisse*: Im Zusammenhang mit der Kommunikation zwischen verschiedenen Anwendungen sowie zwischen verschiedenen Linux-Komponenten und einer Anwendung kommt es ebenfalls zu vielen Ereignissen: Wenn Sie z.B. in der Systemkonfiguration Farbeinstellungen ändern, werden alle Anwendungen davon benachrichtigt, damit sie ihre Farben entsprechend anpassen können.
- Die Menge der Ereignisse ist keineswegs festgelegt: Unter den Events, die Sie im Objektinspektor aufgelistet finden, sind auch einige, die nicht auf Ereignisse der grafischen Oberfläche zurückzuführen sind, sondern direkt von der CLX generiert werden (z.B. alle Events der Datenbankkomponenten, wie sie etwa anlässlich der verschiedensten Datenbankoperationen entstehen). Wenn Sie eigene Komponenten entwickeln, können Sie selbst neue Events definieren.

Selbst ein Shell-Skript, das nur aus einer Liste von einfachen Befehlen besteht, können Sie in dieses Ereignisschema pressen. Danach ist das Skript ein Programm, das nur auf das Ereignis, gestartet zu werden, reagiert und als Reaktion darauf alle Anweisungen der Reihe nach ausführt und schließlich endet.

Die benutzerabhängigen Ereignisse können jederzeit stattfinden, nur die Timer-Ereignisse müssen Sie in Ihrem Programm selbst anfordern. Es macht jedoch nichts, wenn Sie für ein Ereignis keine spezielle Antwort vorsehen: Das Ereignis wird dann entweder ignoriert oder es wird eine Standardaktion ausgeführt. Die Standardbearbeitung von

Ereignissen sorgt z. B. dafür, dass Sie ein leeres Formular, für das Sie noch keine einzige Zeile Code geschrieben haben, wie jedes andere Fenster verschieben, manipulieren und schließen können und dass die Befehle in seinem Systemmenü funktionieren.

Hinweis: Im Folgenden wird die strenge Unterscheidung zwischen Ereignissen im Allgemeinen und den Events, die im Objektinspektor aufgelistet werden, nicht mehr weiter durchgeführt. Die folgenden Abschnitte des ersten Kapitels befassen sich nur noch mit den Events des Objektinspektors, die von nun an ebenfalls als »Ereignisse« bezeichnet werden.

1.1.2 Ereignisse in Kylix

Um einen ersten praktischen Eindruck von Kylix' Ereignissen zu bekommen, ohne gleich ein Programm zu schreiben, können Sie die Seite *Ereignisse* des Objektinspektors aufschlagen. Wenn Sie Kylix neu gestartet haben und sich wie in Abbildung 1.1 ein leeres Formular auf dem Bildschirm befindet, dann zeigt der Objektinspektor die Ereignisse, die Sie für dieses Formular bearbeiten können. Dazu gehören beispielsweise die folgenden:

OnActivate tritt auf, wenn das Formular in den Vordergrund geholt wird.

OnClick zeigt Ihnen an, dass der Benutzer des Programms in das Innere des Formulars geklickt hat.

OnCreate benachrichtigt Sie davon, dass das Fenster gerade erzeugt worden ist.

- *OnCloseQuery* tritt auf, bevor das Formular geschlossen wird. Wenn die Datei, die im Formular bearbeitet wird, noch nicht gespeichert wurde, können Sie bei dieser Gelegenheit den Benutzer fragen, ob er die Datei speichern oder ob er den Vorgang abbrechen will.

Jede Komponente, die Sie in das Formular einfügen, verfügt über eine individuelle Auswahl an Ereignissen. Um das auszuprobieren und eine Komponente in das Formular einzufügen, klicken Sie mit der Maus auf irgendein Symbol der Komponentenpalette und dann einmal in das Formular. Haben Sie sich beispielsweise für die Schalterkomponente (*TButton*) entschieden, so finden Sie im Objektinspektor nun eine andere Ereignisauswahl. Die meisten Ereignisse, wie z. B. *OnClick*, sind auch beim Formular vorhanden, insgesamt haben Schalter aber weit weniger Ereignisse. Die Ereignisse *OnEnter* und *OnExit* finden Sie dagegen nicht im Formular. Sie werden ausgelöst, wenn der Schalter den Tastaturfokus erhält (*OnEnter*) bzw. verliert. (Ein normaler Qt-Schalter zeigt seine Fokussierung dadurch an, dass er den Schaltertext mit einer gepunkteten Linie umrandet. Mit den Pfeiltasten und der Tabulatortaste können Sie in einer Dialogbox normalerweise jeden Schalter ansteuern. Weitere Informationen zum astaturfokus liefert Kapitel 3.3.2.)

Ereignisbearbeitung in verschiedenen Systemen

Während das Konzept der Ereignisse weit verbreitet ist, gibt es doch viele verschiedene Ansätze, es in einer Programmiersprache zu implementieren. Auf Betriebssystem-Ebene werden Ereignisse z.B. häufig in standardisierten Nachrichtenpaketen an eine bestimmte, für die Nachrichtenbearbeitung vorgesehene Programmfunktion übermittelt (X-Window-Programme holen sich diese Pakete z.B. selbst ab). Diese Programmfunktion muss dann zunächst einmal feststellen, um welches Ereignis es sich überhaupt handelt, und dann gegebenenfalls andere, spezielle Funktionen aufrufen.

Objektorientierte Systeme verwenden ausgefeiltere Konzepte und können die Ereignisse beispielsweise zunächst nach dem Objekt (z.B. Fenster, Steuerelement), das für das Ereignis zuständig ist, unterscheiden. Dieses Objekt könnte wiederum für jeden Ereignistyp eine eigene Methode bereitstellen.

Die auf X Window aufbauende Qt-Bibliothek verwendet das besonders flexible Konzept der *Signals* und *Slots*. Hierbei werden Ereignisse (die Signals) direkt mit speziellen dafür zuständigen Funktionen (den Slots) verbunden. Dabei kann jedes einzelne Signal mit einem eigenen Slot verbunden werden, oder aber auch ein Signal mit mehreren Slots oder mehrere Signale mit dem gleichen Slot. Außerdem ist nicht von vornherein festgelegt, welche Objekte ein Ereignis bearbeiten. So kann beispielsweise die Betätigung eines Schalters vom Schalterobjekt selbst, vom Objekt des Fensters, in dem sich der Schalter befindet, oder von einem beliebigen anderen Objekt bearbeitet werden.

Theoretisch könnte man das Ereigniskonzept von Kylix in das Konzept der Signals und Slots von Qt einbetten, ja sogar dadurch erweitern (die visuelle Ereignisverknüpfung in Kylix sieht z.B. bisher noch keine Verknüpfung eines Ereignisses mit mehreren Methoden wie Qt vor!) In der C++-Praxis werden Signals und Slots jedoch manuell über C++-Anweisungen im Programmcode verknüpft, was wesentlich umständlicher ist als die Ereignisverknüpfung über den Objektinspektor von Kylix.

Vorteile der Ereignisbearbeitungsweise in Kylix

Das auffälligste Merkmal der Ereignisbearbeitung unter Kylix ist sicher der komfortable Verknüpfungsmechanismus von Ereignis und Ereignisbearbeitungsmethode. Um Ereignisse zu bearbeiten, müssen Sie zwar immer noch eine Ereignisbehandlungsmethode (einen Event-Handler) schreiben, Kylix erstellt jedoch automatisch ein Gerüst dafür und erspart Ihnen, an anderen Stellen des Programmcodes für den korrekten Aufruf dieses Ereignisses zu sorgen.

Die Flexibilität, ein Ereignis von einem beliebigen Objekt bearbeiten zu lassen, ist auch unter Kylix gegeben, allerdings können Sie über den Objektinspektor nur Methoden eines Formulars mit den Ereignissen verknüpfen. Dies ist in der Praxis aber keine große Einschränkung, bedeutet dafür allerdings eine große Vereinfachung beim Erlernen der

Kylix-Programmierung, insbesondere für Entwickler, denen die objektorientierte Programmierung noch fremd ist. Wenn Sie etwa durch die bisherigen Ausführungen über verschiedene Objekte, die ein Ereignis bearbeiten können«, verwirrt sein sollten, so önnen Sie diese Bemerkungen beruhigt vergessen, denn Sie brauchen sich bei der grundlegenden Ereignisbearbeitung gar nicht bewusst zu sein, dass das Ereignis von einem »Objekt« bearbeitet wird, es genügt für den Anfang, wenn Sie die Methode (alias Funktion, Unterprogramm) kennen, die (das) das Ereignis bearbeitet.

ür Eingeweihte der OOP sei aber schon gesagt: Verknüpfungen zu Methoden anderer Objekte als des Formulars können Sie auch in Kylix durch Anweisungen im Programmcode vornehmen (einfache Zuweisungen genügen, siehe z. B. R59 Seite 155). Die atsache, dass alle Ereignisse eines Fensters vom Formular bearbeitet werden, führt dazu, dass die Deklaration der Formularklasse, ja die ganze Quelltextdatei mit den Methodenimplementationen etwas unübersichtlich werden kann. Falls Sie hier eine ärkere Kapselung/Modularisierung wünschen, können Sie möglicherweise das Konzept der Frames nutzen und ein Formular mit verschiedenen eigenständigen Teilbereichen in Frames aufspalten (siehe Kapitel 3.7.2). Jedes Frame verfügt über eine eigene Klasse, über einen eigenen Satz von Methoden und die Ereignisse der Komponenten eines Frames können sogar mit zwei Methoden gleichzeitig verknüpft werden: mit Methoden des Frames und mit Methoden des Formulars (damit wird die oben erwähnte Beschränkung »ein Ereignis – eine Methode« natürlich etwas aufgeweicht).

1.2 Von Formularen und Komponenten

Neben den Ereignissen spielen Formulare und Komponenten die zweite Hauptrolle in einer Kylix-Anwendung. Doch werfen wir zunächst einen Blick auf die Fensterstruktur einer allgemeinen Anwendung mit grafischer Benutzerschnittstelle, die sich natürlich auch bei einer Kylix-Anwendung wiederfinden lässt.

1.2.1 Die Fensterhierarchie einer GUI-Anwendung

In den meisten Programmierumgebungen sind Fenster die Hauptbausteine einer Anwendung. Haben Sie schon mit dieser Sichtweise gearbeitet, werden Sie sich vielleicht fragen, welcher Zusammenhang zwischen Formularen, Komponenten und Fenstern besteht. Für die Leser, die mit der Fensterhierarchie von GUI-Anwendungen noch nicht vertraut sind, sei diese noch einmal kurz erläutert, denn sie ist natürlich auch bei Kylix vorhanden:

Aus der Sichtweise des Programmierers gelten nicht nur diejenigen Rechtecke als Fenster, die eine Titelzeile und einen mit der Maus manipulierbaren Rahmen besitzen. Auch kleinere Bauteile, die etwa als Steuerelemente in Dialogboxen vorkommen, wie verschiedenartige Schalter, Eingabefelder, Listen usw., werden unter X Window als

Fenster behandelt. Alle Fenster sind in einer Eltern-Kind-Fensterhierarchie angeordnet, d.h. ein Fenster kann mehrere Kindfenster haben und jedes Fenster kann ein Elternfenster haben. Die Hauptfenster einer Anwendung sind (auf X-Window-Ebene, nicht aus der Sicht einer Kylix-Anwendung!) einem Wurzelfenster (root window) untergeordnet, das den gesamten Bildschirmhintergrund ausfüllt. Um eine spätere Verwirrung zu vermeiden, sollten Sie diese Beziehung der Fenster immer als Eltern-Kind-Beziehung ansehen, denn das *Besitzen* von Fenstern ist ein anderes Thema, das sich in diesem Buch speziell auf einen Mechanismus der CLX bezieht und in Kapitel 3.2.1 erörtert wird.

Drei der am häufigsten vorkommenden Fensterkonstruktionen sind:

- *Hauptfenster*: Das Hauptfenster einer einfachen Anwendung hat als Unterfenster oft eine Mauspalette und eine Statuszeile. Zwischen diesen befindet sich ein Bereich, in dem das Programm Informationen darstellt, wie z.B. den Inhalt der Festplatte oder einer Datenbank. Oft kann in diesem Bereich auch ein Dokument bearbeitet werden (z.B. Text oder Grafik). Der Bereich, der nach Abzug von Mauspalette, Statuszeile und eventuell weiteren an den Rändern angebrachten Steuerelementen noch frei bleibt, heißt auch *Arbeitsbereich* oder *Client-Bereich* (*Client-Area*).
- *Dialogboxen*: Eine Dialogbox ist das Elternfenster all seiner Steurerelemente, und auch Elemente, die nur der optischen Gestaltung dienen, wie z.B. eine dreidimensionale Hervorhebung unter einer Schaltergruppe, können als eigene Unterfenster realisiert sein.
- *MDI-Anwendungen*: Diese zeichnen sich dadurch aus, dass mehrere Dokumente (oder andere Informationen) in verschiedenen Fenstern gleichzeitig bearbeitet werden können. Statuszeile und Symbolleiste bleiben im Hauptfenster und der Client-Bereich wird zu einer Fläche, auf der die MDI-Kindfenster angeordnet werden (jedes dieser Kindfenster hat einen eigenen Client-Bereich).

Mit Kylix können Sie Ihre Anwendung auf die drei oben genannten und auf viele weitere Arten strukturieren. Ein Beispiel für eine relativ lockere Verknüpfung einzelner Fenster zu einer Anwendung (ohne diese Fenster in einem gemeinsamen Hauptfenster darzustellen) gibt die Kylix-Oberfläche selber. Zwar gibt es auch hier ein Hauptfenster mit Menü und Schalterleisten, die darunter angeordneten Fenster Objektinspektor, Formular und Quelltexteditor (in Abbildung 1.1 verdeckt) sind jedoch davon getrennt und können unabhängig voneinander auf dem gesamten Bildschirm bewegt werden.

1.2.2 Formulare und Fenster

In Kylix wird eine etwas verfeinerte Terminologie verwendet: Die Fenster der obersten Ebene (Anwendungsfenster, Dialogboxen), allgemein auch Rahmenfenster (solche können beispielsweise im Falle von MDI-Kindfenstern auch unter der obersten

Fensterebene geschachtelt werden) werden als *Formulare* behandelt und können im Formular-Editor entworfen werden. Die kleineren Bauteile der Formulare sind *Komponenten.*

Wie Sie am Beispiel einer MDI-Anwendung sehen können, ist es nicht sinnvoll, jedes Fenster, das bei Verwendung der Anwendung benötigt wird, einzeln zu entwerfen, denn bei MDI-Anwendungen ist die Zahl der Kindfenster nicht vorhersehbar und theoretisch unbegrenzt. Daher ist es wichtig, die Formulare, die Sie in Kylix entwerfen, als Formulare oder Schablonen zu verstehen, nach denen später beliebig viele »echte« Fenster produziert werden können.

Da Kylix dafür sorgt, dass beim Programmstart automatisch ein Fenster nach Muster des Hauptformulars angelegt und aktiviert wird, kann dieses Fenster leicht mit dem Formular selbst verwechselt werden. Im objektorientierten Sprachgebrauch ist das Fenster jedoch eine *Instanz* einer Fenster*klasse*, die Sie durch das Formular definiert haben.

Bei Untersuchung der CLX-Klassenhierarchie werden Sie schnell feststellen, dass die Klasse für die Formulare, *TForm*, von der Komponenten-Klasse, *TComponent*, abgeleitet ist. Ein Formular *ist* also eine Komponente. Sie taucht jedoch nicht in der Komponentenpalette auf, das heißt, Formulare können nicht aus »Unterformularen« zusammengesetzt werden. Wenn Sie aus irgendeinem Grund gerne ein bestehendes Formular wie eine Komponente beim Entwurf eines neuen Formulars verwenden wollen, bieten sich jedoch Frames als Lösung an (siehe Kapitel 3.7.2).

1.2.3 Komponenten

Die Bestandteile des Fensters, die in anderen Programmiersprachen auch als Steuerelemente, Dialogelemente, Controls etc. bezeichnet werden, tauchen in Kylix als Komponenten wieder auf. Komponenten sind jedoch ein noch viel weiter gehendes Konzept, denn Komponenten müssen nichts mit Qt-Steuerelementen zu tun haben und sie können auch unsichtbar sein.

Wie Kapitel 3 genauer erläutern wird, entsprechen die verschiedenen Komponententypen einem Teil der Klassenhierarchie der Kylix. Tatsächlich handelt es sich bei den Komponententypen um Object-Pascal-Klassen, die speziell zur Bearbeitung mit den Kylix-Tools (z. B. dem Objektinspektor) vorbereitet sind. Mehr dazu erfahren Sie in Kapitel 8.

omponenten und Fenster

Wie erwähnt, gelten auf Programmier-Ebene auch kleinere Fensterteile wie Schalter und Listen als eigene Fenster. Unter Kylix müssen Sie sich zunächst keine Gedanken darüber machen, ob die Komponenten, mit denen Sie Schalter, Listen und andere Ele-

mente erzeugen, auch »echte« Qt-Fenster sind. Meistens ist das zwar der Fall, manche Komponenten wie z.B. die *Label*-Komponente sind jedoch keine Qt-Fenster. Als fortgeschrittener Kylix-Programmierer finden Sie in Kapitel 3.1.3 genaue Informationen zu diesem Thema.

1.3 Der Entwicklungszyklus

(Ich darf an dieser Stelle herzlich die Leser begrüßen, die sich für die kleine Route durch Kapitel 1 entschieden haben und hier mit dem Lesen beginnen.)

1.3.1 Der Entwicklungszyklus in der Kylix-IDE

Die Entwicklung einer einfachen Kylix-Anwendung besteht meistens aus den folgenden Schritten (wobei Sie jederzeit zu einem früheren Schritt zurückkehren können, um beispielsweise neue Komponenten hinzuzufügen):

- 1. Auswahl und Hinzufügen der Komponenten
- 2. Einstellung der Komponenten-Properties
- 3. Schreiben der Ereignismethoden
- 4. Starten der Anwendung (Testen, Fehlersuche etc.)

Zwar kann auch ein leeres Fenster Ereignisse empfangen, jedoch werden Sie wahrscheinlich zuerst Kontrollelemente wie Menüs, Schalter und Eingabefelder darin unterbringen wollen. Dazu wählen Sie in Schritt 1 die gewünschten Elemente aus der Komponentenpalette und zeichnen sie in das Formular ein. Je komplexer die Anwendung ist, desto mehr ist es übrigens notwendig, die Benutzerschnittstelle schon hier sorgfältig zu planen, damit sie trotz der Komplexität übersichtlich und logisch bleibt.

Sobald Sie eine Komponente hinzugefügt haben, können Sie als zweiten Schritt im Objektinspektor zahlreiche Feineinstellungen vornehmen, indem Sie die Attribute der Komponenten verändern, z.B. die Hintergrundfarbe, den Schaltertext usw.

Die hinter der Oberfläche befindliche Funktionalität bringen Sie im dritten Schritt als Reaktion auf die passenden Ereignisse unter. Dazu wählen Sie aus der zweiten Seite des Objektinspektors die Ereignisse aus, auf die Ihre Anwendung reagieren soll. Kylix führt Sie in den Code Editor an die Stelle, an der Sie die Anweisungen für das Ereignis eingeben können.

An jedem Punkt dieses Prozesses können Sie die vorläufige Anwendung von Kylix ausführen lassen, sofern sich keine Fehler im Programmcode befinden. Wenn Sie noch keine Ereignisbearbeitung festgelegt haben, erhalten Sie ein nutzloses, aber funktionsfähiges Fenster: Sie können darin schon die Standardaktionen ausprobieren (so lässt

sich in ein Editierfeld Text eintragen, der dort allerdings ein völlig sinnloses Dasein fristen muss, es sei denn, Sie kopieren ihn mit dem Standard-Tastenkürzel oder über das Kontextmenü des Editierfelds in die Zwischenablage).

Darüber hinaus können Sie nahezu beliebig zwischen den Schritten wechseln. Bei groen Anwendungen ergeben sich sicher noch während der Entwicklung neue Aufgaben ür weitere Komponenten, die Sie problemlos nachträglich hinzufügen können. Auch alle Eigenschaften und Ereignisse im Objektinspektor bleiben veränderbar. Das Einzige, was im aufgelisteten Entwicklungszyklus festgelegt ist, ist, dass Sie eine Komponente in das Formular eingefügt haben müssen, bevor Sie deren Eigenschaften und Ereignisbehandlung über den Objektinspektor ändern können.

Ein fester Bestandteil des Entwicklungszyklus' bei einem größeren Programm ist natürlich auch das Testen der Anwendung und die Korrektur von Fehlern. Zum dafür vorgesehenen integrierten Debugger kommen wir in Kapitel 1.7.

Nicht weiter behandelt werden soll in diesem Buch der Entwurf von Software mit Hilfe von Methoden des Software-Engineerings, hierfür gibt es eigene umfangreiche, Programmiersprachen-unabhängige Literatur. Der oben beschriebene Entwicklungszyklus lässt sich jedenfalls problemlos mit verschiedenen Techniken des Software-Entwurfs kombinieren: Auch wenn der Funktionsumfang und das Aussehen einer Anwendung sorgfältig im Voraus geplant wurden, können sich durch die praktische Arbeit mit einem ersten Kylix-Formular-Prototyp leicht neue Ideen ergeben, die ein nachträgliches Hinzufügen von Komponenten empfehlenswert erscheinen lassen.

1.3.2 Übersicht über das Beispielprogramm

Die folgenden Abschnitte 1.4 und 1.5 erläutern den praktischen Umgang mit Formulaen, Komponenten und Ereignissen in der Kylix-IDE. Dabei beschäftigen wir uns auch mit einem kleinen Beispielprogramm, das in mehreren Schritten aufgebaut wird, wobei diese Schritte grob den einzelnen Stationen des oben beschriebenen Entwicklungszyklus' entsprechen. Das Programm selbst soll einen kleinen Editor für eine Linux-Systemdatei darstellen (dieser Editor ist fest auf die Datei `.bashrc` voreingestellt, dies kann jedoch im Programmcode leicht geändert werden). Damit stellt es zwar keine innovative, aber doch eine praktisch verwendbare Funktion zur Verfügung.

Sie können aus den nun folgenden Abschnitten auch eine kurze Praxiseinführung machen, wenn Sie nur die Abschnitte lesen, die sich mit dem Beispielprogramm beschäftigen, und die anderen Abschnitte später bei Bedarf lesen. Die »kurze Route« durch dieses Kapitel verläuft über die folgenden Stationen, die sich mehrheitlich mit dem Beispielprogramm beschäftigen:

- Kapitel 1.4.4, *Entwerfen von Formularen*. Im Abschnitt *Zum Beispielprogramm* (ab Seite 42) bauen Sie das Formular des Beispielprogramms auf, noch ohne die Properties zu setzen.
- Kapitel 1.4.6, *Der Objektinspektor*. Im Abschnitt *Properties für das Beispielprogramm* werden die Properties des Beispielprogramms eingestellt und erläutert.
- In Kapitel 1.5.1 geht es zwar nicht um das Beispielprogramm, die hier praktisch demonstrierten Techniken der Ereignisbearbeitung sind aber Voraussetzung für die übernächste Station.
- Für Pascal-Neulinge: Die Schnellübersicht in Kapitel 1.5.2 soll Ihnen, falls Sie sich mit Object Pascal oder einer ähnlichen Sprache wie C++ noch nicht auskennen, einen Überblick über die Aktionsmöglichkeiten im Innern einer Methode geben.
- In Kapitel 1.5.4 wird das Beispielprogramm, das bisher zwar schon ein editierbares Textfeld enthält, mit seiner eigentlichen Funktion versehen: dem Editieren der Konfigurationsdatei `.bashrc`. Ereignisbearbeitungsmethoden bereichern das Programmfenster durch kleine Konfigurationsmöglichkeiten wie das Wählen einer Hintergrundfarbe und einer Schriftart für den editierten Text.
- Kapitel 1.5.5 fügt dem Programm weitere Ereignisbearbeitungsmethoden hinzu, die der Aktualisierung einer provisorischen Statuszeile dienen (provisorisch meint hier, dass dazu nicht die *TStatusBar*-Komponente verwendet wird, weil das Programm ganz mit den Komponenten der Standard-Seite auskommen soll).
- In Kapitel 1.5.6 wird das Programm durch ein verbessertes Fenster-Layout und eine Sicherheitsabfrage zum Speichern der Datei abgerundet. Für die Änderung des Layouts werden zunächst wieder die Schritte 1 und 2 des Entwicklungszyklus durchlaufen, um neue Komponenten im Formular einzurichten.

Damit ist das erste Beispielprogramm, das Sie in Kylix mit DATEI | PROJEKT ÖFFNEN... auch fertig von der CD-ROM laden können, abgeschlossen. (Zum Laden des Projekts von der CD mounten Sie diese und wählen dann im PROJEKT ÖFFNEN...-Dialog das Projekt *BashRCEditor* aus dem Verzeichnis *Kapitel 1*). Weitere Beispielprogramme werden im abschließenden Teil 1.8 des ersten Kapitels folgen.

Die anschließenden Abschnitte 1.6 und 1.7 haben auch ohne Beispielprogramm einen sehr anschaulichen Inhalt, denn sie machen einen ausführlichen Rundgang durch die wichtigsten Einrichtungen der IDE wie Editor, Projektverwaltung, Objektablage, Browser und Debugger. Pascal-Neulinge brauchen nicht alle der dort beschriebenen Dinge sofort zu verstehen, sondern können sich anhand dieser Beschreibungen einen Überblick darüber verschaffen, was ihnen die Kylix-Entwicklungsumgebung noch alles zu bieten hat.

1.4 Die IDE und die visuellen Werkzeuge

Wir beschäftigen uns nun konkret mit den Bestandteilen der Kylix-Oberfläche (IDE, Integrated Development Environment) und beginnen dabei mit dem Beispielprogramm des Kapitels. Für den Rundgang durch die IDE und das Beispielprogramm benötigen wir zuerst ein neues Projekt mit einem leeren Formular. Wenn Sie ein neu installiertes Kylix starten, legt dieses automatisch ein solches an. Um später nach beliebigen anderen Arbeitsschritten zu diesem Zustand zurückzukehren, wählen Sie aus dem Hauptmenü den Punkt DATEI | NEUE ANWENDUNG. Sie erhalten dann eine Bildschirmanzeige ähnlich der in Abbildung 1.1 gezeigten.

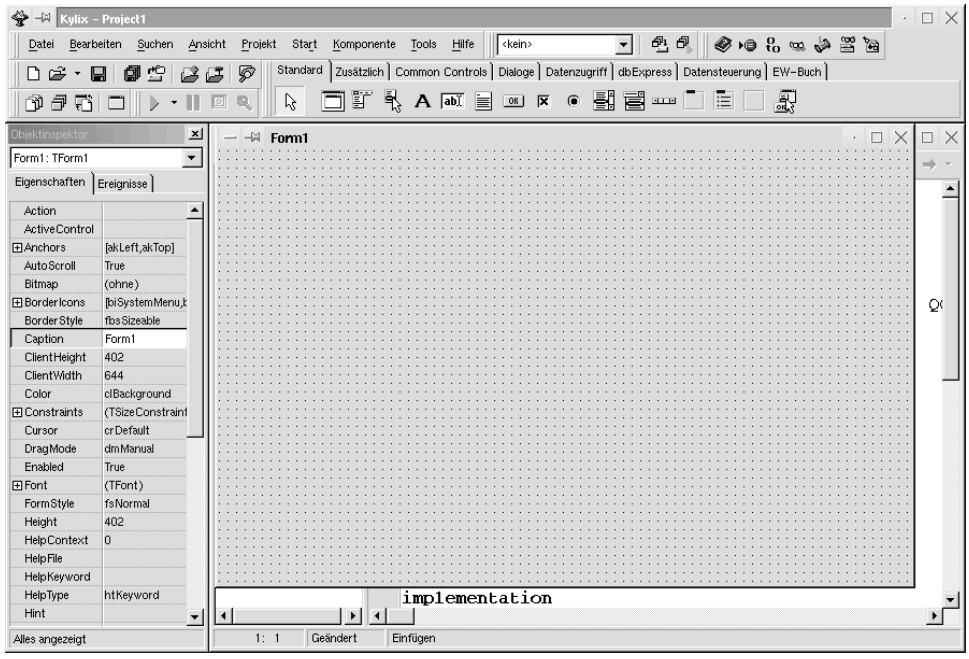


Abbildung 1.1: Ein neues Projekt

1.4.1 Der Aufbau der IDE

Die Kylix-IDE besteht aus mehreren Fenstern, die direkt auf dem Desktop liegen, also kein gemeinsames übergeordnetes Elternfenster haben: Im oberen Teil des Bildschirms befindet sich per Voreinstellung das Hauptfenster, das nur aus dem Menü, ein paar Symbolleisten und der Komponentenpalette besteht. Andere Fenster, darunter die Formulare, werden unter diesem Hauptfenster angeordnet. Symbolleisten und Komponentenpalette können aus dem Hauptfenster herausgezogen und als separate Leisten ei auf dem Bildschirm platziert werden. Auch können Sie die Fenster, die sich unter

dem Hauptfenster befinden, flexibel aneinander docken und übereinander stapeln. Wenn Sie die weiter unten beschriebene Autospeichern-Option einschalten, speichert Kylix für jedes Projekt ein separates Fenster- und Docking-Layout, das bei jedem Öffnen des Projekts wiederhergestellt wird. Sie können sogar verschiedene Layouts mit Namen benennen und über die neben der Menüleiste befindliche aufklappbare Liste beliebig zwischen ihnen wechseln.

Hinweis: Sollte die IDE einmal durch einen Fehler in ihrer weiteren Funktionsweise beeinträchtigt werden (was in der Version 1.0 in Verbindung mit dem Mitte 2001 aktuellen Linux-Loader wahrlich nicht ausgeschlossen werden kann), lässt sie sich aus einem Terminal-Fenster mit dem Befehl »`killall Kylix`« schnell aus dem Speicher entfernen (vorheriges Speichern des Projekts nicht vergessen!) und mit `startkylix -ns &` ohne Anzeige der Intro-Grafik wieder neu starten. In den Erlebnissen des Autors kamen solche Fehler meist nur in einer speziellen Situation vor: Wenn die Ausführung eines Programms im Debugger zu Ende ist, gelingt es der IDE nicht immer, das Programm auch richtig zu beenden und die IDE muss neu gestartet werden, damit man den Debugger wieder verwenden kann. Mit einem ständig offenen Terminal-Fenster, in dem sich die zuletzt ausgeführten Befehle schnell mit ↑ abrufen lassen, ist das ein Aufwand von wenigen Sekunden. (Der Autor meint: Was ist das schon, wenn einem Kylix vorher eine halbe Stunde mühsamer Oberflächenprogrammierung erspart hat?)

Lokale Menüs

Das Hauptmenü steht nicht nur mit der Symbolleiste in Konkurrenz, sondern auch mit den lokalen Menüs (auch als Popup- oder Kontextmenüs bezeichnet). Alle Kylix-Fenster verfügen über derartige Menüs, die Sie mit der rechten Maustaste aufrufen. Kylix listet im Popup-Menü nur Funktionen auf, die sich auf das angeklickte Fenster beziehen. Jedes Fenster verfügt über ein völlig eigenes Popup-Menü.

Falls Sie schon mehrere andere Programme mit Kontextmenüs verwendet haben, fragen Sie sich sicher, welches Konzept bei Kylix zugrunde liegt. Sie werden feststellen, dass der Inhalt der lokalen Menüs nicht streng logisch begründet ist, es sind also weder alle möglichen Aktionen für das Fenster darin enthalten, noch gibt es für alle lokalen Menüpunkte einen gleichwertigen globalen Hauptmenüpunkt (z. B. nicht für das Konfigurieren der Symbolleiste).

Am besten ist es, die verschiedenen lokalen Menüs einfach auszuprobieren, denn sie enthalten viele bequeme Abkürzungen: Statt beispielsweise die Dialogseite TOOLS | UMGEBUNGSOPTIONEN | PALETTE über das Hauptmenü und das Seitenregister aufzurufen, können Sie aus dem lokalen Menü der Komponentenpalette den Punkt EIGENSCHAFTEN aufrufen. Schließlich lassen sich manche besonders selten benötigte Funktionen nur in lokalen Menüs finden, z. B. HINWEISE ZEIGEN im Menü der Komponentenpalette und NUR LESEN im Menü des Editors.

enstermanagement und Docking

Durch den Verzicht auf das MDI-Konzept verliert die IDE auch ein FENSTER-Menü, das in MDI-Anwendungen verschiedene Möglichkeiten bietet, die Fenster anzuordnen. Allerdings sind solche Anordnungsbefehle wie NEBENEINANDER oder ÜBEREINANDER bei der neuen Struktur der IDE nicht mehr so wichtig, denn das Editorfenster ordnet standardmäßig alle Quelltextdateien in einem Seitenregister an und der Objektinspektor bleibt normalerweise an seiner Position. Unübersichtlich kann die Lage dann nur noch durch viele geöffnete Formulare werden. Zwischen diesen können Sie über die Liste, die Sie mit ⇧+F12 erhalten (ANSICHT | FORMULARE), wechseln. ANSICHT bietet noch einige weitere Optionen zur Organisation der IDE wie etwa die Erzeugung eines neuen Editorfensters mit eigenem Seitenregister, aber diese Optionen können Sie sicher leicht selbst erforschen.

Was auch eher selbst zu erforschen bzw. durch Praxisexperimente zu erlernen ist, ist das Fenster-Docking. Zum Andocken in Frage kommen alle Hilfsfenster mit einer schmalen Titelzeile nach dem Beispiel des Objektinspektors. Auch das per Voreinstellung an das Editorfenster angedockte neue CodeExplorer-Fenster und alle Fenster des integrierten Debuggers (aufgelistet im Ansichtsmenü) sind andockbar. Abbildung 1.2 zeigt ein Beispiel einer durch Docking umkonfigurierten Kylix-IDE.

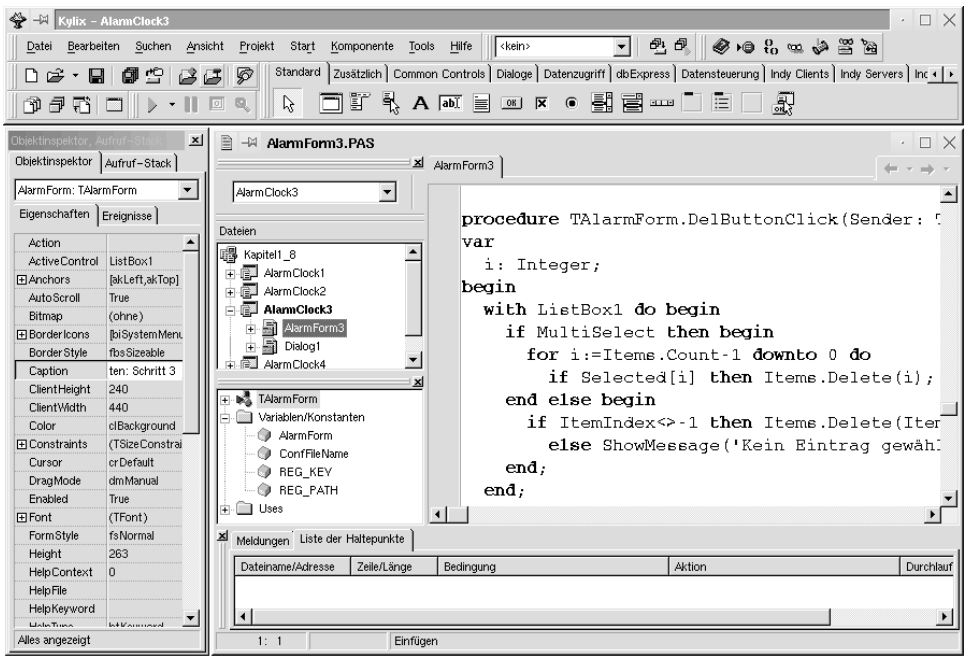


Abbildung 1.2: Nur eine von unzähligen Möglichkeiten, die Fenster der Kylix-IDE zu verbinden und den Platz chts von den Menüpunkten mit einer frei konfigurierbaren Symbolleiste zu füllen.

Angedockt werden kann zum einen an allen Seiten des Editorfensters außer an der Oberseite, zum anderen können Hilfsfenster gegenseitig aneinander oder sogar übereinander andocken. Zum Andocken (oder zum Wiederabtrennen) eines Fensters ziehen Sie es einfach mit der Maus an die gewünschte neue Position bzw. auf die gewünschte Seite des Editorfensters. Während dieses Ziehvorgangs erhalten Sie durch einen grau gezeichneten Vorschaurahmen Rückmeldung darüber, was mit dem Fenster passieren würde, ließen Sie die Maustaste jetzt los. Hierfür gibt es grob gesagt die folgenden Möglichkeiten:

- Das Fenster wird den Bildschirm als freies Fenster betreten. In diesem Fall nimmt der Vorschaurahmen genau die vom normalen Verschieben eines Fensters gewohnten Ausmaße an.
- Das Fenster wird an der Seite eines anderen Fensters angedockt: Der Vorschaurahmen markiert in diesem Fall die betreffende Fensterseite.
- Das Fenster wird mit einem anderen Fenster zu einem mehrseitigen Fenster gekoppelt. Für diesen Fall kennzeichnet der Vorschaurahmen den mittleren Teil des anderen Fensters.

Dieser Dockingmechanismus bringt eine so gewaltige Flexibilität mit sich, dass es schon fast wieder problematisch wird: So spielt es für das Andocken an ein anderes Fenster keine Rolle, ob dieses andere Fenster vielleicht gar nicht sichtbar ist. Wenn Sie beispielsweise den CodeExplorer von der linken auf die rechte Seite des Editorfensters verschieben wollen und sich unter dem Editorfenster andere Hilfsfenster der IDE verstecken, so kann es sein, dass diese Sie mit einem hin- und herzuckenden Vorschaurahmen aus dem Konzept zu bringen versuchen.

Zum Glück gibt es jedoch einen Weg, die Verwirrung beim Docking in Grenzen zu halten: Bei Fenstern, die Sie einfach nur verschieben wollen, ohne dass sie an jeder anderen Fensterecke hängen bleiben, schalten Sie im Kontextmenü einfach die Option ANDOCKBAR aus. Wenn Sie temporär auf das Docking bzw. auf die Vorschaurahmen verzichten wollen, drücken Sie während des Mausziehvorgangs auf Strg.

Die Konfiguration der IDE speichern

Sehr empfehlenswert ist es, im Umgebungsdialog (TOOLS | UMGEBUNGSOPTIONEN) auf der Seite PRÄFERENZEN in der Gruppe OPTIONEN FÜR AUTOSPEICHERN die Option (PROJEKT-)DESKTOP einzuschalten, bei der Kylix den Aufbau der IDE (offene Fenster, Position dieser Fenster) und vor allem die Liste der im Editorfenster geöffneten Dateien speichert. Wenn Sie in einem Projekt viele Dateien, vielleicht auch einige der CLX-Quelltexte gleichzeitig offen haben, müssten Sie diese Dateien ohne diese Option mühsam neu laden, wenn Sie das Projekt erneut öffnen. Auch die Docking-Konfiguration wird beim erneuten Laden des Projekts wiederhergestellt (verändert sich aber eventuell, wenn Sie zu einem anderen Projekt wechseln).

Falls Sie Auto-Speichern einschalten, legt Kylix die Daten zu jedem Projekt jeweils in einer Datei mit der Endung `.desk` ab. Nicht zur IDE-Konfiguration zählen beispielsweise die Compiler- und Linker-Optionen, die automatisch in `.kof`-Dateien gespeichert werden, ohne dass Sie dazu einen Schalter aktivieren müssten. Die projektübergreifenden Einstellungen der IDE speichert Kylix übrigens in seinem Ressourcenverzeichnis, der genaue Pfad der Datei lautet `~/.borland/delphi60rc` (für Linux-Neulinge: »~« meint das Heimatverzeichnis des Benutzers, also etwa `/root` oder `/home/benutzername`).

Desktop-Konfigurationen speichern

Kylix bietet eine bequeme Möglichkeit, verschiedene Projekt-Desktops zu speichern, die auch innerhalb desselben Projekts einfach abgerufen werden können. Hierzu gibt es eine kleine, standardmäßig neben der Menüleiste angeordnete Toolbar mit einer Liste von gespeicherten Desktops (zu Beginn leer), einem Schalter zum Speichern eines Desktops sowie einen Schalter zum Festlegen des Debug-Desktops.

Die Speichern-Funktion fordert Sie zum Angeben eines Namens für den Desktop-Aufbau auf und fertigt dann einen Schnappschuss des Desktops an, den Sie sich gewissermaßen auch als Schablone vorstellen können. Denn er lässt sich von da an in allen Projekten abrufen und stellt dann dort die Fenster so ein wie zum Zeitpunkt der Speicherung, ändert aber ihren Inhalt nicht. Sie rufen einen gespeicherten Desktop ab, indem Sie einfach den selbst gewählten Namen aus der aufklappbaren Liste auswählen.

Die gespeicherten Desktops beeinträchtigen eine eventuell eingeschaltete Desktop-Autospeichern-Funktion nicht. Diese veranlasst Kylix weiterhin, beim Öffnen eines Projekts den letzten Desktop dieses Projekts wiederherzustellen, egal ob es sich dabei um einen separat abgespeicherten Desktop handelt oder nicht. Die Desktops aus der Desktop-Liste werden also nur dann aktiviert, wenn Sie sie manuell auswählen.

Eine Ausnahme dazu bildet der Debug-Desktop. Dies ist ein über den dritten Schalter der Desktop-Symbolleiste (oder über ANSICHT | DESKTOPS | DEBUG-DESKTOP EINSTELLEN) speziell festgelegter Desktop, der automatisch aktiviert wird, sobald Sie eine Anwendung in Kylix starten und damit den integrierten Debugger aktivieren.

Hinweis: Kylix speichert die projektübergreifenden Desktops in seinem Ressourcenverzeichnis `~/.borland` jeweils unter dem Desktop-Namen, erweitert mit der Dateiendung `.dst`. Beim Start sucht es einfach alle existierenden `.dst`-Dateien zusammen, so dass Sie die Desktop-Auswahl durch Hineinkopieren neuer `.dst`-Dateien erweitern oder Sicherungskopien durch Duplizieren der `.dst`-Dateien anfertigen können.

1.4.2 Hilfe zu IDE und Sprachreferenz

Abgesehen davon, dass hier der Platz fehlt, um alle Möglichkeiten der IDE zu beschreiben, wäre eine Referenz über alle Menübefehle, Schalter und Optionen sicher wenig interessant. Im restlichen Kapitel sind daher die für den grundlegenden Arbeitsablauf wichtigen und einige weitere besonders interessante Fähigkeiten der IDE herausgegriffen. Zu weiteren Details gibt Ihnen Kylix mit den üblichen Mitteln Hilfestellung (Hinweisfenster über den Schaltern der Symbolleisten, Hilfe mit F1 in den Menüs und über Hilfeschalter in Dialogboxen). Wenn Sie sich also irgendwann für die Compileroptionen interessieren, sollten Sie unter den Projektoptionen (PROJEKT | OPTIONEN) die Seite COMPILER aufschlagen und dort die Hilfsinformation abrufen. In den meisten Fällen empfiehlt es sich, als Erstes im lokalen Menü des aktuellen Fensters nachzusehen, ob dieses bereits die gewünschte Funktion enthält.

CLX-Referenz

Um bei der Programmierung schnelle Hilfe zu einem Bestandteil der CLX zu erhalten, können Sie Borlands Online-Sprachreferenz aus dem Editor heraus mit F1 aufrufen. Kylix sollte dann die Seite der Hilfedatei aufschlagen, auf der das Wort an der aktuellen Editorposition beschrieben ist. In den Hilfeseiten der einzelnen Klassen (und Komponenten, wie etwa *TButton*, *TMainMenu*) können Sie ein eigenständiges Nebenfenster öffnen, das eine Übersicht über alle Eigenschaften, Methoden oder Ereignisse der Klasse gibt und sogar noch eine alphabetische Sortierung erlaubt. Wenn Sie dann z. B. eines der Ereignisse auswählen, wird das Übersichtsfenster nicht geschlossen, sondern bleibt neben der detaillierten Beschreibung dieses Ereignisses sichtbar und erlaubt Ihnen ein schnelles Weiterblättern zu den anderen Ereignissen oder einen Sprung zurück zur Hauptseite der Klasse.

Lücken in der Online-Referenz

Neue Versionen von Borlands Softwareentwicklungssystemen sind immer heiß begehrt und nur wenige möchten auf die neuen und eigentlich fertigen Produkte warten, bis auch die Online-Hilfe perfekt ist. Das führt dazu, dass die Online-Hilfen eines neuen Produkts, so auch die der ersten Kylix-1-Version, fast immer einige auffällige Lücken aufweisen wie undokumentierte neue Properties oder gar ganze undokumentierte Komponentenklassen (wie in der ersten Kylix-Version etwa die Klasse *TIconView*). Doch Abhilfe ist auf verschiedene Weise möglich:

- Borland stellt Updates der Hilfedateien im Internet bereit, die oft schon kurz nach Erscheinen des Produkts in den Newsgroups angekündigt werden.
- Ultimative Information über alle verfügbaren Properties und Methoden einer Klasse erhalten Sie aus dem Quelltext der CLX, in dem Sie schon durch wenige Tastendrücke die richtige Position finden können: Zunächst können Sie ohne Zuhil-

fenahme eines Dateiauswahldialogs und ohne das Browsen durch tiefe Verzeichnisstrukturen die gewünschte Unit öffnen, indem Sie im Editor der IDE die Eingabemarke auf den Namen der Unit setzen und Strg+↵ drücken. (Falls Sie den Namen der Unit nicht kennen: Die für eine bereits im Formular vorkommende Komponente in Frage kommenden Units befinden sich in der *uses*-Anweisung der Formular-Unit.) Im zweiten Schritt bringt Sie eine Suchfunktion des Suchen-Menüs schnell zur gewünschten Klasse. Direkt zur Klassendeklaration von *ClassName* kommen Sie z. B. bei der inkrementellen Suche spätestens nach Eingabe von `TClassName = class(`. Von dort können Sie nun bei Bedarf mit Hilfe von Strg+⇧+↓ sogar direkt in die Implementationen von Methoden springen.

Natürlich weisen die wenigsten Klassen im Quelltext eine Dokumentation auf, daher ist Erfahrung in der Komponentenentwicklung sehr hilfreich, um mit dem Quelltext der CLX etwas anfangen zu können. (Falls Sie noch nicht über solche Erfahrung verfügen, wäre ein erster Tipp, dass Sie die mit *private* und *protected* überschriebenen Bereiche der Klassendeklaration überspringen und sich gleich den *public*- und *published*-Bereichen zuwenden.)

Hinweis: Units können nur dann mit der oben erwähnten Tastenkombination Strg+↵ geöffnet werden, wenn sie in einem Pfad enthalten sind, der in den Umgebungsoptionen unter BIBLIOTHEK / SUCHPFAD aufgeführt ist.

1.4.3 Die Komponentenpalette

Alle in ein Formular einfügbaren Komponenten sind in der Komponentenpalette versammelt. Aufgrund ihrer großen Zahl mussten sie auf mehrere Seiten verteilt werden. Einen Überblick über einzelne Komponenten werden Sie in Kapitel 1.8 finden, an dieser Stelle soll ein Überblick über die Seiten der Palette genügen (nicht alle Seiten sind in allen Kylix-Versionen verfügbar):

Auf der Seite *Standard* befinden sich die grundlegenden Kontrollelemente, die in allen Betriebssystemen bzw. grafischen Oberflächen schon seit Jahren zum – nun: eben zum Standard gehören, z. B. Listenfenster, Markierungsfelder, Eingabefelder. Auch die Menüs sind mit einer Haupt- und einer Popup-Menükomponente auf dieser Seite vertreten, und passend dazu die Aktionslisten-Komponente *TActionList*, die kein eigenständiges Steuerelement ist und in Kapitel 4.6.4 genauer erläutert werden wird. Schließlich finden Sie ganz links noch einen Schalter für *TFrame*: Dies ist eigentlich gar keine Komponente, jedenfalls keine sofort verwendbare. Um Frames verwenden zu können, müssen Sie zuerst selbst eines definieren. Wie das geht, beschreibt Kapitel 3.7.2.

- *Zusätzlich*: Hier befinden sich weitere Elemente, die in vielen Anwendungen vorkommen; von einfachen Speedbuttons bis zu komplexen Eingabekomponenten wie z.B. zwei Tabellenkomponenten. Delphi-Benutzer werden auf dieser Seite auch die bisher von der Seite *System* bekannten Komponenten *Timer* und *PaintBox* finden. Da alle anderen Komponenten der Seite *System* unter Kylix nicht verfügbar sind (überwiegend deshalb, weil Linux ohne die Microsoft-Technologien DDE und OLE arbeitet), wurde die Seite *System* mit *Zusätzlich* fusioniert.
- Die Seite *Common Controls* enthält eine weitere Ebene von »Standard-Komponenten«, und zwar solche, die *nicht* aus der Steinzeit grafischer Benutzeroberflächen stammen: Baumansichten, Listenansichten, Fortschrittsbalken etc. Delphi-Benutzer (bis Delphi 5) werden diese Seite unter dem Namen *Win32* kennen, da die meisten ihrer Komponenten mit Windows 95 in die Windows-Welt eingeführt wurden (aus dieser Welt stammt auch der Name der »Common Controls«). Unter Linux hat es mehrere Ansätze gegeben, mit den Windows-Common Controls vergleichbare Steuerelemente zu »standardisieren«, wobei Kylix natürlich dem Ansatz der Qt-Bibliothek am nächsten steht. Daher stellen einige (aber nicht alle) der Komponenten auf dieser Seite lediglich komfortable Verpackungen der Qt-Klassen dar (eine Tabelle hierzu finden Sie in der Einleitung von Kapitel 3). Delphi-Benutzer seien besonders auf die Komponenten *TextViewer* und *TextBrowser* hingewiesen, welche in Kylix als Ersatz für die unter Linux erwartungsgemäß nicht verfügbare Internet Explorer-Komponente von Microsoft dienen können.
- *Dialoge*: Diese Seite enthält Komponenten, die sechs der am häufigsten in Anwendungen verwendeten Dialoge kapseln. In Kylix werden diese Dialoge von der Qt-Bibliothek geliefert und sind im optischen Aufbau weitgehend an den Standarddialogen von Windows angelehnt.
- *Datenzugriff*: Hier befinden sich klassische, von Delphi eingeführte Datenbankkomponenten, mit denen Sie Verbindungen zu Datenbanken und Datenbankservern aufbauen können. Da die BDE, auf der die meisten dieser von Delphi für Windows bekannten Komponenten basieren, nicht nach Linux portiert wurde, ist diese Seite unter Kylix mit nur drei Komponenten ziemlich spärlich besiedelt, sollte aber trotzdem nicht gering geschätzt werden, enthält sie doch zwei absolut grundlegende Komponenten wie *TDataSource* und *TClientDataSet*.
- *dbExpress* ist Borlands neue Datenbankarchitektur, die in Kylix die mit der BDE arbeitenden Datenzugriffskomponenten ersetzt. In Delphi für Windows kommen diese Komponenten als Alternativen zu den BDE-Komponenten hinzu und Kapitel 7 beschreibt, warum sich die Verwendung dieser neuen Komponenten in jedem Fall vorteilhaft auswirken kann.
- *Datensteuerung* könnte in dieser Übersetzung mit dem *Datenzugriff* verwechselt werden. Was aber wirklich gemeint ist, sind *Datensteuerelemente*, das sind überwie-

gend Standardkomponenten, die um eine Datenbankanbindung erweitert wurden. Mit Hilfe dieser Komponenten können Sie den Inhalt der Datenbanken auf vielfältige Weise anzeigen und editieren lassen (diese Seite ist nur in den Produktversionen verfügbar, in denen auch die Seite *Datenzugriff* vorhanden ist).

Internet enthält hauptsächlich nicht-visuelle Komponenten, die in Internet-Server-Anwendungen benötigt werden, aber auch eine Komponente für Clients (*TcpClient*). Die unter Delphi für Windows an dieser Stelle gewohnte *WebBrowser*-Komponente wird durch *TTextBrowser* auf der Seite *Common Controls* ersetzt.

- Schließlich gibt es noch drei Palettenseiten, auf die sich die Internet-Komponenten des Open Source-Pakets *Internet Direct* (Indy) verteilen.

Neben den schon vordefinierten Komponenten und Registerseiten befindet sich nahezu unbegrenzter Platz zum Installieren neuer Komponenten (dank der beiden kleinen Schalter zum Scrollen, falls nicht alle Registerzungen auf den Bildschirm passen). Darüber hinaus können Sie die schon installierten Komponenten beliebig zwischen den einzelnen Seiten austauschen und neue Seiten erstellen, denn auch die Komponentenpalette gibt in ihrem lokalen Menü den Punkt EIGENSCHAFTEN an, der Sie aber nicht zu einem eigenen Editor, sondern lediglich auf eine Seite führt, die es auch im Optionsdialog des Menüpunktes TOOLS | UMGEBUNGSOPTIONEN gibt.

Eine alphabetische Auflistung aller Komponenten finden Sie unter dem Menüpunkt NSICHT | KOMPONENTENLISTE. Diese Liste kann Ihnen bei der Suche nach einer bestimmten Komponente, von der Sie nur den Namen kennen, behilflich sein.

Installation neuer Komponenten

Eine wichtigere Möglichkeit, die Komponentenpalette anzupassen, stellt die Installation neuer Komponenten dar. Diese ist Thema des Kapitels 8.2.3.

1.4.4 Entwerfen von Formularen

Beim visuellen Entwurf einer Kylix-Anwendung liegt der Mittelpunkt des Interesses auf dem Formular, das in Abbildung 1.1 unter der Überschrift *Form1* bereits einen groen Teil des Bildschirms einnimmt.

Entwurfszeit und Laufzeit

Bei der Arbeit mit einem Formular wird zwischen zwei sehr verschiedenen Arbeitsbedingungen unterschieden:

Zur *Entwurfszeit* entwerfen Sie das Formular mit den visuellen Werkzeugen der Kylix-IDE, platzieren Komponenten darin und stellen Eigenschaften im Objektinspektor ein.

- Wenn Sie eine Anwendung über das START-Menü von Kylix starten, erscheint die *Laufzeit*-Version des Fomulars. Zur Laufzeit haben Sie von Kylix aus keine Möglichkeit mehr, das Formular zu beeinflussen. Der Programmcode, mit dem Sie auf die verschiedenen Ereignisse des Formulars und seiner Komponenten reagieren, ist nur zur Laufzeit aktiv.

Sie können das Formular zur Entwurfszeit unter verschiedenen Blickwinkeln betrachten:

- *Als passives Objekt*, das Sie mit der Maus und mit den Werkzeugen gestalten, die um das Formular herum angeordnet sind: Verändern Sie im Objektinspektor Attribute des Formulars und seiner Bestandteile, so passen diese meistens ihr Aussehen entsprechend an (z. B. die Farbe, die Schriftart etc.). Wählen Sie aus der Komponentenpalette eine Komponente aus, können Sie diese zum Formular hinzufügen. Die *Ausrichtungspalette* (Menü ANSICHT) gibt Ihnen weiteren Einfluss auf das Formular.
- *Als teilweise aktiver Prototyp* des Fensters, das zur Laufzeit des Programms erscheinen wird. Tatsächlich laufen schon beim Entwurf des Formulars große Teile des Codes der Komponenten ab, hauptsächlich diejenigen Teile, die die Komponente auf den Bildschirm zeichnen. So können beispielsweise *DBGrid*-Komponenten den Inhalt einer Datenbank schon zur Entwurfszeit darstellen.

Die zweite Sichtweise wird besonders dann wichtig, wenn Sie eigene Komponenten entwerfen, daher erfahren Sie in Kapitel 8 Näheres dazu, *wie* aktiv der Programmcode schon zur Entwurfszeit des Formulars ist.

Hinweis: Die obigen Erläuterungen zeigen bereits, dass es in der Kylix-IDE keine klar erkennbare Einrichtung gibt, die sich als Formular-Editor oder *Formular-Designer* bezeichnen lässt. Vielmehr arbeiten verschiedene Einrichtungen der IDE (Objektinspektor, Komponentenpalette, Ausrichtungspalette, BEARBEITEN-Menü) mit der Entwurfszeit-Version des Formulars zusammen und bilden mit dieser quasi einen Formular-Designer. Kylix selbst hat intern eine klare Definition des Formular-Designers und stellt diese den Komponentenentwicklern teilweise über Schnittstellen des OpenTools-API zur Verfügung. Es ist jedoch im Folgenden nicht erforderlich, von einem eigenständigen Formular-Designer zu sprechen.

Hinzufügen von Komponenten

Kylix befindet sich nach dem Start in einem »Manipulationsmodus« (erkennbar an der Hervorhebung des Zeigersymbols links neben den Komponenten), in dem Sie mit der Maus bestehende Komponenten verrücken oder in der Größe ändern können. Um eine Komponente hinzuzufügen, müssen Sie den Typ der gewünschten Komponente zuerst in der Komponentenpalette auswählen. Kylix behandelt daraufhin die einzuzeichnende Komponente ähnlich wie ein Grafikprogramm die Elemente einer Grafik. Sie

önnen die Komponenten also mit einem Mausklick positionieren und bei gedrückter Maustaste auf die gewünschte Größe bringen. Es genügt jedoch auch ein einziger Klick, um die Komponente in ihrer vordefinierten Größe einzuzeichnen.

Nachdem Sie ein Exemplar des gewählten Komponententyps eingefügt haben, aktiviert Kylix automatisch wieder das Zeigersymbol (also den »Manipulationsmodus«). Wenn Sie mehrere Komponenten desselben Typs hintereinander frei einzeichnen wollen, müssen Sie den Komponentenschalter besonders fest eindrücken, jedoch nicht mit zusätzlichem Druck auf die Maustaste, sondern durch gleichzeitiges Drücken von ⇧. Der Komponentenschalter bleibt dann so lange aktiv, bis Sie ihn durch einen erneuten Klick auf irgendeinen Schalter wieder lösen.

Es gibt noch einen zweiten schnellen Weg. Mit einem Doppelklick auf die Komponente in der Palette erreichen Sie, dass Kylix ein Exemplar davon in die Mitte des Formulars setzt bzw. auf die gerade markierte Komponente (sofern diese in der Lage ist, untergeordnete Komponenten aufzunehmen). Diesen Doppelklick können Sie so oft wiederholen, wie Sie Komponenten benötigen, müssen diese aber danach per Maus in die richtige Lage und Position bringen.

Editieroperationen

Die folgende kurze Zusammenfassung soll Ihnen einen Überblick über die Möglichkeiten des Formularentwurfs geben, zunächst über die Manipulation der Komponenten mit Maus und Tastatur (die Tastatursteuerung ist nur möglich, wenn kein Element der Komponentenpalette gewählt ist):

Mit einzelnen Mausklicks markieren Sie einzelne Komponenten; halten Sie zusätzlich ⇧ gedrückt, werden dabei die schon bestehenden Markierungen nicht gelöscht, so dass Sie mehrere Komponenten gleichzeitig auswählen können.

Um von Komponente zu Komponente zu springen, verwenden Sie die Pfeiltasten oder die Taste ⇆ (mit »Springen« ist gemeint, dass jeweils eine andere Komponente markiert wird).

Indem Sie mit der Maus ein Rechteck um mehrere Komponenten ziehen, wählen Sie alle Komponenten aus, die innerhalb des Rechtecks liegen. Wenn dies nicht möglich ist, weil sich die Komponenten innerhalb einer anderen Komponente befinden, die Sie durch die Maus verschieben würden, halten Sie Strg gedrückt, wenn Sie beginnen, das Rechteck aufzuziehen. Oder halten Sie ⇧ gedrückt und klicken Sie die auszuwählenden Komponenten einzeln an.

Mehrere ausgewählte Komponenten können Sie mit der Maus gleichzeitig verschieben. Eine gemeinsame Größenänderung ist nur über den Menüpunkt GRÖSSE... (lokales Menü des Formulars und BEARBEITEN-Menü) und beim Skalieren

möglich (Menüpunkt SKALIERUNG...). Beim Skalieren werden jedoch Höhe und Breite der gewählten Komponenten um denselben Faktor vergrößert.

- Wenn sich Komponenten überlappen, können Komponenten ganz oder teilweise von anderen Komponenten verdeckt werden. Um verdeckte Komponenten wieder sichtbar zu machen, verwenden Sie einen der lokalen Menübefehle NACH HINTEN SETZEN und NACH VORNE SETZEN. Ersterer platziert die markierten Komponenten hinter alle nicht markierten, Letzterer verhält sich genau umgekehrt. Diese Verschiebung ist jedoch nicht immer wirksam. So können Sie *SpeedButtons* oder *Labels* beispielsweise nicht auf einen *Button* legen. Allgemein können Komponenten, die keine Fenster sind, nicht auf Komponenten liegen, die Fenster sind. Da Sie sich fürs Erste nicht weiter mit diesem Unterschied zu beschäftigen brauchen, kommt erst Kapitel 3.1.3 wieder darauf zurück.
- Es gibt auch Tastaturkürzel zum Verschieben, Vergrößern und Verkleinern der aktuellen Komponente(n), diese bieten sich aufgrund ihrer Genauigkeit für Feintuning an: Mit Strg und den Pfeiltasten verschieben Sie die gesamte Komponente, mit ⇧ und den Pfeiltasten verschieben Sie nur die rechte untere Ecke. Mit den Pfeiltasten alleine bewegen Sie sich von Komponente zu Komponente.
- Die üblichen Ausschneiden/Einfügen-Operationen (Cut & Paste) stehen natürlich auch für Komponenten zur Verfügung. Ein Anwendungsbeispiel ist das schnelle Vervielfältigen einer gerade eingezeichneten Komponente: Diese ist nach dem Einzeichnen automatisch markiert, so dass sie mit BEARBEITEN | KOPIEREN schnell in die Zwischenablage kopiert werden kann. Mit dem EINFÜGEN-Befehl können Sie von dieser Vorlage nun beliebig viele Kopien anfertigen, die Kylix etwas verschoben über dem Original anordnet. Diese Methode ist natürlich nur dann schnell, wenn Sie statt der Menübefehle die Tastenkürzel verwenden. Diese hängen jedoch von Ihrer gewählten Tastatureinstellung ab und werden neben dem entsprechenden Menüpunkt angezeigt.
- Zum Ausrichten der ausgewählten Komponente(n) besitzt Kylix eine eigene Schalterleiste, die Sie mit ANSICHT | AUSRICHTUNGSPALETTE aktivieren. Die einzelnen Schalter erklären sich über automatisch erscheinende Hinweise selbst, sobald Sie den Mauszeiger kurze Zeit über das fragliche Symbol halten. In der Dialogbox AUSRICHTEN... (lokales Menü des Editors) finden Sie dieselben Optionen in schriftlicher Form wieder.
- Um die markierten Komponenten zu löschen, drücken Sie Entf. Sie löschen damit auch die Einstellungen, die Sie im Objektinspektor vorgenommen haben, können die Löschung aber mit BEARBEITEN | RÜCKGÄNGIG wieder ungeschehen machen.

Reihenfolgen der Komponenten

Die Komponenten eines Formulars liegen nicht einfach so im Formular herum, sondern sind in zweierlei Reihenfolgen angeordnet: Alle sichtbaren Elemente, die von der astatur angesteuert werden können, können Sie innerhalb der TABULATORREIHENFOLGE (lokales und globales Menü BEARBEITEN) umgruppieren – das ist die Reihenfolge, in der die Elemente durch Drücken von ⇆ angesteuert werden (wie in jedem blichen Dialog). Eine andere Möglichkeit, die Tabulatorreihenfolge zu ändern, liegt in der Eigenschaft *TabOrder*, die Sie bei jeder ansteuerbaren Komponente im Objektinspektor finden (dazu später mehr).

In der zweiten Reihenfolge (ERSTELLUNGSREIHENFOLGE) brauchen Sie für den Einstieg noch keinen Sinn zu sehen: Sie bezieht sich nur auf die nicht-visuellen Komponenten, beispielsweise die Datenzugriffskomponenten. Sie können die Reihenfolge festlegen, in denen diese initialisiert und im Array *Components* angeordnet werden (siehe Kapitel 3.2.1).

Sichtbare Elemente, die nicht von der Tastatur angesteuert werden, können Sie in keiner der Reihenfolgen anordnen, da ihre Reihenfolge programm- und bedientechnisch keine Rolle spielt (zu diesen Komponenten gehören wieder die Komponenten, die sich schon weder nach vorne noch nach hinten schieben ließen, wie z. B. die *Label*-Komponente).

Eine weitere Möglichkeit, die Sie zum Entwurfszeitpunkt der Anwendung haben, ist, Komponenten zu gruppieren. Gruppierte Elemente sind einem gemeinsamen Gruppenelement, meistens einer Komponente des Typs *TPanel*, untergeordnet, gehören aber weiterhin in den Besitz des Formulars. Mehr dazu finden Sie in R1 Seite 349.

Zum Beispielprogramm

Das Hauptfenster für unseren kleinen Systemdateien-Editor erhält die folgenden Komponenten von der Seite *Standard*:

- ein mehrzeiliges Textfeld (*Memo*) als Editierbereich,
- einen Schalter (*Button*) zum Speichern der Datei,
- einen weiteren Schalter, über den die Schriftart des Editors geändert werden kann,
- eine aufklappbare Liste (*ComboBox*) zum Auswählen einer anderen Hintergrundfarbe,
- ein Textelement (*Label*), das als Beschriftung für die aufklappbare Liste dienen soll,
- ein weiteres Label, in dem später die aktuelle Editierposition im Memo angezeigt werden soll,

ein Markierungsfeld (*Checkbox*), über welches das Memo auf »Nur Lesen«-Betrieb umgeschaltet werden kann – in diesem Modus sind keine Veränderungen seines Inhalts möglich.

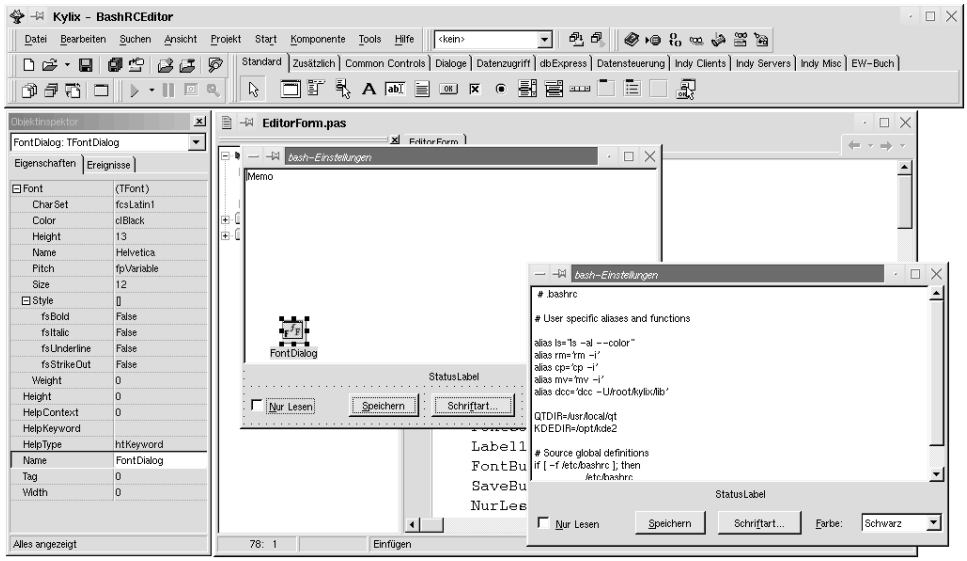


Abbildung 1.3: Das erste Formular des Beispielprogramms zur Entwurfszeit, rechts unten zur Laufzeit (nach Hinzufügen der Ereignisbearbeitungsmethoden in Kapitel 1.5.4)

Zeichnen Sie, wenn Sie die einzelnen Schritte in der Praxis mitverfolgen möchten, die genannten Elemente nach Belieben in das Formular ein, das Layout spielt für die Funktionsfähigkeit des Programms (noch) keine Rolle.

Soweit an dieser Stelle zum Beispielprogramm; aufwändigere Aktionen sind für den nächsten Schritt zu erwarten, wenn in Kapitel 1.4.6 die Properties eingestellt werden. Vorher benötigen wir eine Möglichkeit, das gerade begonnene Projekt erstmals zu speichern.

1.4.5 Dateiverwaltung

Wenn Sie ein neues Projekt erzeugen (DATEI | NEUE ANWENDUNG), erhalten alle Dateien dieses Projekts zunächst einen vorgegebenen Namen, den Sie später auf Anfrage von Kylix ändern können. Die vorgegebenen Namen lauten *Project1* für das Projekt, *Form1* für das Formular und *Unit1* für die zugehörige Quelltextdatei, wobei Kylix die Ziffer 1 der Formulare und Units erhöht, wenn Sie mehrere neue davon unbenannt in ein Projekt einfügen.

Es folgt eine Zusammenfassung der wichtigsten Punkte des Menüs DATEI:

Mit ALLES SPEICHERN speichern Sie sowohl die Projektdatei als auch alle Dateien, die zum Projekt gehören und seit der letzten Änderung noch nicht gesichert worden sind.

Mit SPEICHERN und SPEICHERN UNTER ... speichern Sie nur die aktuelle Datei bzw. das aktuelle Formular (Formular und zugehörige Unit werden immer gleichzeitig gespeichert, sofern sie verändert wurden).

Mit PROJEKT SPEICHERN UNTER ... geben Sie der obersten Datei des aktuellen Projekts (der Projektdatei) einen neuen Namen. Beim ersten Speichern eines Projekts, wenn das Projekt also noch keinen Namen hat, gelangen Sie automatisch immer in diese Dialogbox. Beim ersten Speichern erhalten Sie außerdem noch eine zweite *Speichern-unter*-Dialogbox, um die Formulardatei samt zugehöriger Quelltext-Unit zu benennen. Da Kylix bei jedem neuen Projekt mit den Namen *Project1* und *Unit1* neu zu zählen beginnt, kann es schnell zu Dateinamenskonflikten kommen, wenn Sie zwei Projekte gleichzeitig im selben Verzeichnis speichern wollen, ohne die vorgegebenen Namen zu ändern (abgesehen von den anderen offensichtlichen Nachteilen quasi namenloser Dateien).

- Das Laden von Projekten kann auf verschiedene Weise geschehen: Wenn Sie die Autospeichern-Option für den Desktop verwenden, lädt Kylix beim Start immer automatisch das zuletzt bearbeitete Projekt. Über den Menüpunkt ÖFFNEN... können Sie ein anderes Projekt laden, wenn Sie in der Dateiauswahlbox eine Projektdatei (erkennbar an der Endung `.dpr` und dem von `.pas`-Dateien unterschiedlichen Icon) auswählen. Außerdem finden Sie unter dem Menüpunkt NEU ÖFFNEN und beim entsprechenden DropDown-Schalter der Symbolleiste eine Liste der zuletzt bearbeiteten Projekte.

Hinweis: Obwohl Projektdateien und Units sich bereits durch die Endung unterscheiden, müssen Sie ihnen unterschiedliche Namen geben. Sie können also den Namen der Datei, die das Hauptformular enthält, nicht für den Namen des Projekts wiederverwenden.

Weitere Informationen über Dateien und Projekte finden Sie ab Kapitel 1.6.2.

1.4.6 Der Objektinspektor

Jede Komponente und jedes Formular verfügt über eine Vielzahl an veränderbaren Eigenschaften, den *Properties*, von denen Sie die meisten im Objektinspektor verändern önnen. Die Werte der Properties aller Komponenten des Formulars und des Formulars selbst gehören zur »Fensterschablone« (dem Formular), von der zur Laufzeit echte

Fenster gebildet werden (siehe Kapitel 1.2.2). Das heißt, dass jedes Fenster, das Sie auf Grundlage dieses Formulars erstellen, nicht nur die Komponenten enthält, die Sie in das Formular eingezeichnet haben, sondern dass auch die Properties der Komponenten und des Formulars selbst automatisch mit diesen Einstellungen initialisiert werden.

Es gibt einige Properties (bei manchen Komponenten sogar sehr viele), die Sie nicht zur Entwurfszeit, sondern nur zur Laufzeit des Programms ansprechen können, so z. B. die Zellinhalte einer String-Tabelle (*StringGrid*, Property *Cells*) oder die Zeichenfläche *Canvas* eines Formulars oder einer Zeichenfläche (*PaintBox*).

Die Daten der Komponenten verteilen sich auf die beiden Seiten *Eigenschaften* und *Ereignisse*. Sie bestehen jeweils aus einer zweispaltigen Liste, die Sie mit einer Bildlaufleiste bewegen können. Die erste Spalte gibt den Namen des Properties bzw. des Ereignisses an, die zweite dessen Wert. Wir beschäftigen uns in diesem Kapitel nur mit der ersten der beiden Seiten.

Hinweis: Dieses Buch bezeichnet Properties fast immer deutlich als Properties, während sie in der Online-Hilfe meistens als »Eigenschaft« bezeichnet werden, was den sehr engen Bezug zum klar definierten Object-Pascal-Schlüsselwort *property* (das ja auch nicht übersetzt wurde) verschleiert. Die Bezeichnung *Eigenschaft* ist in diesem Buch allgemeineren Zwecken vorbehalten, so dass keine Verwechslungsgefahr besteht.

Property-Kategorien

Wenn Ihnen die lange Liste von Properties einer Komponente zu unübersichtlich ist, können Sie sie von Kylix in Kategorien einteilen lassen. Wählen Sie dazu einfach ANORDNEN | NACH KATEGORIE aus dem Popup-Menü des Objektinspektors. Abbildung 1.4 zeigt die dadurch hervorgerufene Darstellung des Objektinspektors. Wenn Sie sich zum ersten Mal Kategorien anzeigen lassen, sehen Sie zunächst nur die Liste der Kategorien. Über das Plus-Symbol erhalten Sie dann die Liste der Properties, die zur jeweiligen Kategorie gehören, wobei viele Properties zu mehreren Kategorien gleichzeitig gehören. Kylix merkt sich, auch wenn Sie zwischendurch wieder zur alphabetisch sortierten Normaldarstellung zurückwechseln, welche Kategorien Sie angezeigt haben wollen.

Einen Vorteil hat die Kategorisierung der Properties auch, wenn Sie die normale alphabetische Sortierung im Objektinspektor gewählt haben: Sie können Properties von uninteressanten Kategorien ausblenden und erhalten so eine übersichtlichere alphabetische Liste. Für diese Filterfunktion ist das Popup-Untermenü ANSICHT des Objektinspektors zuständig.

Auswahl des Objekts für den Objektinspektor

Der Objektinspektor zeigt von sich aus immer die Einstellungen der im aktuellen Formular ausgewählten Komponente an. Ist keine Komponente ausgewählt, stellt er die Daten des Formulars selbst dar. Außer durch die Markierung am Bildschirm können Sie das ausgewählte Element aus der aufklappbaren Liste unter der Titelzeile des Objektinspektors ersehen. Aus dieser Liste können Sie jede andere in Ihrem Projekt befindliche Komponente auswählen, um sie zu bearbeiten.

Diese Liste ist insbesondere dann wichtig, wenn eine Komponente vollständig durch ne andere Komponente verdeckt wird, was häufig vorkommt, wenn die eine Komponente (z.B. ein *Panel*) als »Basisfläche« für eine andere Komponente dient und nicht ber einen eigenen Rand verfügt (wenn also etwa *Panel.BorderWidth* gleich Null ist). In diesem Fall können Sie die verdeckte Komponente nicht direkt mit einem Mausklick erreichen. Falls Sie der Komponente aber einen leicht zu merkenden Namen gegeben haben, können Sie sie über die aufklappbare Liste des Objektinspektors leicht finden (eine weitere Alternative ist das Durchlaufen aller Komponenten mit ⇆).

Um das Formular selbst auszuwählen, klicken Sie auf einen freien Bereich, nicht auf dessen Titelzeile. Wenn das gesamte Formular mit Komponenten belegt ist, können Sie es immer noch auswählen, indem Sie die Taste Strg gedrückt halten, während Sie auf eine beliebige Komponente klicken.

Hinweis: Mit Kylix scheint das Selektieren eines mit Komponenten belegten Formulars noch nicht ganz so einfach zu sein wie mit Delphi für Windows. Im Test mit der ersten Kylix-Version funktionierte es aber zuverlässig, wenn zusammen mit Strg auf eine *TPanel*-Komponente geklickt wurde.

Properties für das Beispielprogramm

Die meisten Einstellungen für das Formular unseres Beispielprogramms beziehen sich auf die Beschriftung der einzelnen Komponenten, die jeweils im Property *Caption* gespeichert wird, und vor allem auf den Namen (Property *Name*). Da Kylix als Name standardmäßig den Namen des Komponententyps mit einer Nummerierung verknüpft, entstehen dadurch wenig aussagekräftige Namen wie *Button1*, *Button2*, *Button3* usw. Das Beispielprogramm belässt es nur dann bei dieser Vorgabe, wenn es sich um eine Komponente handelt, die im Programmcode nicht direkt angesprochen wird.

Die folgende Tabelle enthält alle Einstellungen, die Sie in den Eigenschaften der Komponenten und des Formulars selbst vornehmen können, um die Beispielanwendung nachzubauen. Es handelt sich bis auf eine Ausnahme ausschließlich um einfache Einstellungen, bei denen es genügt, den Wert in das Editierfeld einzutragen. Lediglich für das *Items*-Property der *ComboBox* müssen Sie über den kleinen Schalter im Objekt-

inspektor einen kleinen Editor-Dialog aufrufen, in dem Sie dann die fünf Zeilen eingeben können, welche als Inhalt der aufklappbaren Liste dienen sollten. Mehr zu den komplexeren Properties erfahren Sie in Kapitel 1.4.7.

Wählen Sie also nacheinander das Formular selbst und die einzelnen Komponenten aus und geben Sie ihren Properties die folgenden Werte:

Komponente	**Property**	**Wert**
Formular	Name	frmBashEditor
	Caption	bash-Einstellungen
	ActiveControl	Memo
Memo	Name	Memo
	ScrollBars	ssAutoBoth
Speichern-Button	Name	SaveButton
	Caption	&Speichern
Schriftwahl-Button	Name	FontButton
	Caption	Schri&ftart
Farbauswahl-Liste	Name	FontComboBox
	Items (mehrzeiliger Text, per Schalter im Objektinspektor editierbar)	Schwarz Rot Grün Blau Gelb
Beschriftungs-Label	Name	Label1
	Caption	&Farbe:
	FocusControl	FontComboBox
Editierpositions-Label	Name	StatusLabel
	Alignment	taCenter
	AutoSize	False
»Nur Lesen«-Checkbox	Name	NurLesen
	Caption	&Nur Lesen
TFontDialog-Komponente	(keine Propertyeinstellungen vorgenommen)	

Alle Properties dieses Beispiels sind nicht nur in den hier verwendeten Komponenten, sondern auch in einigen anderen zu finden, doch die in der Tabelle folgende Beschreibung hält sich ausschließlich an das Beispielprogramm.

Property	Bedeutung
Alignment	positioniert den von einer Komponente (hier *TLabel*) angezeigten Text und sorgt im Beispiel mit dem Wert *taCenter* dafür, dass die Statuszeile zentriert wird, da die *Label*-Komponente die gesamte Fensterbreite abdeckt. Mit den beiden anderen möglichen Werten für *Alignment* richten Sie Text linksbündig oder rechtsbündig aus.
ActiveControl	gibt die Komponente des Formulars an, die bei Programmstart über den Tastaturfokus verfügt, im Beispiel gehen die Eingaben also sofort an das Memo-Textfeld.
AutoSize	besagt, ob das *Label*-Element seine Größe automatisch dem Platzbedarf der Beschriftung anpasst. Dies würde jedoch im Beispiel die Zentrierung über die *Alignment*-Einstellung unterlaufen, da das *Label* nach der automatischen Größenanpassung nicht mehr die gesamte Fensterbreite umfassen würde. Die automatische Größenanpassung muss hier daher ausgeschaltet, also auf *False* geändert werden.
Caption	enthält den Text, der die Komponente beschriftet (im Unterschied zum Property *Text*, das Sie stattdessen in Editierfeldern antreffen. Dieser grundsätzliche Unterschied zwischen *Caption* und *Text* findet sich bei allen Komponenten wieder: *Text* steht für editierbare Werte, während *Caption*-Text vom Benutzer zur Laufzeit nicht geändert werden kann. Indem Sie einem Zeichen ein '&' voranstellen, erreichen Sie, dass dieses Zeichen unterstrichen dargestellt wird.
ocusControl	siehe Beschreibung unter »Rücksichtnahme auf die Tastaturbedienung«
Items	Liste der Punkte, die in der aufklappbaren Liste erscheinen sollen.
ScrollBars	legt fest, inwieweit das Memo-Feld Gebrauch von Scrollbars machen kann. Im Beispiel sorgt der Wert *ssAutoBoth* dafür, dass beide möglichen Scrollbars automatisch angezeigt werden, sofern der Text horizontal oder vertikal nicht ganz in den sichtbaren Bereich passt. Mit den anderen Werten können Sie die Scrollbars fest ein- und ausschalten oder die Automatik auf den vertikalen oder horizontalen Scrollbar beschränken.

Bemerkung: Die oben gedruckte Tabelle der Properties enthält nur die Werte, die nicht mit der Standardeinstellung übereinstimmen. Es stimmt jedoch nicht, dass der Autor des Beispielprogramms mühsam darüber Buch führen musste, welche Properties er geändert hat, denn Kylix speichert selbst nur die geänderten Properties in den Formulardateien ab. Zur Bearbeitung der Formulardateien in der aufschlussreichen Textdarstellung siehe Kapitel 3.5.5.

ücksichtnahme auf die Tastaturbedienung

Auch wenn man beim visuellen Design eines Kylix-Formulars die Tastatur leicht vergessen kann, sollte man die Dialoge so gestalten, dass sie auch mit der Tastatur angenehm zu bedienen sind. Dabei kommt es besonders darauf an, dass der Benutzer jedes Dialogelement mit einer einzigen Taste ansteuern kann. Auf grafischen Oberflächen önnen Sie die Dialogelemente oft direkt anspringen, indem Sie Alt in Kombination mit dem Buchstaben drücken, der in der Beschriftung des Elements unterstrichen ist,

wie das auch bei den Menüpunkten üblich ist. Wenn keine Zeichen unterstrichen sind, bleibt Ihnen nur die Möglichkeit, mit ⇆ oder mit den Pfeiltasten durch das Fenster zu navigieren.

Um nun ein einfach mit der Tastatur zu bedienendes Programm zu entwickeln, genügt es bei Schaltern bereits, dass Sie in der Beschriftung (*Caption*) vor das gewünschte Zeichen ein '&' setzen, um die Ansteuerungslogik automatisch zu aktivieren. Bei *Label*-Elementen verhält es sich anders, denn wenn Sie das in diesen hervorgehobene Zeichen eingeben, soll nicht die Beschriftung, sondern das beschriftete Element angesprungen werden. Da ein *Label* nicht wissen kann, zu welchem benachbarten Element es gehört, müssen Sie dieses im *Label*-Property *FocusControl* angeben.

Da die Komponente *AlarmEdit* des Beispielprogramms über keine integrierte Beschriftung verfügt, benötigen wir eine *Label*-Komponente, deren erster Buchstabe unterstrichen ist und deren *FocusControl*-Property auf *AlarmEdit* weist. Nachdem Sie die Komponente *AlarmEdit* in das Formular eingefügt haben, können Sie diese im *FocusControl*-Property der *Label*-Komponente einfach aus der aufklappbaren Liste auswählen.

Weitere Maßnahmen, die die Tastaturbedienung erleichtern, sind:

- Die Tabulatorreihenfolge der Dialogelemente sollte nachvollziehbar sein. Wenn Sie ein Dialogfenster umorganisieren, kann diese Reihenfolge ganz schön durcheinanderkommen, so dass Sie mit ⇆ kreuz und quer durch das Dialogfenster springen. Zum Umstellen der Reihenfolge verwenden Sie das Property *TabOrder* oder den Punkt TABULATORREIHENFOLGE... im lokalen oder im Bearbeiten-Menü.
- Dialoge sollten zwischen kleinen Sprüngen mit den Pfeiltasten und großen Tabulatorsprüngen unterscheiden. Hierzu verwenden Sie das Property *TabStop*. Hat dieses den Wert *True*, erreichen Sie diese Komponente mit ⇆, ansonsten überspringen Sie es mit dieser Taste und müssen stattdessen die Pfeiltasten verwenden.
- Nach dem Öffnen des Dialogs sollte das erste Element gewählt sein. Dies können Sie im Formular-Property *ActiveControl* sicherstellen.

Wegweiser

Der nächste Schritt beim Aufbau des Beispielprogramms ist die Bearbeitung der Ereignisse. Damit geht es in Kapitel 1.5 weiter, die Ereignisse des Beispielprogramms werden erst in Abschnitt 1.5.4 auf Seite 67 hinzugefügt. Die folgenden Abschnitte des Kapitels 1.4 befassen sich noch weiter mit der Programmierumgebung: mit dem Editieren von Properties, Menüs und Bildern.

Properties von mehreren Komponenten gleichzeitig verändern

Eine weitere Besonderheit des Objektinspektors sollte nicht unerwähnt bleiben: Sie können mehrere Komponenten auswählen und deren Properties gleichzeitig ändern. Der Objektinspektor zeigt dann natürlich nur noch die Properties an, die allen gewählten Komponenten gemein sind. So können Sie beispielsweise die Funktionen zum linksbündigen Ausrichten von Komponenten (siehe Ausrichtungspalette) ersetzen, indem Sie alle Elemente wählen, die an einer Linie linksbündig ausgerichtet werden sollen, und dann das Property *Left* gemeinsam ändern.

Oft gibt es jedoch effektivere Möglichkeiten, wie in diesem Fall die Ausrichten-Funktion oder in einem weiteren Beispiel: Sollen alle Komponenten mit einer anderen Schriftart versehen werden, ist es ratsam, die Schriftart des Formulars zu wechseln und darauf zu achten, dass das Property *ParentFont* bei den Komponenten auf *True* gesetzt ist, damit sie die Schriftart des Formulars übernehmen.

1.4.7 Property-Typen

Die Art und Struktur der Properties kann sehr verschieden sein. Im einfachsten Fall handelt es sich um einfache Zahlen (z.B. bei den Properties *Left/Top*, die die Position der Komponente angeben) oder Buchstabenfolgen (Strings, z.B. *Caption* für den Fenstertitel und *Name* für den Komponentenbezeichner). In manchen Fällen müssen Sie aus mehreren vorgegebenen Werten auswählen oder können ganze Listen eingeben. Um diese verschiedenen Typen bequem editierbar zu machen, wird jeder Typ auf eine spezielle Art editiert. In der Terminologie der Borland RAD-Tools wie Kylix bedeutet das: Es gibt für jeden Property-Typ einen *Property-Editor.*

Property-Editoren

Für Zahlen und Strings besteht dieser Editor aus dem Eingabefeld, in dem Sie den Wert einfach eingeben. Falls ein komplizierterer Property-Editor für ein Feld existiert, zeigt Ihnen ein Schalter neben dem Wert des Properties dies an. Mit dem Schalter rufen Sie den Property-Editor auf. Der Schalter ist allerdings nur sichtbar, wenn das Property gerade ausgewählt ist. Die meisten anderen Property-Typen entsprechen einem der Datentypen von Object Pascal, die in Kapitel 2.4 behandelt werden.

Der Zahl der verschiedenen Property-Editoren sind jedoch keine erreichbaren Grenzen gesetzt, denn Komponentenentwickler können ihren Komponenten eigene Property-Editoren beifügen. Neben dem einfachen Property-Editor, der nur aus einem Eingabefeld besteht, gibt es in Kylix für die folgenden Property-Typen eigene Editoren:

Aufzählungstypen: Hier gibt es mehrere Optionen zur Auswahl, jede Option wird über einen Namen angesprochen. Der Editor für Properties diesen Typs ist eine aufklappbare Liste, in der alle Optionen aufgeführt werden. Sie wählen eine neue

Option, wie Sie es von einer solchen Liste gewöhnt sind. Die Properties eines Formulars geben viele Beispiele für diese Aufzählungstypen: Für den *Cursor* können Sie im gleichnamigen Property eine Form auswählen, die dieser dann annimmt, wenn er sich über der freien Fläche des Formulars befindet, z.B. *crCross* für ein Zeichenkreuz, *crHourGlass* für den Sanduhr-Zeiger und *crDefault* für den normalen Mauszeiger. Manchmal ist es auch möglich, die Listenvorgaben zu ignorieren: So kann Kylix für das Property *Color* nicht alle 16,7 Millionen möglichen Farbwerte, sondern nur die am häufigsten verwendeten in einer Liste anbieten. Sie können daher mit einem Doppelklick auf das Editierfeld dieses Properties einen Farbauswahldialog öffnen oder die Farbe direkt als Zahlenwert angeben, z.B. *$FFAF55* für einen seltenen Blauton (*Color* ist allerdings aus Object-Pascal-Sicht auch kein Aufzählungstyp). In einigen aufklappbaren Listen werden auch Grafiken angezeigt, so etwa bei *Color* die Farbe oder bei *Cursor* ein Bild der gewählten Mauszeigerform.

- Referenzen auf Komponenten: In manchen Properties geben Sie eine Komponente des Formulars an, z.B. in den oben verwendeten Properties *FocusControl* und *ActiveControl*. Auch hier steht Ihnen eine aufklappbare Liste zur Verfügung, die Ihnen alle in Frage kommenden Komponenten zur Wahl stellt.

- Objekte mit geschachtelten Properties (siehe Abbildung 1.4): Wenn manche Komponenten all ihre Optionen in der Objektinspektor-Liste angäben, würde diese sehr unübersichtlich werden. Daher sind manche Properties standardmäßig nicht sichtbar, sondern einem Eintrag untergeordnet, der durch ein kleines Plus-Zeichen links neben dem Namen markiert ist. Durch einen Doppelklick auf dieses Zeichen können Sie diesen Zweig der Liste expandieren, so dass alle Unterpunkte in der Liste dargestellt werden. Zu einem Zweig zusammengefasst werden meistens Properties, die zu einem gemeinsamen Aufgabengebiet gehören, oder einzelne Flags, falls es sehr viele davon gibt (siehe nächster Punkt).

  Ein Beispiel für die erste Variante sind die *Font*-Properties vieler Komponenten wie auch des Formulars. Zu den Eigenschaften einer Schriftart gehören die Einstellungen, die Sie auch in einer Schriftauswahl-Dialogbox vornehmen können. Sie haben in diesem Fall sogar die Wahl zwischen einer manuellen Eingabe aller Attribute in die einzelnen Property-Felder und einer bequemen Auswahl aus einer Schriftauswahl-Dialogbox. Programmtechnisch handelt es sich bei den Properties wie *Font* um Objekte, die ihrerseits wieder Properties besitzen.

- Mengen von Flags (siehe Abbildung 1.4 rechts): Ebenfalls als expandierbare Zweige dargestellt werden Listen von Flags (ein- und ausschaltbare Properties), beispielsweise das Property *BorderIcons* des Formulars. In diesem Property wählen Sie aus, ob Systemmenü und Schalter zum Maximieren und Minimieren in der Titelleiste des Fensters erscheinen sollen, wobei Sie jedes Icon separat an- und abstellen können. Bei dieser Property-Art können Sie die Flags auch im Feld des übergeordneten Eintrags an- und ausschalten. In diesem werden die Flags so dargestellt wie eine

Menge in Object Pascal, beispielsweise *[biMinimize, biMaximize]*, falls diese beiden Flags eingeschaltet, *biSystemMenu* aber inaktiv ist. Um Flags einzeln umzuschalten, genügt ein Doppelklick auf das entsprechende Unterfeld (das dann als aufklappbare Liste mit den Werten *True* (»wahr«, eingeschaltet) und *False* (abgeschaltet) realisiert ist).

- Properties, die mit speziellen Dialogen editiert werden können, manchmal *nur* mit Hilfe eines Dialogs. Zum Beispiel: Properties, die einen Dateinamen angeben, haben meistens eine Dateiauswahlbox als Editor; eine Schriftwahldialogbox erleichtert die Einstellung des Properties *Font*. Sie rufen diese Dialogboxen auf, indem Sie den Schalter mit den drei Punkten rechts neben dem Wert des Properties anklicken. Weitere Beispiele für derartige Dialogboxen sind der Stringlisten-Editor (z.B. für das *Tabs*-Property der *TabSet*-Komponente oder die *Items* einer *ListBox* oder *ComboBox*) oder der Grafik-Editor (z.B. für das *Picture*-Property der *Image*-Komponenten oder das *Glyph*-Property der Bitmapschalter).

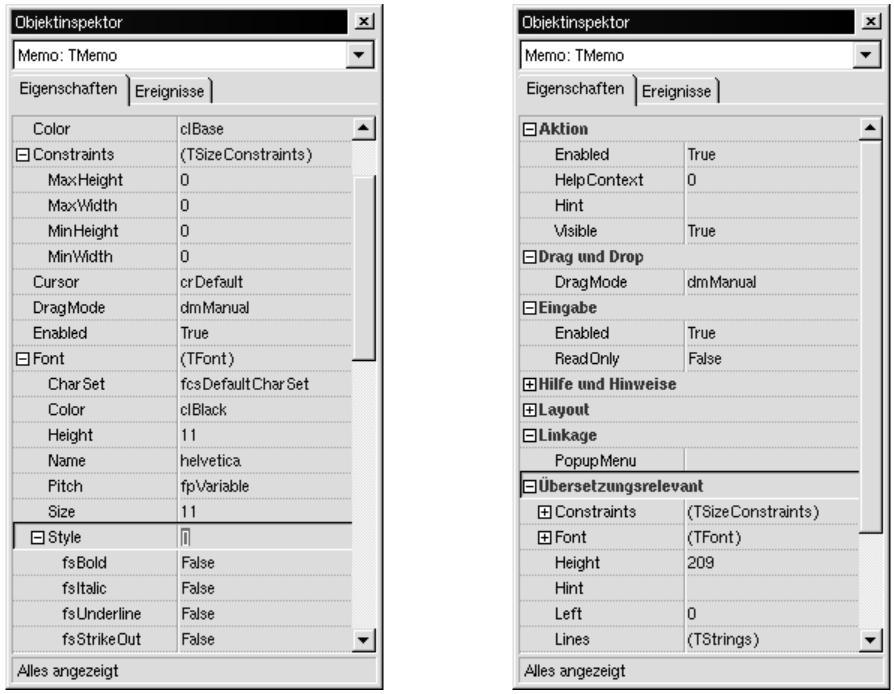


Abbildung 1.4: Geschachtelte Properties und nach Kategorien sortierte Properties

1.4.8 Menüs

Zum Editieren stellt Kylix einen Menü-Designer zur Verfügung, der auf den Menükomponenten ganz links auf der ersten Seite der Komponentenpalette basiert. Um ein Hauptmenü zu erhalten, nehmen Sie eine Komponente des Typs *TMainMenu* in Ihr Formular auf.

Der Aufbau eines Menüs verbirgt sich hinter dem Property *Items* im Objektinspektor. Um diese Eigenschaft zu verändern, d.h. also, den Menüeditor aufzurufen, können Sie entweder den zugehörigen Property-Editor aufrufen oder auf die Menükomponente im Formular doppelklicken.

Minikomponenten bilden ein Menü

Durch die Definition eines Menüs lösen Sie eine Schwemme von Minikomponenten aus: Jeder Menüpunkt ist eine eigene Komponente, die jedoch nicht mühsam mit der Maus aus einer Palette herantransportiert werden muss, sondern automatisch erstellt wird. Der Objektinspektor ist lediglich dazu da, dass Sie dort die Attribute eines Menüpunkts (ob er markiert oder grau dargestellt ist, ob es sich um eine Trennlinie handelt etc.) einstellen können, und dazu, ein Eingabefeld für den Menütext zur Verfügung zu stellen. Parallel dazu wird das Menü im Menü-Designer visuell am Bildschirm aufgebaut, so, wie es später zur Programmlaufzeit aussieht (siehe Abbildung 1.5).

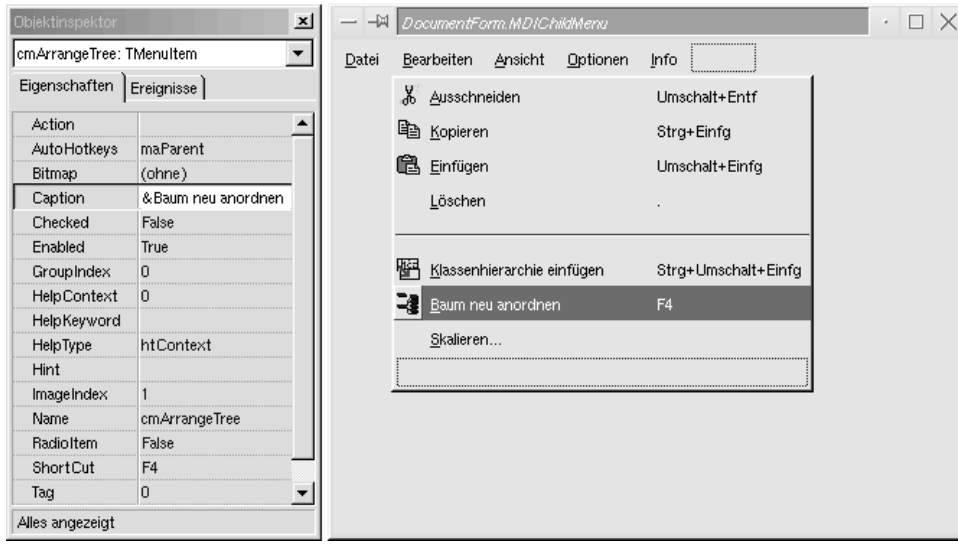


Abbildung 1.5: Der Menü-Designer

An den Stellen, wo neue Menüpunkte eingegeben werden können, zeigt der Menü-Designer zusätzlich einen leeren Menüpunkt als Platzhalter an (am Anfang gibt es nur links oben ein Exemplar davon). Im Menü-Designer können Sie später mit der Maus beliebige Punkte zur Bearbeitung auswählen oder sogar per Drag&Drop verschieben.

Effektive Bedienung des Menü-Designers

Um das gesamte Menü am schnellsten eingeben zu können, müssen Sie sich dem Wechselspiel von Objektinspektor und Menü-Designer anpassen. Angenommen, Sie haben gerade den Designer für ein neues Menü aufgerufen. Dieser ist dann standardmäßig das aktive Fenster. Geben Sie dort ein Zeichen ein, so leitet der Designer dieses automatisch an den Objektinspektor weiter, in dem praktischerweise gleich das *Caption*-Property aktiviert ist (sofern Sie vorher kein anderes Property ausgewählt haben). Wenn Sie in *Caption* den Text für den Menüpunkt eingeben und ↵ drücken, befördern Sie den eingegebenen Text automatisch in das Menü und erreichen, dass darunter ein neuer leerer Menüpunkt angelegt wird, dessen Text Sie sofort eingeben können. Nach dem Drücken von ↵ hat Kylix außerdem das Menüfenster wieder aktiviert, so dass Sie in diesem mit den Pfeiltasten zwischen den Menüpunkten navigieren können. Wollen Sie einfach nur unter dem zuletzt eingegebenen Menüpunkt weitermachen, genügt es wieder, einfach ein Zeichen einzugeben, Kylix aktiviert dann sofort wieder den Objektinspektor.

Um einen neuen Hauptmenüpunkt zu erstellen, tragen Sie einen Text in das *Caption*-Property des immer vorhandenen freien Platzhalter-Menüpunkts im Hauptmenü ein. Sobald Sie dies mit ↵ bestätigt haben, öffnet sich unter dem neuen Punkt ein neues Menü mit einem Platzhaltereintrag und der Platzhalter des Hauptmenüs wandert eine Position nach rechts.

Um verschachtelte Untermenüs zu erzeugen, wählen Sie zunächst den Untermenüpunkt aus, der zu einem weiteren Menü führen soll. Wählen Sie dann aus dem lokalen Menü des Menü-Designers den Punkt UNTERMENÜ ERSTELLEN.

Um ein bestimmtes Zeichen des Menütextes zu unterstreichen und gleichzeitig für das geöffnete Menü zu einem Tastenkürzel zu machen, geben Sie vor diesem das Zeichen &' ein.

rennlinien zur Gliederung eines Menüs erhalten Sie, indem Sie einfach als *Caption* ein -« eingeben.

Menüpunkt-Properties

Die weiteren wichtigen Editieroperationen im Menü-Designer (Löschen und Verschieben von Menüpunkten, Einfügen von Schablonen und Menü-Ressourcen, siehe lokales Menü des Menü-Designers) sollen hier aufgrund ihrer Einfachheit nicht weiter erläu-

tert werden. Interessant sind vor allem weitere Properties, die die Menüpunkte neben dem *Caption*-Property zu bieten haben:

- *Action*: Hier geben Sie optional ein Aktionsobjekt an, das beim Anwählen des Menüpunktes ausgeführt werden soll (siehe Kapitel 4.6.4).
- *AutoHotkeys* gibt es als Property für das gesamte Menu (*TMainMenu* oder *TPopupMenu*) und als Property für die einzelnen Menüpunkte (dann bezieht es sich auf ein unter diesem Menüpunkt befindliches Untermenü). *AutoHotkeys* sorgt, auf *maAutomatic* gesetzt, für eine automatische Zuweisung von Tastenkürzeln zu allen Menüpunkten des Menüs oder Untermenüs, denen Sie nicht manuell zu einem solchen Tastenkürzel verholfen haben (durch vorangestelltes '&' im *Caption*-Property). Falls ein Tastenkürzel mehrfach vorkommt, wird diese Situation durch Änderung der Duplikate bereinigt. Sie können das Property auch auf *maParent* setzen, um das Verhalten des übergeordneten Menüpunktes bzw. Menüs zu erben.
- *AutoLineReduction* verfährt wie *AutoHotkeys*, um doppelte Trennlinien (Linien, die unmittelbar auf eine andere Linie folgen) zu entfernen.
- *Bitmap* dient zur Speicherung eines Bildsymbols, das neben dem Menüpunkt eingeblendet werden soll (im Menü-Designer ist es allerdings noch nicht sichtbar). Alternative dazu: Property *ImageIndex*.
- *Break:* Hiermit können Sie Zeilenumbrüche in Menüzeilen und Spaltenumbrüche in den Spalten bewirken. Diese werden vor dem Menüpunkt eingefügt, der das entsprechende Property besitzt. Sie werden nicht im Menü-Designer, sondern im Formular dargestellt, wenn das bearbeitete Menü im *Menu*-Property des Formulars eingestellt ist.
- *Checked* sorgt, wenn es eingeschaltet ist, für die Markierung des Menüpunkts. Für die Umschaltung der Markierung zur Laufzeit müssen Sie im Programm sorgen, Beispiele finden Sie in Kapitel 6.2.4.
- *Default* macht einen Menüpunkt zur Voreinstellung eines Menüs. Sein Text wird dann durch Fettschrift hervorgehoben und er kann durch einen Doppelklick auf den übergeordneten Menüpunkt aufgerufen werden.
- *Enabled* sind per Voreinstellung alle Menüpunkte. Schalten Sie dieses Flag aus, ist der Menüpunkt nicht mehr anwählbar, aber normalerweise noch sichtbar und lesbar, falls die Farbeinstellung für deaktivierte Menüpunkte im Betriebssystem nicht anderweitig manipuliert wurde.
- *GroupIndex* spielt erst beim Verschmelzen von Menüs in MDI-Anwendungen eine Rolle und wird in Kapitel 6.7.3 erklärt.

HelpContext verknüpft den Menüpunkt über eine Kennzahl mit einer Seite der Hilfedatei, die Sie Ihrer Anwendung zuordnen können (siehe Property *Application.HelpFile*).

In *Hint* geben Sie einen Hinweistext für den Menüpunkt an, den Sie zur Laufzeit sichtbar machen können, wie in Kapitel 1.8.5 gezeigt.

ImageIndex kann einen Index in eine Bilderliste (*TImageList*) angeben und so wie das *Bitmap*-Property dazu führen, dass das Menü zur Laufzeit mit einem Bildsymbol versehen wird. Bilderlisten werden im Zusammenhang mit der *TListView*-Komponente in Kapitel 1.8.5 und genauer in Kapitel 3.6.2 erläutert.

Name wird von Kylix automatisch auf eine besondere Variation des *Caption*-Textes gesetzt, gefolgt von der obligatorischen Nummerierung. Für Änderungen des Namens gilt dasselbe wie bei den Namen aller anderen Komponenten.

RadioItem: Wenn Sie diese auf *True* setzen, wird die Markierung des Menüpunktes nicht mit einem Häkchen, sondern mit einer an einen Radioschalter erinnernden Kreisfläche dargestellt. Auf diese Weise können Sie auch in einem Menü die Funktion von Radioschaltern simulieren (siehe Komponente *TRadioGroup* bzw. *TRadioButton*). Siehe auch Kapitel 6.2.4, Abschnitt *Menümarkierung und -deaktivierung*.

Shortcut gibt das Tastenkürzel an, über das der Menüpunkt auch dann aufgerufen werden kann, wenn das Menü nicht gerade angezeigt wird. Wenn Sie hier eine Tastenkombination aus der Liste auswählen, erreichen Sie zweierlei: Die Tastenkombination wird im Menü angezeigt und zugleich als funktionsfähiges Tastenkürzel im Programm installiert.

SubMenuImages ist für Menüpunkte gedacht, die zu einem Untermenü führen. Sie weist auf eine *TImageList*-Komponente, welche die Bildsymbole enthält, die in den *ImageIndex*-Properties der Untermenüpunkte referenziert werden.

Tag ist keinem besonderen Zweck verpflichtet und steht Ihnen damit uneingeschränkt für eigene Pläne zur Verfügung. Ein Beispiel für den Einsatz der *Tag*-Properties anderer Komponenten finden Sie in Kapitel 3.5.2.

- *Visible* schaltet zwischen Sichtbarkeit und Unsichtbarkeit des Menüpunkts um.

Menübefehle ausführen

Das Einzige, was dem Menü jetzt noch fehlt, ist die Fähigkeit, die Befehle auch auszuführen. Dies geschieht über das *OnClick*-Event, das jeder einzelne Menüpunkt besitzt. Die beiden schnellsten Wege, Anweisungen einzugeben, die mit einem Menüpunkt verknüpft werden sollen, sind:

- Doppelklicken Sie im Menü-Designer auf den Menüpunkt.
- Oder wählen Sie den Punkt (zur Entwurfszeit) im Formular aus (falls das Formular das Menü nicht anzeigt, müssen Sie zuerst das Property *TForm.Menu* auf die Menükomponente setzen).

In beiden Fällen gelangen Sie direkt in den Code-Editor, in dem bereits ein automatisch erzeugter Methodenrumpf (oder eine früher schon erzeugte Methode) wartet. Mehr zum Verknüpfen von Ereignissen mit Methoden erfahren Sie im Kapitel über die Ereignisbehandlung (Kapitel 1.5).

Wichtige Menü-Themen sind auch das Erstellen, Verändern und Verschmelzen von Menüs zur Laufzeit des Programms. Sie werden in den Kapiteln 4.6, 6.2.4 und 6.7.3 besprochen.

1.4.9 Grafiken

Ebenfalls eine wichtige Rolle beim Entwurf moderner Benutzeroberflächen spielt das Erstellen von diversen Grafiken wie Bitmaps, Icons (eigentlich nur eine spezielle Größenvariante von Bitmaps) und Mauszeigerformen. Während Delphi für Windows hierfür einen einfachen Bildeditor mitbringt, verlässt sich Kylix ausschließlich auf das reichhaltige Angebot an frei verfügbaren Grafikeditoren für Linux. So dürften die meisten Linux-Distributionen bereits Editoren vom einfachen *XPaint* bis zum hochkomplexen *Gimp* enthalten.

Die beiden genannten Tools *XPaint* und *Gimp* etwa erlauben es wie der Bildeditor von Delphi für Windows, die Grafiken auch in großen Zoomstufen zu bearbeiten (was wichtig ist, wenn man in kleinen Bitmaps auf jedes einzelne Pixel zugreifen will). Sie verfügen mindestens über die vom Bildeditor bekannten Zeichenwerkzeuge und können Grafiken in den unter Kylix verwendbaren Formaten *xpm*, *png*, *gif* und *jpeg* speichern. Was sie nicht können, ist, Ressourcendateien im Windows-Format (`.res`) zu erzeugen, und diese werden von Kylix verlangt, wenn Sie Bitmaps zu ausführbaren Dateien hinzulinken wollen.

Bis auf diese Ausnahme sind sie als Ersatz des Bildeditors durchaus geeignet, zumal sie ihre teilweise sehr umfangreichen zusätzlichen Bildbearbeitungsfunktionen bescheiden in kleinen Menüs verstecken. Lediglich für ihre Mehrfenster-Bedienung erfordern sie gegenüber dem Bildeditor eine Umstellung.

Da wir uns mit der sehr unterschiedlichen Bedienungsweise der verschiedenen Editoren in diesem Kapitel nicht weiter befassen können, werden die folgenden Abschnitte voraussetzen, dass die entsprechenden Grafikdateien erstellt wurden und kurz beschreiben, wie sie in die Kylix-Anwendung eingebunden werden können.

Bitmaps

ür das Einlesen, Zeichnen und Speichern von Bitmaps in Kylix-Anwendungen ist die Qt-Bibliothek verantwortlich, auf die Sie als Kylix-Anwender jedoch nicht direkt zugreifen müssen. Eine aktuelle Liste der Formate gibt es in der Qt-Dokumentation etwa auf der Seite der Klasse *QImageIO*. Da dies neben einigen in Linux/Unix heimischen Formaten (wie *xpm*) auch das Windows-*bmp*-Format und darüber hinaus die beliebten Bildformate *png*, *gif* und *jpeg* unterstützt, kommt hier auf Benutzer, die bisher mit Delphi für Windows und der VCL gearbeitet haben, ein ungewohnter Luxus zu.

Die Beispielanwendung *TreeDesigner* aus Kapitel 6 verwendet mehrere Bitmaps für seine Mauspaletten-Schalter. Mauspaletten-Schalter haben ein Property *Glyph*, in das Sie eine Bitmap laden können, die auf der Oberfläche des Schalters gezeichnet wird. Drücken Sie im Objektinspektor den Schalter neben dem Editierfeld von *Glyph*, um eine Dialogbox zu erhalten, mit der Sie eine Bitmap-Datei laden können.

Wenn Sie in Ihrer Anwendung einfach so feste Bitmaps einbinden wollen (als Illustration, nicht als Schalter), bietet sich die Komponente *TImage* an. Wenn Sie Bitmaps selbst zeichnen wollen, können Sie dies mit *TCanvas* (Kapitel 4.4) und der Klasse *TBitmap* (Kapitel 4.5.2) tun.

Ressourcendateien

Ressourcendateien im Windows-Format (`.res`) können mehrere Grafiken im Bitmap-, Icon- oder Cursor-Format beinhalten. Unter Linux sind zwar nur die Bitmaps nötig, da sowohl Icons als auch Cursor-Formen hier durch normale Bitmaps definiert werden, aber dennoch verlangt Kylix das `res`-Format, wenn Sie Bitmaps direkt zu einem Teil der ausführbaren Datei machen wollen. Für die Funktionsweise des Programms macht es keinen Unterschied, wenn Sie die Bitmaps *nicht* als Ressource einbinden, sondern als einzelne Dateien ablegen und dann aus diesen Dateien laden (*TBitmap.LoadFromFile*). Es ist unter Linux auch durchaus üblich, dass Anwendungen ihre Bitmaps als vereinzelte Dateien speichern. Das Einbinden in die ausführbare Datei hat natürlich den Vorteil, dass dann für Anwendung und Bitmaps nur noch eine einzige Datei weitergegeben werden muss.

Hinweis: Cursor-Dateien von Windows (`*.cur`) können zwar über eine Ressourcendatei in eine Kylix-Anwendung eingebunden werden, es gibt aber keine Funktionen in Qt oder der CLX, mit denen sich eine solche Ressource nutzen ließe. Mehr Sinn macht es dagegen, Windows-Icon-Dateien (`*.ico`) in die Kylix-Anwendung einzubinden, denn die CLX-Klasse *TIcon* ist in der Lage, diese zu lesen und zu verarbeiten.

Ressourcendateien einbinden *R6*

Das TreeDesigner-Projekt aus Kapitel 6 speichert ein Fadenkreuz als eigene Mauszeigerform in einer Ressource. (Dies dient nur der Demonstration, denn eigentlich würde auch das vordefinierte Fadenkreuz ausreichen, das Sie z.B. über das *Cursor*-Property einer Komponente mit der Konstante *crCross* aktivieren können). Da Cursor-Ressourcen unter Linux nicht unterstützt werden, wurde der von der Windows-Version des TreeDesigners stammende Cursor im Bildeditor von Delphi 5 in eine Bitmap konvertiert. Zur Laufzeit muss diese Bitmap wie in Kapitel 3.3.3 gezeigt geladen werden. Dazu muss die Ressource jedoch vorher in die ausführbare Datei aufgenommen worden sein. Damit Kylix Ressourcendateien automatisch zur ausführbaren Datei hinzubindet, fügen Sie eine Zeile wie die folgende in den Pascal-Quelltext ein:

```
{$R CURSORBITMAP.RES}
```

Wichtig ist dabei, dass der Name der Ressourcendatei *nicht* mit dem Namen des Projekts übereinstimmt, denn unter diesem Namen speichert Kylix bereits eine Standard-Ressourcendatei für das Projekt, in der sich meistens nur das Icon für die Anwendung befindet.

1.5 Ereignisse

In diesem Kapitel geht es darum, wie Sie die Ereignisse bearbeiten und wie Kylix die Erstellung und Verwaltung der Ereignisbearbeitungsmethoden vereinfacht. Wie schon in Kapitel 1.1 beschrieben, sind Ereignisse die Elemente, die eine Kylix-Anwendung antreiben. Für dieses Kapitel schlagen wir also die Seite *Ereignisse* des Objektinspektors auf, auf der sich alle häufig benötigten Ereignisse befinden, die in Zusammenhang mit der gerade bearbeiteten Komponente auftreten können.

1.5.1 Einführung in die Ereignisbearbeitung

Die Ereignisse, die nur das Formular, also keine seiner untergeordneten Komponenten betreffen, finden Sie auf der Ereignisseite des Objektinspektors, wenn Sie das Formular auswählen (Abbildung 1.6). Dazu gehören Ereignisse, die der Benutzer mit seinen Eingaben direkt auslöst (z.B. *OnClick*, *OnDblClk*, *OnKeyPress* für Mausklicks und Tastatureingaben) und solche, die auch auf andere Weise hervorgerufen werden können: *OnDeactivate* tritt beispielsweise als Nebeneffekt auf, wenn der Benutzer ein anderes Fenster aktiviert, *OnPaint* wird vom System aufgerufen, wenn Fensterteile neu gezeichnet werden müssen, und das *OnDestroy*-Ereignis informiert Sie darüber, dass das Fenster gerade geschlossen wird. Das Programm selbst kann diese Ereignisse ebenfalls auslösen, indem es beispielsweise ein anderes Fenster in den Vordergrund holt, ein Fenster schließt oder andere Aktionen durchführt.

Verknüpfung von Ereignissen und Methoden

Die Spalten der Ereignisseite des Objektinspektors sind so aufgebaut, wie die der Eigenschaftsseite. Am Anfang eines Projekts ist die rechte Spalte der Ereignisse leer, d.h., dass in Ihrer Anwendung bisher auf kein Ereignis eine spezielle Antwort vorgesehen ist.

Grundsätzlich gehört in die rechte Spalte der Name eines Programmteils, der das Ereignis bearbeitet (eine *Ereignisbearbeitungsmethode*). Sie können die Ereignisse schon bearbeiten, ohne sich vorher tiefer gehend mit dem Aufbau von Methoden und der objektorientierten Programmierung beschäftigt zu haben, denn für einfache Aufgaben benötigen Sie nur sehr geringe Pascal-Kenntnisse. Um schnell eigene Experimente machen zu können, genügt es, die formlose Zusammenfassung dessen, was Sie in einer Methode tun können, in Kapitel 1.5.2 (*Schnellübersicht: Pascal für Ereignisbearbeitungsmethoden*) zu lesen. Das gesamte Potenzial der Ereignisbearbeitungsmethoden können Sie mit den Object-Pascal- und Kylix-Kenntnissen ausschöpfen, die ab Kapitel 2 vermittelt werden.

Eine Beispielmethode

Nehmen wir an, Ihre Anwendung soll auf jeden Mausklick ins Innere des Hauptfenrs reagieren. Sie benötigen also Code, der das Ereignis *OnClick* bearbeitet. Durch einen Doppelklick im Objektinspektor auf das leere Feld neben dem Ereignisnamen *OnClick* erreichen Sie, dass Kylix eine leere Bearbeitungsmethode für das Mausklick-Ereignis anlegt (Abbildung 1.6). Kylix gibt dieser Methode einen Namen und trägt ihn in das Feld des Objektinspektors ein, wo Sie ihn auch ändern können. Darüber hinaus holt Kylix den Quelltexteditor in den Vordergrund und zeigt den vorgegebenen Methodenrumpf an:

```
procedure TForm1.FormClick(Sender: TObject);
begin

end;
```

Kylix setzt auch die Eingabemarkierung bereits an die geeignete Stelle zwischen *begin* und *end*, so dass Sie sofort Anweisungen einfügen und von den mächtigen Mitteln der Sprache Object Pascal Gebrauch machen können. Da diese erst in Kapitel 2 besprochen werden, müssen wir uns an dieser Stelle etwas einschränken. Auf die (als *Parameter* bezeichnete) Angabe »*Sender: TObject*«, die Sie im so genannten *Methodenkopf* finden önnen, kommen wir in Kapitel 1.5.7 zu sprechen.

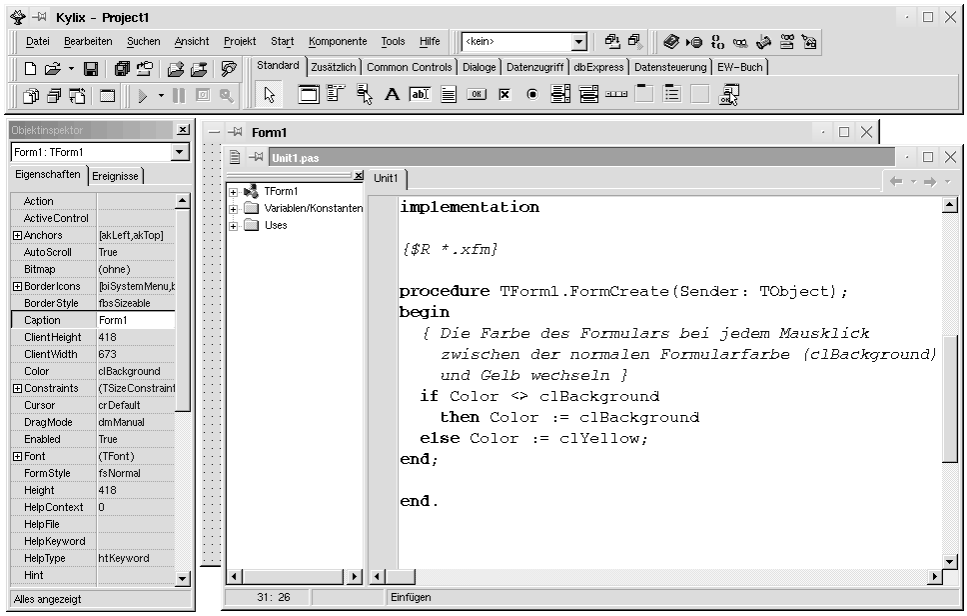


Abbildung 1.6: Eine Methode für das OnClick-Ereignis

An dieser Stelle nur ein kurzes Beispiel: Angenommen, die Farbe des Fensters soll bei jedem Mausklick zwischen Silbergrau und Weiß wechseln. Ergänzen Sie dazu den Methodenrumpf mit dem folgenden Inhalt:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  if Color <> clSilver
    then Color := clSilver
  else Color := clWhite;
end;
```

(Erklärung: Die *if*-Anweisung überprüft, welche Farbe das Fenster zurzeit hat. Dazu vergleicht sie den Wert des Properties *Color* mit dem Wert *clSilver*. Wenn das Fenster nicht die Farbe *clSilver* hat, gibt die zweite Zeile ihm diese Farbe, ansonsten wechselt die dritte Zeile zur Farbe Weiß.)

Der erste Programmstart

Sie können die Anwendung sofort starten, indem Sie F9 (Menüpunkt START | START) drücken. Ohne am Bildschirm durch Meldungen großes Aufsehen zu erzeugen, übersetzt Kylix nun Ihr Programm und schreibt den Entwurf Ihres Formulars auf die Festplatte. Da dieser Vorgang bei kleinen Programmen wie diesem sehr schnell geht, fällt er gar nicht auf (falls es doch zu merklichen Verzögerungen kommt, ist Speicherknappheit oder hohe anderweitige Prozessorauslastung der Grund).

Wenn alles funktioniert hat, können Sie testen, ob das Fenster wie gewünscht auf Mausklicks reagiert. Sie werden feststellen, dass sich die Farbe nach dem ersten Mausklick ändert, aber erst dann, wenn Sie die Maustaste losgelassen haben. Wenn Sie versuchen, eine schnelle Folge von Mausklicks auf das Fenster loszulassen, scheint es, als habe dieses Reaktionsprobleme.

Dies liegt daran, dass *OnClick* ein relativ komplexes Ereignis ist. Es definiert einen Mausklick als das Niederdrücken der Taste und das anschließende Loslassen im selben Fenster. Es berücksichtigt darüber hinaus, dass zwischen zwei Klicks keine zu kurze Zeitspanne liegen darf, denn sonst handelt es sich um einen Doppelklick, und statt eines zweiten *OnClick*-Ereignisses wird ein *OnDblClk*-Ereignis hervorgerufen (dabei kann das schon gesendete *OnClick*-Ereignis natürlich nicht mehr rückgängig gemacht werden).

Hinweis: Im Menü PROJEKT der IDE befinden sich mit *Name* COMPILIEREN, *Name* ERZEUGEN und SYNTAXPRÜFUNG VON *Name* (Dabei ist *Name* der Name des aktuellen Projekts) drei Punkte, mit denen Sie den Compiler aufrufen können, ohne das Projekt gleichzeitig zu starten. Beim Aufrufen des START-Menüpunkts verhält sich der Compiler wie beim Menüpunkt COMPILIEREN, übersetzt also nur die Dateien, die nicht mehr in einer aktuellen Übersetzung vorliegen. Genauere Informationen dazu gibt Ihnen die Online-Hilfe.

Wahl eines passenderen Ereignisses

Wir ändern das Programm nun so, dass das Fenster bei jedem Drücken der Maustaste sofort reagiert. Dazu müssen wir ein anderes Ereignis suchen, denn *OnClick* ist dazu wie gesehen nicht geeignet. Das reaktionsschnellere Ereignis heißt *OnMouseDown*. Erzeugen Sie durch einen weiteren Doppelklick im Objektinspektor einen neuen Methodenrumpf für dieses Ereignis. Dieser Methodenrumpf unterscheidet sich vom bisherigen außer im Namen durch die Parameter: Neben *Sender* finden Sie hier beispielsweise *X* und *Y*, die die Mausposition anzeigen.

Um den Rumpf zu füllen, verschieben Sie nun die Zeilen, die Sie oben in *FormClick* eingefügt haben, in die Methode *FormMouseDown* (*FormClick* sollte sich im Quelltexteditor direkt über der neuen Methode befinden), oder versuchen Sie es mit der folgenden ariation, deren neue Object-Pascal-Teile nur demonstrativen Zwecken dienen:

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
const Farbtafel: array[0..9] of TColor =
  (clMaroon, clGreen, clOlive, clTeal, clSilver,
   clRed, clBlue, clLime, clFuchsia, clAqua);
begin
  Color := Farbtafel[random(10)];
end;
```

Diese Methode wählt per Zufallszahl eine aus zehn vorgegebenen Farben aus und weist sie dem *Color*-Property zu. Und egal, wie schnell Sie klicken, die Reaktion des Programms sollte jetzt immer sofort auf das Niederdrücken einer Maustaste erfolgen.

Wegweiser

Mit dem Beispielprogramm und den Ereignissen der Komponenten geht es in Kapitel 1.5.4 weiter, zunächst folgen ein für Umsteiger interessantes Kapitel und ein Blick hinter die Kulissen der gerade besprochenen Kylix-Automatik.

1.5.2 Schnellübersicht: Pascal für Ereignisbearbeitungsmethoden

Falls Sie noch nicht in objektorientiertem Pascal, in C++ oder in einer ähnlichen Sprache programmiert haben, sollen Ihnen die beiden folgenden Aufzählungen einen groben Überblick über die Operationen geben, die Sie innerhalb einer Ereignisbearbeitungsmethode erledigen können.

Eine kleine Vorbemerkung: Sie können zwischen beliebigen Anweisungen, Wörtern und Zahlen Kommentare in den Quelltext einfügen. Kommentare müssen auf eine von zwei Arten vom Rest des Programms abgetrennt werden:

```
{ dies ist die erste Möglichkeit, einen Kommentar einzuschließen. }
(* auch so kann ein Kommentar geklammert werden *)
```

Die erste der beiden folgenden Listen beschreibt die drei verschiedenen Arten von Anweisungen, die Sie verwenden können, die zweite Liste zeigt, auf welche Elemente diese Anweisungen wirken bzw. mit welchen sie arbeiten. Die Anweisungsarten sind:

- *Zuweisung*: Eine sehr häufige Operation ist die Zuweisung eines Wertes an ein Property oder an eine Variable; für die Zuweisung verwenden Sie den Zuweisungsoperator »:=« wie im letzten Beispiel. Auf der rechten Seite der Zuweisung können Sie beliebige Ausdrücke angeben, solange diese sich mit der linken Seite vertragen (z.B. *Breite := 10*SpaltenBreite;*). Oft werden Sie sich dabei auf Properties beziehen wollen. Dies geschieht meistens wie in der gleich folgenden zweiten Liste beschrieben.
- *Methodenaufruf*: Neben der Zuweisung spielt der Aufruf von Methoden (Prozeduren und Funktionen) eine wichtige Rolle. Ein Methodenaufruf ist beispielsweise: *MessageBeep(MB_OK);*
- *Kontrollstrukturen*: Zuweisung und Methodenaufruf wären bereits das Einzige, was Sie innerhalb einer Methode tun könnten, wenn es nicht noch die Kontrollstrukturen und Schlüsselworte gäbe. So können Sie z.B. mit der *if*-Anweisung wie im letzten Beispiel Alternativen abfragen, mit der *for*-Schleife bestimmte Operationen wiederholen lassen oder gar mit *asm* Assemblercode einfügen.

Die obige Aufstellung ist zwar grob, aber umfassend (in dieser Einteilung sind dies die einzigen Arten von Anweisungen, die es gibt). Anders verhält es sich mit der folgenden Liste: Sie zeigt die Elemente, mit denen Sie am häufigsten arbeiten. Alle im folgenden genannten Elemente haben als Gemeinsamkeit, dass sie einen Namen, den so genannten *Bezeichner*, benötigen:

Um sich auf Properties des Formulars zu beziehen, schreiben Sie einfach deren Namen hin (z.B. *Color := clSilver;* für das *Color*-Property des Formulars).

Um Properties von Komponenten anzusprechen, stellen Sie den Namen der Komponente, gefolgt von einem Punkt, vor den Namen des anzusprechenden Properties, so weisen Sie z.B. mit *Panel1.Color := clWhite;* nur der Komponente mit dem Namen *Panel1* die weiße Farbe zu.

Ebenso können Sie zwischen Methoden des Objekts und einer Komponente unterscheiden: Um beispielsweise eine Komponente in den Vordergrund zu holen (entsprechend dem Befehl NACH VORNE SETZEN im lokalen Menü des Formulars), schreiben Sie *Komponente.BringToFront;*. Um das Formular/Fenster selbst vor die anderen Formulare/Fenster zu bringen, schreiben Sie nur *BringToFront;*

Um innerhalb einer Methode Werte speichern zu können, deklarieren Sie lokale Variablen. Diese sind nur in der Methode gültig und behalten ihre Werte nicht zwischen zwei verschiedenen Aufrufen dieser Methode.

Um Werte zwischen zwei Ereignissen zu speichern, geben Sie dem Formular eine neue Variable. So kann beispielsweise die *MouseDown*-Methode speichern, zu welcher Zeit die Maustaste gedrückt wurde. Wenn die *MouseUp*-Methode ihre eigene Zeit feststellt, kann sie mit Hilfe der gespeicherten Zeit der *MouseDown*-Methode die Differenz zwischen beiden Zeiten ermitteln und damit errechnen, wie lange die Maustaste gedrückt war.

- Sie könnten in Object Pascal auch jederzeit globale Variablen deklarieren, jedoch ist meistens die Deklaration einer Formularvariablen empfehlenswerter. Object Pascal macht globale Variablen nahezu unnötig.

Ein wichtiger und selbstverständlicher Bestandteil des Programmcodes sind Konstanten wie Zahlen, Zeichenketten ('In Hochkommas') und Bezeichner, die für einen konstanten Wert stehen (z.B. die Namen für die Farben *clBlack* und *clWhite*), insbesondere die Pascal-Namen für »wahr« und »falsch«: *True* und *False*.

Soweit zum Inhalt einer Methode und damit zum Mikrokosmos eines Object-Pascal-Programms. Eine Übersicht über den gesamten Quelltext, den Kylix für das letzte *OnClick/OnMouseDown*-Beispielprogramm fast vollständig automatisch erzeugt hat, finden Sie in der folgenden Beschreibung des Makrokosmos.

1.5.3 Eine Übersicht über die Formular-Unit

Bisher haben Sie nur im Inneren der Ereignisbearbeitungsmethoden mit dem eigentlichen Programmcode zu tun gehabt. Wir sehen uns hier nun den vollständigen Quelltext des Formulars an, allgemeine und weitere Informationen zu Units finden Sie in Kapitel 2.1.7.

Kylix verwaltet für jedes Formular eine Unit, die dafür vorgesehen ist, alles, was mit dem Formular untrennbar logisch verbunden ist, in einer Datei zusammenzufassen (Kylix selbst erweitert diese Unit nur um Komponenten und Methoden des Formulars; Sie können weitere Hilfsklassen, -typen, -funktionen und andere Dinge darin deklarieren). Wir betrachten zuerst die Unit, die erzeugt wird, wenn Sie einem leeren Formular eine *FormMouseDown*-Methode zuweisen wie im letzten Beispiel.

Die Unit zum Formular

Im oberen Teil der Unit befindet sich die so genannte *Deklaration* des Formulars. In ihr sind alle Bestandteile (Komponenten und eventuell weitere Variablen) und Fähigkeiten (Ereignisbearbeitungsmethoden) des Formulars aufgeführt. Das zuletzt bearbeitete Formular enthält als einziges Element die Methode für das Ereignis *FormMouseDown*:

```
unit Unit1;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms,
  QDialogs;

type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  if Color <> clSilver
    then Color := clSilver
  else Color := clWhite;
end;

end.
```

Im zweiten Abschnitt der Unit, der durch die Überschrift *implementation* eingeleitet wird, ist die konkrete Implementierung der oben angegebenen Methode(n) untergebracht. Alle Methoden, die in der Deklaration aufgelistet sind, kommen hier ein zweites Mal vor, jedoch diesmal mit dem zugehörigen Code, der in einem von *begin* und *end* eingeschlossenen Block steht.

Weitere Bestandteile dieser Unit sind:

Die Zeilen *unit...*, *interface*, *implementation* und *end.* bilden das Gerüst einer jeden Object-Pascal-Unit.

Der *uses*-Abschnitt ist notwendig, um – grob gesagt – die CLX verwenden zu können. Alle Namen, die hier aufgelistet werden, sind ebenfalls *Units*, die zum größten Teil die Funktionalität der CLX bilden.

Die Deklaration der Variable *Form1: TForm1*. Auf diese Weise wird von der Formularschablone ein Fensterobjekt hergestellt. Warum und wie ein solches nach dem Starten des Programms automatisch angezeigt wird, beschreibt das Kapitel zur Projektverwaltung (1.6.3).

- Die *$R*-Compiler-Anweisung gehört quasi zum Protokoll und sorgt für die Verbindung der Formulardatei mit der ausführbaren Datei, wenn das Programm übersetzt wird.

Sie können übrigens das Unit-Gerüst, das Kylix für alle neuen Formular-Units verwendet, ändern, indem Sie eine geänderte Formular-Unit in die Objektablage aufnehmen und beim Erstellen eines neuen Formulars kopieren (zum Thema Objektablage siehe Kapitel 1.6.4).

omponenten in der Formulardeklaration

Wenn Sie dem Formular Komponenten hinzufügen, nimmt Kylix auch diese in die Formulardeklaration auf. Der folgende Ausschnitt ist der Unit des Beispielprogramms entnommen und zeigt die Deklaration der einzelnen Komponenten seines Formulars:

```
type
  TfrmBashEditor = class(TForm)
    Panel1: TPanel;
    NurLesen: TCheckBox;
    SaveButton: TButton;
```

```
    Panel2: TPanel;
    Memo: TMemo;
    FontButton: TButton;
    FontDialog: TFontDialog;
    Label1: TLabel;
    FontComboBox: TComboBox;
    StatusLabel: TLabel;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Die Bereiche *private* und *public* sind für Ihre Formularvariablen und für Nicht-Ereignisbearbeitungsmethoden vorgesehen, die Sie beide manuell an diesen Positionen eintragen müssen.

1.5.4 Die Ereignisse des Beispielprogramms

Wir können nun die Ereignisse der Komponenten in unserem Bash-Konfigurationseditor bearbeiten. Wenn Sie das Programm ohne Ereignisbearbeitungsmethoden starten, werden Sie einen funktionsfähigen Editor vorfinden, dessen Textinhalt zu Beginn lediglich aus einer Zeile des Inhalts »Memo1« besteht. Die grundlegendste Fähigkeit, die dem Progamm bisher noch fehlt, besteht also darin, die gewünschte Datei zu laden.

Das OnCreate-Ereignis

In unserem Fall soll nicht der Benutzer die Datei wählen können (wie etwa später im TreeDesigner von Kapitel 6), sondern die Datei soll gleich bei Programmstart geladen werden. Nun gibt es im Objektinspektor zwar kein Ereignis »Programmstart«, aber ein Ereignis »Erzeugung des Formulars (zur Laufzeit)«. Es heißt *OnCreate* und tritt im Falle des Hauptformulars einer Anwendung (unsere Anwendung hat nur ein Formular, welches daher auch das Hauptformular ist) beim Programmstart auf.

Wählen Sie also das Formular im Objektinspektor aus, doppelklicken Sie auf der Seite *Ereignisse* auf das Ereignis *OnCreate* und fügen Sie eine Zeile in den automatisch erzeugten Methodenrumpf ein:

```
const
  FileName = '/root/.bashrc';

procedure TfrmBashEditor.FormCreate(Sender: TObject);
begin
  Memo.Lines.LoadFromFile(FileName);
end;
```

Die CLX liefert mit *LoadFromFile* bereits eine Methode, die eine Datei lädt, so dass wir es hier sehr einfach haben und uns nicht um das Öffnen und Schließen von Dateien

und Ähnlichem zu kümmern haben. Die einzige »Schwierigkeit« liegt darin, dass wir nicht *Memo.LoadFromFile* schreiben können, sondern *Memo.Lines.LoadFromFile*, da der extinhalt des Memos von einer Art »Unterobjekt« – eben *Memo.Lines* – verwaltet wird (Näheres dazu siehe in Kapitel 4.1.1 über die Klasse *TStrings*).

Der Dateiname wurde oben als Konstante des Namens *FileName* definiert. Dies ermöglicht es, den vorgegebenen Dateinamen leicht im Programmcode zu ändern, auch wenn er noch an anderen Stellen des Programms verwendet wird (z. B. beim Speichern der Datei). Wenn Sie nicht die Konfigurationsdatei des `root`-Benutzers editieren wollen, ändern Sie die Konstante entsprechend. Wollen Sie das Heimatverzeichnis des jeweils aktuellen Benutzers verwenden, gehen Sie nach R118 Seite 563 vor.

Aktionsschalter-Betätigung

Als Nächstes wollen wir den Schalter für das Speichern der Datei implementieren. Wenn er gedrückt wird, kommt es zu einem *OnClick*-Ereignis. Analog zu *LoadFromFile* ür das Laden rufen wir nun *SaveToFile* auf. Um die Methode etwas anspruchsvoller zu gestalten, fragen wir vorher noch ab, ob überhaupt Änderungen gemacht wurden. Das *Memo*-Feld merkt sich dies automatisch in seinem Property *Memo.Modified*. Wenn keine nderungen vorliegen, soll unsere Methode mit *ShowMessage* ein Meldungsfenster anzeigen. Wenn Änderungen gemacht wurden, dann wird mit *SaveToFile* gespeichert und das Property *Modified* wird wieder auf *False* zurückgesetzt (da das *Memo* dies beim Speichern nicht automatisch erledigt):

```
procedure TfrmBashEditor.SaveButtonClick(Sender: TObject);
begin
  if Memo.Modified = True then begin
    Memo.Lines.SaveToFile(FileName);
    Memo.Modified := False; // seit dem Speichern nicht geändert
  end else
    ShowMessage('Keine Änderungen zu speichern!');
end;
```

Beachten Sie, dass die Datei unter Linux nur gespeichert werden kann, wenn die ensprechenden Schreibrechte für den Benutzer des Programms vergeben wurden.

Als Nächstes soll der Schalter für die Schriftauswahl implementiert werden. Sein *OnClick*-Ereignis wird mit Hilfe der *FontDialog*-Komponente implementiert, die wir am Anfang in das Formular eingefügt haben:

```
procedure TfrmBashEditor.FontButtonClick(Sender: TObject);
begin
  FontDialog.Font := Memo.Font;
  if FontDialog.Execute then
    Memo.Font := FontDialog.Font;
end;
```

Die erste Zeile sorgt dafür, dass der Dialog die aktuelle Schriftart des Memos (*Memo.Font*) als Voreinstellung anzeigt. In der zweiten wird der Dialog aufgerufen (*FontDialog.Execute*) und in der dritten wird die Schriftart des Memos auf die im Dialog gewählte Schriftart gesetzt. Dieser Schritt wird aber nur ausgeführt, wenn der Dialog mit dem OK-Schalter geschlossen wurde. Dies ist dadurch erkennbar, dass *FontDialog.Execute* den Wert *True* zurückgibt, was oben mit der *if*-Anweisung abgefragt wird.

Änderung von CheckBox und ComboBox-Auswahl

Die nächsten beiden Ereignisse betreffen das Ändern einer Einstellung, zunächst die Einstellung »Nur Lesen« der entsprechenden CheckBox. Auch hier heißt das Ereignis *OnClick*. Ob eine Checkbox markiert ist, können Sie im Code über ihr *Checked*-Property abfragen. Das Beispielprogramm stellt dann entsprechend das *ReadOnly*-Property der Memo-Komponente ein:

```
procedure TfrmBashEditor.NurLesenClick(Sender: TObject);
begin
  if NurLesen.Checked then Memo.ReadOnly := True
    else Memo.ReadOnly := False;
// kürzer geht es so:  Memo.ReadOnly := NurLesen.Checked;
end;
```

Um die Hintergrundfarbe des Memos zu ändern, kann der Benutzer unseres Beispielprogramms einen von fünf Einträgen aus der aufklappbaren Liste auswählen. Damit das Programm sofort auf eine Änderung diesbezüglich reagiert, wird das *OnChange*-Ereignis der *FontComboBox* bearbeitet und darin der Index des gewählten Eintrages (erste Zeile = Index 0, fünfte Zeile = Index 4) abgefragt:

```
procedure TfrmBashEditor.FontComboBoxChange(Sender: TObject);
begin
  case FontComboBox.ItemIndex of
    0: Memo.Color := clBlack;
    1: Memo.Color := clRed;
    2: Memo.Color := clGreen;
    3: Memo.Color := clBlue;
    4: Memo.Color := clYellow;
  end;
end;
```

Die Programmierhilfen im Editor

Wenn Sie den oben gezeigten Text selbst eingeben, erhalten Sie per Voreinstellung kleine Hilfestellungen bei der Eingabe. Selbst bei der einzeiligen Methode *FormCreate* werden bereits die beiden wichtigsten der so genannten Programmierhilfen aktiviert, vorausgesetzt, dass Sie an den entsprechenden Stellen des Quelltextes die kurzen Pausen machen, durch die diese Funktionen erst aktiviert werden.

Im Beispiel von *FormCreate* öffnet der Editor eine DropDown-Liste, sobald Sie den Punkt hinter *Memo* eingegeben haben. Die Liste enthält alle Properties und Methoden des Memos, darunter auch das Property *Lines*, das wir hier benötigen. Sie brauchen in diesem Fall *Lines* nicht ganz auszuschreiben, sondern können schon nach der Eingabe von »L« mit der Eingabetaste den dann in der Liste ausgewählten Text *Lines* in den Editor übernehmen.

- Wenn Sie nach der öffnenden Klammer hinter *LoadFromFile* wieder einen Moment zögern, erscheint ein kleines Hinweisfenster, das die Namen und Typen der Parameter der Methode *LoadFromFile* auflistet. Dabei hebt es immer den Parameter, den Sie seiner Meinung nach als Nächstes eingeben sollten, durch Fettschrift hervor. Im vorliegenden Fall ist das noch nicht besonders umwerfend, da es nur einen Parameter gibt – im Falle der für kurze Rückfragen an den Benutzer häufig verwendeten Funktion *MessageDlg* ist das jedoch eine ganz andere Sache. Selbst der Autor dieses Buchs, der *MessageDlg* schon seit mehr als fünf Jahren regelmäßig verwendet, freut sich noch immer über die automatisch erscheinende Erinnerung an die Reihenfolge der Parameter.

Starten des Programms

Sie können das Programm jetzt wieder über den Menüpunkt START | START oder die zugehörige Tastenkombination starten. Wenn Sie zur Laufzeit die Schriftart entsprechend anpassen, können Sie ein Fenster erhalten, das aussieht wie in Abbildung 1.3 gezeigt.

1.5.5 Methoden für die Statuszeile

Bisher noch ohne Funktion geblieben ist im Beispielprogramm das Label-Element zur Anzeige der aktuellen Eingabeposition im Editor. Der von diesem Label angezeigte ext muss offensichtlich immer dann aktualisiert werden, wenn sich auch die Eingabeposition im Editor ändert. Es stellt sich also auch hier wieder die Frage nach dem passenden Ereignis. Allerdings gibt es diesmal mehr als eine Antwort, denn der Benutzer kann die Eingabeposition über die Tastatur ändern oder indem er mit der Maus eine andere Position anklickt. Das Beispielprogramm bearbeitet daher die Ereignisse *OnMouseDown* und *OnKeyDown* (eine Taste der Tastatur wird gedrückt, für weitere Informationen zu den Tastaturereignissen siehe Kapitel 3.3.2).

Sehen wir uns zunächst die Methode für das *OnKeyDown*-Ereignis an:

```
procedure TfrmBashEditor.MemoKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  Status: String;
begin
  Status := Format('  %d / %d', [Memo.CaretPos.Line, Memo.CaretPos.Col]);
```

```
    if Memo.Modified then Status := Status + '  (ungespeichert)';
    StatusLabel.Caption := Status;
  end;
```

Die vordefinierte *Format*-Prozedur, auf die in Kapitel 2.8.4 näher eingegangen wird, baut hier zunächst einen String nach dem Muster »33 / 2« auf (für »Zeile 33, Spalte 2«), der dann, falls das Memo seit dem letzten Speichern editiert wurde (*Memo.Modified*), noch um einen entsprechenden Hinweis erweitert. Schließlich wird er dem *Caption*-Property des Labels zugewiesen.

Nun sollen die gerade beschriebenen Aktionen auch dann stattfinden, wenn das *OnMouseDown*-Ereignis auftritt. Wir könnten dazu die obigen Zeilen einfach in den leeren Rumpf einer *OnMouseDown*-Methode kopieren, was aber ein sehr unelegantes Duplikat zur Folge hätte. Müssten wir dann nicht immer, wenn wir die *OnKeyDown*-Methode ändern, auch an die Kopie in *OnKeyDown* denken? Dies gibt uns eine gute Gelegenheit, einmal eine Methode zu schreiben, deren Gerüst nicht automatisch von Kylix erzeugt wird. Und zwar wird das Aktualisieren des Memos in eine eigene Methode namens *UpdateMemoStatus* ausgelagert.

Schreiben eigener Methoden

Die Beispielmethode soll zwei Parameter erhalten: einen für das Memo-Feld, dessen Position angezeigt werden soll, und einen für das Label-Element, das für die Anzeige zuständig ist. Zwar gibt es im vorliegenden Beispielprogramm für beides nur eine Möglichkeit und die Parameter sind daher noch nicht notwendig, jedoch könnte das Programm mit einer parametrisierten Methode leicht auf mehrere Memo-Felder und mehrere Statusanzeige-Labels erweitert werden.

Unter der Annahme, dass Ihnen die Grundprinzipien der Verwendung von (je nach Programmiersprache) Funktionen, Prozeduren, Methoden, Unterprogrammen etc. bekannt sind, sei zu den genauen technischen Details der Methodendeklarationen in Object Pascal auf Kapitel 2 verwiesen. An dieser Stelle soll es hauptsächlich um die praktische Anwendung gehen. Das folgende Listing zeigt die drei Stellen der Formular-Unit, die von der neuen Methode betroffen sind:

- In der Klassendeklaration des Formulars wird die Methode im *private*-Bereich genannt, weil sie nur innerhalb der Klasse bzw. innerhalb der Formular-Unit selbst aufgerufen werden soll. Wichtig ist, dass Sie die eigenen Methoden nicht *vor* dieser ersten *private*-Überschrift deklarieren, denn das ist der von Kylix automatisch verwaltete Bereich.
- Der Code der Methode gehört in den *implementation*-Teil der Unit. Das Methoden-Gerüst, bestehend aus dem Kopf *procedure TfrmBashEditor.UpdateMemoStatus(...)* und dem *begin...end* können Sie von Kylix automatisch erzeugen lassen, indem Sie

die Klassenvervollständigungs-Funktion nutzen. Drücken Sie dazu innerhalb der Klassendeklaration (also z.B. direkt nachdem Sie den Methodenkopf in der Klassendeklaration eingegeben haben) einfach Strg+⇧+C.

- Der Aufruf der Methode findet in diesem Beispiel zweimal statt: Einmal bei Tastatureingaben in der Methode *MemoKeyDown* und einmal bei Mausklicks in der Methode *MemoMouseDown*:

```
type
  TfrmBashEditor = class(TForm)
    Panel1: TPanel;
    ... (von Delphi erzeugte Deklarationen)
    ...
  private
    { Private declarations }
    procedure UpdateMemoStatus(M: TMemo; ALabel: TLabel);
  public
    { Public declarations }
  end;

implementation

...

procedure TfrmBashEditor.UpdateMemoStatus(M: TMemo; ALabel: TLabel);
var
  Status: String;
begin
  Status := Format('  %d / %d', [M.CaretPos.Line, M.CaretPos.Col]);
  if M.Modified then Status := Status + '  (ungespeichert)';
  ALabel.Caption := Status;
end;

procedure TfrmBashEditor.MemoKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  UpdateMemoStatus(Memo, StatusLabel);
end;

procedure TfrmBashEditor.MemoMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  UpdateMemoStatus(Memo, StatusLabel);
end;
```

Verknüpfen mehrerer Ereignisse mit einer Methode

Eine Besonderheit an diesem Beispiel liegt darin, dass sowohl *MemoKeyDown* als auch *MemoMouseDown* aus der gleichen Anweisung bestehen. Normalerweise würde man vielleicht versuchen, diese Anweisung nur einmal schreiben zu müssen. Und Kylix

unterstützt ein solches Ansinnen auch dadurch, dass es Sie mehrere Ereignisse mit derselben Methode verknüpfen lässt.

In diesem Beispiel ist dies jedoch nicht möglich, da *MemoKeyDown* und *MemoMouseDown* über verschiedene Parameterlisten verfügen. Wenn Sie jedoch die Statusanzeige auch beim Loslassen der Maustaste aktualisieren wollen, können Sie im Objektinspektor unter *OnMouseUp* die bestehende Methode *MemoMouseDown* auswählen, da *OnMouseUp* und *OnMouseDown* übereinstimmende Parameterlisten aufweisen. (Bei der Gelegenheit könnten Sie übrigens auch noch *OnMouseMove* bearbeiten, so dass die Statusanzeige laufend aktualisiert wird, während der Benutzer mit der Maus einen Textbereich markiert. *OnMouseMove* bedarf aufgrund seiner speziellen Parameter jedoch wieder einer neuen Methode.)

1.5.6 Weiterer Progamm-Ausbau

In diesem Abschnitt wird unser erstes Beispielprogramm durch ein paar kleine Erweiterungen abgerundet.

Exceptions

Zunächst geht es darum, einen selten auftretenden Fehler abzufangen. Es kann nämlich sein, dass die Datei, die von *Memo.Lines.LoadFromFile* (siehe Kapitel 1.5.4) geladen werden soll, gar nicht existiert. *LoadFromFile* reagiert auf diese Situation, indem sie eine Exception (Ausnahmebedingung) des Typs *EFOpenError*-Exception hervorruft, was letztendlich zum Abbruch der Methode und zur Anzeige einer Fehlermeldung durch die CLX führt (wenn auf Ihrem System eine `.bashrc`-Datei vorhanden ist, können Sie die *FileName*-Konstante des Programms ändern, um den Fehler hervorzurufen).

Eine detaillierte Beschreibung der Funktionsweise von Exceptions lesen Sie in Kapitel 2.6. In diesem Fall geht es darum, die Exception abzufangen, damit keine Fehlermeldung angezeigt wird, sondern einfach ein leeres Memo-Feld dargestellt wird (beim Speichern des Memo-Inhalts wird dann automatisch eine neue Datei erzeugt).

Zur Behandlung der Exception wird die *FormCreate*-Methode durch die strukturierte Anweisung *try...except...end* wie folgt erweitert:

```
procedure TfrmBashEditor.FormCreate(Sender: TObject);
begin
  try
    Memo.Lines.LoadFromFile(FileName);
  except
    on E: EFOpenError do;
  end;
end;
```

Alle Exceptions, die im *try...except*-Block auftreten, werden an den *except*-Block weitergeleitet und – sofern sich eine Zeile findet, in der der Typ der Exception (hier *EFOpenError*) in einer *on*-Klausel genannt ist – dort behandelt. Die Behandlung besteht in diesem Fall einfach aus einer leeren Anweisung (dem Semikolon), da die Exception ja gerade ignoriert werden soll.

Flexibleres Formular-Design

Das bisherige Formular-Design ist statisch und nicht dafür ausgelegt, dass der Benutzer die Größe des Fensters zur Laufzeit ändert. Zwar kann der Benutzer die Fenstergröße verändern, aber bei einer Vergrößerung erhält er lediglich ungenutzte Fensterfläche am rechten bzw. unteren Rand, und bei einer Verkleinerung schneidet er Komponenten am Rand ab.

Hinweis: Wenn ein Fenster so unkooperativ auf Größenänderungen reagiert, sollte man die Größenveränderbarkeit eventuell ganz abschalten, indem man das *BorderStyle*-Property des Formulars etwa auf *fbsDialog* stellt oder nach R**19** Seite 342 vorgeht.

Das Formular soll nun so erweitert werden, dass sich die Größe des Memos automatisch der Größe des Fensters anpasst. Und die Komponenten am unteren Fensterrand sollen zwar nicht ihre Größe ändern, aber sich mit dem unteren Rand des Fensters bewegen, statt starr an ihrer Position zu bleiben.

Die meisten Komponenten der CLX verfügen zu solchen Zwecken über ein Property Namens *Align*. Wird *Align* auf einen der Werte *alLeft*, *alTop*, *alRight* oder *alBottom* eingestellt und ist die Komponente direkt dem Formular untergeordnet, so erstreckt sie sich automatisch über die gesamte Fensterhöhe bzw. -breite. Bei *alLeft* beispielsweise wird die Komponente am linken Fensterrand platziert, und wenn sich die Höhe des Fensters verändert, passt sich die Komponente automatisch daran an.

Wenn nun eine oder mehrere Komponenten auf diese Weise an den Fensterrändern angeordnet sind, verbleibt in der Mitte des Formulars normalerweise (spätestens, wenn das Formular vergrößert wird) ein freier Bereich. Der *Align*-Wert *alClient* führt dazu, dass eine Komponente sich automatisch dieses gesamten freien Bereichs bemächtigt.

Neue Formularversion mit Panels

Oft werden die *Align*-Einstellungen in Verbindung mit *TPanel*-Komponenten verwendet, so etwa in der neuen Version des Beispielprogramms, die Sie auf der CD unter dem Namen *BashRCEditor2* finden (Abbildung 1.7).



Abbildung 1.7: Die neue Programmversion zur Entwurfszeit

Das Panel hinter dem Memo und dem Status-Label (in der Abbildung durch die acht Markierungspunkte des Formulareditors hervorgehoben) ist mit *alClient* ausgerichtet, der darunter liegende Bereich wird von einem *alBottom*-ausgerichteten Panel eingenommen. Innerhalb des oberen Panels ist wiederum das Status-Label mit *alBottom* ausgerichtet und das Memo mit *alClient*. Da Memo und Status-Label Kindelemente des Panels sind, werden sie auch relativ zu diesem ausgerichtet und nicht etwa relativ zur gesamten Fensterhöhe.

Fragt sich nur noch, wie die Komponenten des Beispielprogramms zu Kindelementen der beiden Panels gemacht werden können. Beim Neueinzeichnen von Komponenten verhält es sich einfach so: Die Komponenten werden zum Kind eines Panels, wenn Sie sie auf der Fläche des Panels einzeichnen.

Die Komponenten des Beispielprogramms wurden aber im Formular eingezeichnet und sind daher diesem als Kind zugeordnet. Eine Veränderung der Eltern von Komponenten ist beim Formularentwurf nur möglich, indem Sie die Komponenten ausschneiden (BEARBEITEN | AUSSCHNEIDEN), die neue Elternkomponente markieren und die Komponenten wieder einfügen (BEARBEITEN | EINFÜGEN). Beachten Sie dabei, wenn Sie das Hauptmenü verwenden, dass vorher das Formular das aktive Fenster ist. Wenn stattdessen der Objektinspektor aktiv ist, wird dort der aktuelle Text ausgeschnitten bzw. eingefügt.

Um das Beispielprogramm zu aktualisieren, können Sie also zunächst die Komponenten für das untere Panel ausschneiden, das Panel einfügen und seinen *Align*-Wert auf *alBottom* einstellen. Dann können Sie die ausgeschnittenen Komponenten wieder einfügen (achten Sie darauf, dass vor dem Einfügen das Panel markiert ist, in welches die

Komponenten eingefügt werden sollen). Das Gleiche wiederholen Sie dann mit dem zweiten Panel sowie Memo und Status-Label.

Kapitel 3.5.1 wird noch einmal etwas genauer auf das Thema der Schachtelung von Steuerelementen eingehen.

Hinweis: Beim Einfügen von Komponenten in eine andere Elternkomponente verschiebt sich ihre Position abhängig von der linken oberen Ecke der neuen Elternkomponente. Zusätzlich stellt Kylix sicher, dass die eingefügten Komponenten nicht über den Rand der neuen Elternkomponente hinausragen, indem es ihre Position gegebenenfalls anpasst. Das kann zu unerwünschten Effekten führen, dass z. B. nach dem Einfügen alle Komponenten am unteren Rand der neuen Elternkomponente übereinander gestapelt sind. Sie können dies vermeiden, indem Sie die Komponenten schon vor dem Ausschneiden innerhalb der bisherigen Elternkomponente an eine Position schieben, die auch noch im Koordinatensystem der neue Elternkomponente (relativ zu deren linken oberen Fensterecke) im sichtbaren Bereich gelegen ist.

Sicherheitsabfrage

Als Letztes soll nun noch eine Sicherheitsabfrage implementiert werden, die dem Benutzer beim Schließen des Formulars die Gelegenheit gibt, ein versäumtes Speichern nachzuholen (oder den mit dieser Sicherheitsabfrage rechnenden Benutzer zufrieden stellt, der absichtlich nicht speichert, weil er lieber gefragt werden will). Gesucht ist also ein Ereignis, dass vor dem Schließen des Fensters auftritt, in dessen Bearbeitungsmethode das Schließen jedoch noch verhindert werden kann (wenn der Benutzer den Schalter *Abbrechen* wählt).

OnCloseQuery ist ein Ereignis mit den gewünschten Eigenschaften; es wird im Beispielprogramm wie folgt bearbeitet:

```
procedure TfrmBashEditor.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
  CanClose := True;
  if Memo.Modified then
    case MessageDlg('Memo wurde verändert, speichern?',
        mtConfirmation, [mbYes, mbNo, mbCancel], 0) of
      mrYes: Memo.Lines.SaveToFile(FileName);
      mrCancel: CanClose := False;
    end;
end;
```

Um die Frage, ob die Änderungen gespeichert wurden, müssen wir uns nur wenig ümmern, da das Memo bereits von sich aus darüber Buch führt: Bei Änderungen setzt es sein *Modified*-Property automatisch auf *True*. Und beim Speichern der Datei in Kapitel

1.5.4 haben wir bereits dafür gesorgt, dass es dann auch wieder auf *False* zurückgesetzt wird. Der obigen Methode bleibt also nur noch, *Modified* abzufragen und eventuell mit der vordefinierten Funktion *MessageDlg* zu fragen, ob noch gespeichert werden soll:

Neben dem eigentlichen Text geben die Parameter von *MessageDlg* an, welches Symbol und welche Schalter im Meldungsfenster erscheinen sollen. Zu den zur Verfügung stehenden Optionen siehe die Online-Hilfe. In diesem Beispiel werden drei Schalter für *Ja*, *Nein* und *Abbruch* verwendet. Welcher dieser Schalter gedrückt wurde, kann anhand des Rückgabewerts von *MessageDlg* überprüft werden. So steht etwa *mrYes* für *Ja* (Schalter *mbYes*), *mrCancel* für *Abbruch*.

Um das Schließen des Fensters abzubrechen, genügt es, die Variable *CanClose*, die der *OnCloseQuery*-Methode ja als Parameter überliefert wird, auf *False* zu setzen.

Hinweis: Die obige Methode berücksichtigt noch nicht den Fall, dass es beim Speichern der Datei zu einem Fehler kommt, denn dann sollte das Programm natürlich auch nicht beendet werden, damit der Benutzer eventuell die Fehlerursache beheben und noch mal speichern kann. Hierzu könnte die *EWriteError*-Exception abgefragt werden wie in einem früheren Listing bereits die Exception *EFOpenError*. Im Falle der Exception sollte dann eine Fehlermeldung ausgegeben und *CanClose* auf *False* gesetzt werden; siehe hierzu auch R62 Seite 702.

1.5.7 Ereignisverknüpfung: Navigieren, verändern, lösen

Das einfache Anlegen einer Ereignisbearbeitungsmethode ist nicht die einzige Aufgabe, bei der der Objektinspektor Ihnen behilflich ist. Auch Änderungen der Verknüpfung, das Löschen von Ereignissen und das Navigieren zwischen Ereignissen werden von Kylix besonders unterstützt. Dazu kommt noch, dass jede Komponente über ein *Standardereignis* verfügt.

Das Standardereignis

Ein Standardereignis ist ein Ereignis, das besonders häufig bearbeitet wird. Zu diesem Ereignis können Sie in Kylix besonders schnell eine Methode erstellen – ohne Umweg über den Objektinspektor gelangen Sie mit einem Doppelklick auf die Komponente direkt vom Formular in den Quelltexteditor zu einer Methode für das Standardereignis. Einige Beispiele:

- Das Standardereignis des Formulars ist *OnCreate*, das bei der Öffnung des Fensters zur Laufzeit aufgerufen wird.
- Ein Doppelklick auf eine *Edit*-Komponente befördert Sie zu einer Methode für das Ereignis *OnChange*, das bei jeder Änderung des Benutzers im Editierfeld erzeugt wird.

- *OnClick* ist naheliegenderweise das Standardereignis der *Button*-Komponente und einiger anderer Komponenten.

Navigieren zwischen Ereignissen

Wenn bereits eine Methode für ein Ereignis existiert, bringt Sie ein Doppelklick in das entsprechende Methodenfeld des Objektinspektors schnell zur bestehenden Methode im Code-Editor. Das erspart Ihnen viel Sucharbeit im Quelltext, wenn Sie viele Komponenten mit vielen Ereignissen haben: Während die Methoden dort völlig unsortiert und ungegliedert quasi auf einer einzigen langen Seite aufgezählt sind, sind sie im Objektinspektor logisch und schnell auffindbar nach Komponenten geordnet.

Die Ereignisverknüpfung ändern und löschen

Sie können den Namen der Ereignismethode im Objektinspektor verändern. Wenn Sie einen noch nicht existierenden Namen eingeben, ändert Kylix den Namen der Methode an den automatisch erzeugten Stellen des Quelltextes (in der Deklaration des Formulars und in der Definition der Methode im Implementationsteil). Haben Sie den alten Namen an weiteren Stellen selbst verwendet, müssen Sie ihn dort selbst anpassen.

Statt eines neuen Namens können Sie auch eine schon existierende Methode angeben oder besser noch aus der aufklappbaren Liste des Objektinspektors auswählen. Der orteil dieser Liste ist auch, dass sie nur die Methoden anzeigt, die mit dem Ereignis kompatibel sind (die also die erforderlichen Parameter für dieses Ereignis aufweisen). Wenn Sie eine existierende Methode angeben, ändert Kylix die Verknüpfung, löscht aber die bisher verknüpfte Methode nicht.

Auf diese Weise ist es möglich, mehrere Ereignisse aus einer oder aus verschiedenen Komponenten mit einer einzigen Methode zu verknüpfen. Um dann innerhalb der Methode zwischen den möglichen Komponenten zu unterscheiden, fragen Sie den Parameter *Sender* ab.

Dieser bezeichnet die Komponente, von der das Ereignis ausgelöst wurde. *Sender* ist auch schon das erste Beispiel für ein polymorphes Objekt (siehe Kapitel 2), denn obwohl sein Typ als *TObject* deklariert ist, kann es sich bei *Sender* um eine Komponente der verschiedensten Typen handeln.

Eine sehr häufige Verwendung von *Sender* ist die Abfrage von *Tag*-Properties, wie sie in Kapitel 3.5.2 demonstriert wird. Die einfachste Möglichkeit, *Sender* zu verwenden, besteht jedoch in einem direkten Vergleich mit den Komponenten des Formulars:

```
if Sender = Button1 then ...
else if Sender = Button2 then ...
...
```

Es ist übrigens auch nachträglich (zur Programmlaufzeit) möglich, Methoden mit Ereignissen zu verknüpfen oder Verknüpfungen zu verändern, genauso, wie Sie nachträglich die anderen Properties im Programm ändern können, denn die Information, welche Methode zu einem bestimmten Ereignis gehört, *ist* ein Property.

Ebenso einfach, wie Sie die Verknüpfung ändern können, können Sie sie auch löschen. Entfernen Sie dazu einfach den Methodennamen aus dem Feld des Objektinspektors. Wie schon beim Ändern der Verknüpfung wird dadurch die bisher verknüpfte Methode nicht gelöscht.

Automatisches Entfernen leerer Ereignismethoden

Es gibt einen besonders schnellen Weg, Ereignisbearbeitungsmethoden vollständig zu löschen. Löschen Sie dazu einfach alle Zeilen zwischen dem *begin* und dem *end* der zu löschenden Methode, so dass nur noch der von Kylix automatisch erzeugte Methodenrumpf übrig bleibt, und speichern Sie die Datei. Beim Speichern entdeckt Kylix solche leeren Methoden und entfernt den leeren Rumpf, die Deklaration der Methode in der Formulardeklaration und alle Ereignisverknüpfungen im Objektinspektor, die auf diese Methode weisen. Selbstverständlich würde es Kylix niemals wagen, andere Methoden, die Sie nicht über den Objektinspektor verknüpft haben, zu entfernen, nur weil sie leer sind.

1.5.8 Drei Blicke hinter die Kulissen

Dieses Kapitel beschäftigt sich mit den Zusammenhängen zwischen visuellem Formular und Quelltext und klärt dabei die folgenden Fragen:

- Welche Änderungen nimmt Kylix automatisch an der Formular-Unit vor und welche Voraussetzungen müssen erfüllt sein, damit Kylix dazu in der Lage ist?
- Was passiert, wenn Sie Komponenten nachträglich löschen oder umbenennen?
- Wie behandelt Kylix die automatisch erzeugten Bereiche bei der Übersetzung und können manuelle Änderungen an diesen zu Fehlern führen?

Wenn ein Programmierwerkzeug die Quelltextdateien des Programmierers verändert, besteht immer dann die Gefahr eines Konflikts, wenn das Programmierwerkzeug bestimmte Bereiche für sich reserviert, die der Programmierer nicht verändern soll. Kylix verändert die zu den Formularen gehörenden Units in vorsichtiger Weise, und falls Sie automatisch erzeugte Zeilen ändern, können Sie Fehler hervorrufen, die aber schnell behoben werden können.

nderungen mit den visuellen Tools

Das Einfügen neuer Komponenten läuft normalerweise problemlos ab. Wenn Sie Formular und Quelltexteditor nebeneinander sichtbar machen und dann eine Komponente ins Formular einfügen, sehen Sie sofort, welche Änderungen Kylix am Quelltext vornimmt (in diesem Fall fügt es ein Feld zur Formularklasse hinzu). Vor jeder dieser nderungen durchsucht Kylix den Quelltext nach der Deklaration des Formulars. Solange Sie also die Formularklasse im Quelltext nicht entfernen oder umbenennen, ürften Sie von Kylix keine Beschwerden erhalten. Andererseits können Sie ausprobieen, den Namen des Formulars im Quelltext zu ändern und dann eine Komponente einzufügen. Kylix wird Ihnen mitteilen, dass die Deklaration der Formularklasse fehlt oder fehlerhaft ist, und es wird die Komponente nicht einfügen. Beruhigend bei diesem Test ist, dass Kylix es beim nächsten Einfügen einer Komponente wieder von neuem versucht und dass alles wieder funktioniert, wenn Sie die Formulardeklaration wieder in den Ursprungszustand zurückversetzt haben.

Es kommt wohl selten vor, dass man auf Anhieb allen Komponenten zufrieden stellende Namen gibt und genau die Komponenten und Ereignisse auswählt, die man braucht. Nun wäre es sehr unbequem, wenn diese Änderungen – nachträgliches Umbenennen und Löschen von Komponenten und Ereignissen – von Hand im Code-Editor vollzogen werden müssten. Sehen wir uns zuerst das Löschen an.

öschen von Komponenten

Um herauszufinden, wie Sie eine Komponente aus einem Formular löschen können, brauchen Sie sich nur auf die üblichen Handgriffe zu besinnen: Markieren Sie die Komponente und drücken Sie die Taste Entf. Was geschieht nun, wenn Sie für diese Komponente bereits Ereignismethoden geschrieben haben? Selbstverständlich kann Kylix diese nicht einfach mitlöschen, denn Sie könnten viel Arbeit darin investiert haben und wollen Teile des Codes vielleicht an anderer Stelle wiederverwenden. Kylix ührt daher nur die folgenden Aktionen aus:

Es löscht die Komponente aus dem Formular und aus der Formulardeklaration. Wenn Sie beispielsweise das Memo-Feld des Beispielprogramms löschen, entfernt Kylix die Zeile

```
Memo: TMemo;
```

Damit sind auch alle Einstellungen, die Sie im Objektinspektor vorgenommen haben, gelöscht, also auch die *Verknüpfung* zwischen Ereignissen und Methoden, aber wie schon gesagt nicht die Methoden selbst. Sie können die Löschung schnell über BEARBEITEN | RÜCKGÄNGIG komplett rückgängig machen.

Umbenennen von Komponenten und Methoden

Zum Umbenennen von Ereignisbearbeitungsmethoden im Objektinspektor siehe Kapitel 1.5.7, *Die Ereignisverknüpfung ändern und löschen*. Was vielleicht häufiger vorkommt als das Umbenennen von Methoden, ist das Umbenennen von Komponenten, denn während man die von Kylix vorgegebenen Komponentennamen (*Button1*, *Edit1*,...) am Anfang vielleicht für ausreichend hält, hätte man nach einigen Erweiterungen doch gerne eine bessere Unterscheidungsmöglichkeit für *Button1*, *Button2*, *Button3* und *Button4*.

Wenn Sie nun den Namen einer Komponente im Objektinspektor verändern, bemüht sich Kylix nicht sonderlich, den Code der Unit entsprechend anzupassen. Falls Sie es allerdings beim vorgegebenen Namen für die mit den Komponentenereignissen verknüpften Methoden belassen haben (dieser Name enthält den Namen der Komponenten, z.B. *Button1Click*), so ändert Kylix den Namen im Objektinspektor und im Code entsprechend. Aus *Button1Click* könnte so z.B. der Name *SucheLaserDiscClick* werden. Wie schon beim eigenhändigen Ändern der Methodennamen (siehe Kapitel 1.5.7) müssen Sie die Vorkommnisse des Komponentennamens im Programmtext manuell ändern, da der Editor die Syntax zu wenig kennt, als dass er die Bezeichner selbst austauschen könnte, ohne gleiche Bezeichner aus verschiedenen Gültigkeitsbereichen zu verwechseln.

Wenn Sie den Namen eines Formulars ändern, passt Kylix auch den Namen des Formulars in den Kopfzeilen der selbst definierten Methoden an:

```
  TForm1.PrivateMethode;
{ wird zu }
  TArtikelAuswahlDialog.PrivateMethode;
```

Innerhalb dieser Methoden ersetzt Kylix die Vorkommnisse des Namens *TForm1* und *Form1* genauso wenig wie die Namen der Komponenten, wenn Sie diese ändern.

Änderungen im reservierten Bereich

Der reservierte Bereich der Formulardeklaration beginnt direkt nach der ersten Zeile, die das Schlüsselwort *class* enthält. Er beginnt mit der Deklaration aller Komponenten, die Sie zur Entwurfszeit in das Formular eingefügt haben, und endet mit der Deklaration der Ereignisbearbeitungsmethoden, die Kylix automatisch erzeugt hat. Der reservierte Bereich ist mit der ersten *private*-Überschrift in der Formulardeklaration beendet. Bei jeder Übersetzung überprüft Kylix diesen reservierten Bereich in besonderer Weise:

- Kylix testet, ob dieser reservierte Bereich Variablendeklarationen enthält, die zu keiner der Komponenten gehören. Falls das der Fall ist, fragt Kylix Sie, ob diese Deklaration entfernt werden soll. Falls Sie aus Versehen eine Formularvariable im

reservierten Bereich anstatt in den Bereichen *private* und *public* deklariert haben, wählen Sie *Nein* und verschieben Sie die Deklaration an die richtige Stelle. Ansonsten handelt es sich womöglich um eine Komponentendeklaration, zu der das Formular keine Komponente mehr enthält. Ein solcher Fall wäre dann vorstellbar, wenn Sie das Formular nicht visuell ändern, sondern die `xfm`-Datei in den Quelltexteditor laden und dort eine Komponente löschen.

Wenn es umgekehrt im Formular eine Komponente gibt, die in der Formulardeklaration nicht aufgeführt ist, macht das meistens überhaupt nichts, solange Sie nicht versuchen, diese Komponente im Programm anzusprechen. Es ist auch möglich, eine einmal entfernte Komponentenvariable wieder in den reservierten Teil der Formulardeklaration einzufügen. Solange Sie dieser einen Namen geben, der mit dem *Name*-Property der Komponente übereinstimmt, funktioniert die Verbindung von Komponente und Variable zur Laufzeit problemlos. (Es ist jedoch wichtig, dass mindestens eine Variable von jeder verwendeten Komponentenklasse deklariert wird, da der Linker die Klasse sonst nicht in die ausführbare Datei bindet, was zur Laufzeit zu einem Fehler beim Laden des Formulars führt.)

- Bei den Methoden ist es umgekehrt wie bei den Komponenten: Überzählige Methoden werden von Kylix ignoriert, fehlende aber reklamiert. Fehlende Methoden sind die, die im Objektinspektor mit einem Ereignis verknüpft sind, die aber »heimlich« (d.h. nicht automatisch von Kylix) aus der Unit gelöscht worden sind.

Wie gesagt kann es zu solchen Differenzen nur kommen, wenn Sie per Hand die reservierten Bereiche ändern. Bei Änderungen im Formular passt Kylix die Unit automatisch an.

1.6 Bearbeiten von Projekten

Dieses Kapitel beschäftigt sich mit den Werkzeugen, die Sie bei der Arbeit mit Projekten beliebiger Größe unterstützen. Da Sie die Werkzeuge selbst mit beliebigen Projekten ausprobieren können, findet in diesem Kapitel keine Arbeit mit Beispielprogrammen statt. Es werden die folgenden Werkzeuge behandelt:

der integrierte Quelltexteditor,

die Projektverwaltung,

die Objektablage,

der Projekt-Browser,

- die ToDo-Listen.

1.6.1 Der Quelltext-Editor

Die grundsätzliche Bedienung des in die Kylix-IDE integrierten Editors entspricht dem, was man von anderen unter grafischen Oberflächen heimischen Editoren gewöhnt ist (die Einschränkung auf die »grafischen Oberflächen« soll in dieser Formulierung dazu dienen, Verwechslungen mit speziellen Unix/Linux-Editoren wie *vi* zu vermeiden). Dieser Abschnitt befasst sich nicht mit einfachen Editieroperationen, sondern konzentriert sich auf die Bereiche der so genannten Programmierhilfen und der AppBrowser-Funktionen wie CodeExplorer und Hyper-Links im Quelltext.

Genauere Information zur Tastenbelegung und zu Menüpunkten des Editors (Kontext-Menü und BEARBEITEN-Menü) gibt Ihnen die Online-Hilfe. Eine Übersicht über die Tastenbelegungen finden Sie beispielsweise, wenn Sie die Hilfe zur Dialogseite *Editor* unter TOOLS | EDITOROPTIONEN aufrufen. Um keine versteckten Fähigkeiten des Editors zu übersehen, sollten Sie sich auf dieser Seite auch die einzelnen Optionen des Editors genauer ansehen.

Die AppBrowser-IDE

Mit der Einführung von Delphi 4 hat Borland mehrere neue Funktionen unter dem marketingtauglichen Begriff des *AppBrowsers* zusammengefasst. Natürlich sind diese Funktionen nun auch unter Kylix zu finden. Die Vorteile dieser Funktionen beginnen schon bei ihrer leichten Erlernbarkeit, aufgrund derer sie hier nur kurz aufgezählt werden müssen, obwohl sie mit dem gewaltigen Nutzen, den sie uns Kylix-Anwendern bringen, eigentlich eine sehr ausführliche Erläuterung verdient hätten:

- Beim *CodeExplorer* handelt es sich um das Baumansichtsfenster, das nach einer Neuinstallation von Kylix per Voreinstellung am linken Rand des Editorfensters angedockt wird. Dieses Fenster gibt Ihnen einen Überblick über die in der aktuellen Quelltextdatei enthaltenen Programmobjekte, aufgeteilt in die vier Kategorien Klassen, Variablen/Konstanten, Typen sowie eingebundene Units. Die Sortierung der einzelnen Listen und die Verwendung weiterer Unterkategorien (z.B. *private*, *protected*, *public*) können Sie in den Umgebungsoptionen auf der Seite EXPLORER einstellen. Mit einem Doppelklick auf ein Symbol springen Sie vom CodeExplorer direkt an die Deklaration des Symbols im Quelltext (Tastenkürzel: ⇧+Strg+E). Im Kontextmenü finden Sie außerdem einen Menüpunkt, über den Sie den Namen des Symbols ändern können. Im Falle von Methoden ist dies eine nützliche Funktion, da Kylix den Namen automatisch in der Deklaration *und* in der Definition aktualisiert. Sie können im Code-Explorer sogar neue Symbole definieren (Popup-Menüpunkt NEU), müssen sich aber an die korrekte Deklarationssyntax halten, sonst wird Ihre Eingabe einfach gelöscht (das abschließende Semikolon muss allerdings nicht angegeben werden).

- Klassenvervollständigung ist eine grafisch unauffällige Funktion, die Sie mit dem Tastenkürzel ⇧+Strg+C aufrufen. Wenn sich die Eingabemarkierung zu diesem Zeitpunkt in der Deklaration einer Klasse befindet, fügt Kylix bei Bedarf leere Methodenrümpfe für alle von Ihnen deklarierten Methoden dieser Klasse in den Implementationsteil der Unit ein. Wenn Sie mehrere Klassen in einer Unit definieren, berücksichtigt Kylix sogar die Zwischenüberschriften wie etwa »{ TClass2 }« für alle Methoden der Klasse *TClass2* und ordnet die Methoden unter der passenden Zwischenüberschrift ein.

  Wenn Sie die Klassenvervollständigung von einem Methodenrumpf aus aufrufen, stellt sie wieder zuerst die Klasse der Methode fest, die sich an der Eingabeposition befindet, und arbeitet dann umgekehrt zum oben beschriebenen Fall: Sie deklariert alle noch nicht deklarierten Methoden dieser Klasse, die Sie schon in den Implementationsteil geschrieben haben. Kylix nimmt übrigens an, dass es sich um private Methoden handelt, und deklariert sie als *private.*

  Ob auch unvollständige Property-Deklarationen vervollständigt werden sollen – mitsamt einer privaten Variablen zur Speicherung und einer Property-Schreibmethode zum Beschreiben des Wertes –, können Sie ebenfalls unter EXPLORER einstellen (UNVOLLST. EIGENSCHAFTEN VERVOLLST.).

  Auch zur Navigation zwischen der Deklaration einer Methode in der Klassendeklaration und dem Methodenrumpf gibt es nun Tastenkürzel: ⇧+Strg+↓ zum Springen nach unten (zum Methodenrumpf) und ⇧+Strg+↑ in umgekehrter Richtung.

  Bei gedrückter Steuerungstaste werden außerdem alle Bezeichner im Editor, über die Sie die Maus halten, zu einem hervorgehobenen Hyperlink. Wie in einem Web-Browser springen Sie durch einen Klick zur Definition des Symbols, sofern Kylix den entsprechenden Quelltext finden kann bzw. der Quelltext überhaupt vorliegt. In der rechten oberen Ecke des Editorfensters finden Sie sogar Schalter zum Hin- und Herblättern zwischen verschiedenen Sprungpositionen inklusive einer aufklappbaren History-Liste. Leider berücksichtigt diese Liste keine Sprünge, die auf eine andere Weise gemacht wurden.

- Sie können das Sprungziel des Hyperlinks auch in Erfahrung bringen, ohne den Link zu verfolgen. Setzen Sie dazu den Mauszeiger über einen Bezeichner, *ohne* Strg zu drücken. Kylix zeigt dann die Deklarationsposition des Symbols, über dem sich der Mauszeiger befindet, in einem Hinweisfenster an, z.B. »var Name – unit1.pas (48)«.

Hinweis: Der CodeExplorer markiert standardmäßig alle in der Klassendeklaration genannten Methoden, zu denen er noch keine Definition finden kann, durch fette Schrift. Dadurch sehen Sie, welche Methoden vervollständigt werden, wenn Sie die Klassenvervollständigungs-Funktion aufrufen. Die Fett-Markierung können Sie in den CodeExplorer-Einstellungen auch unterbinden.

CodeExplorer-Einstellungen

Die Einstellungen für den CodeExplorer finden Sie in den Umgebungsoptionen auf der Seite EXPLORER, Sie können sie aber auch direkt aus dem Kontextmenü des Explorers mit EIGENSCHAFTEN aufrufen (Abbildung 1.8). Einige auf dieser Seite zu findenden Optionen beziehen sich nur auf den CodeExplorer, andere nur auf den in Kapitel 1.6.5 beschriebenen Browser, wieder andere gelten für beide zugleich. Die Option UNVOLLST. EIGENSCHAFTEN VERVOLLST. bezieht sich wie erwähnt auf die Klassenvervollständigungs-Funktion.

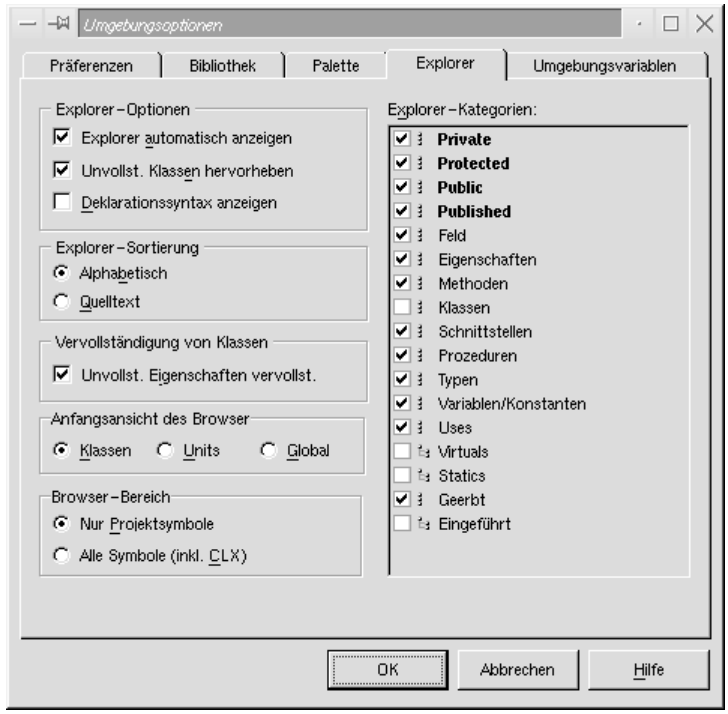


Abbildung 1.8: Die Optionen für CodeExplorer, Browser und Klassenvervollständigung

Interessant sind vor allem die folgenden Einstellungen, die auch für den Browser gelten:

DEKLARATIONSSYNTAX ANZEIGEN ist per Voreinstellung abgeschaltet und bewirkt, dass Sie eine augenschonende, aber wenig informative Liste der Symbolnamen erhalten – weder erfahren Sie die Typen von Variablen noch die Parameter und Funktionsergebnisse von Methoden. All das ändert sich, wenn Sie diese Option einschalten.

Die EXPLORER-SORTIERUNG kann entweder alphabetisch sein oder sich an der Reihenfolge der Symbole im Quelltext orientieren.

- In den EXPLORER-KATEGORIEN können Sie angeben, welche Arten von Symbolen der Explorer in einem »Ordner« (Baumknoten) zusammenfassen soll. Sind beispielsweise die Kategorien *private* und *protected* gewählt, *public* und *published* aber nicht, so werden alle als *public* und *published* deklarierten Symbole direkt unter dem Klassen-Knoten aufgelistet (sofern nicht andere gewählte Kategorien dagegensprechen). Um *private*- und *protected*-Symbole zu sehen, müssten Sie in diesem Beispiel erst die entsprechenden durch Ordner-Icons dargestellten und mit *private* und *public* beschrifteten Unterknoten öffnen.

  Wenn Sie eine Kategorie nicht in einem Knoten zusammenfassen, hat das zwar den Vorteil, dass Sie einen bestimmten Symbolnamen schneller finden können, aber Sie erhalten auch keinerlei Informationen mehr, welcher Kategorie das Symbol nun angehört.

  Um sich einen Teil des Mausklickens zu ersparen, können Sie auch angeben, welche Kategorien standardmäßig expandiert werden sollen. Klicken Sie dazu in der Kategorie-Liste auf das unscheinbare graue Symbol neben dem Markierungsfeld.

Programmierhilfen

In Kapitel 1.5.4 sind uns schon zwei der »Programmierhilfen« über den Weg gelaufen, die in der amerikanischen Originalversion unter der Bezeichnung *Code Insight* geführt werden, welche treffend darauf hindeutet, dass der Editor eine besondere »Einsicht« in den Programmcode erreicht hat: Bei eingeschalteten Programmierhilfen arbeitet der Compiler im Hintergrund und aktualisiert laufend die internen Symbolinformationen. Dabei arbeiten die Programmierhilfen offenbar mit einer anderen Technik als der AppBrowser, denn sie können im Gegensatz zu den AppBrowser-Funktionen nicht während des Debuggens genutzt werden.

Code Insight« besteht aus den folgenden Funktionen, die Sie in den Umgebungsoptionen auf der Seite *Programmierhilfe* wiederfinden (siehe Abbildung 1.9):

- *Code-Vervollständigung:* zum Aufklappen einer Liste der Unterelemente eines Objekts, dessen Namen Sie gerade in den Editor geschrieben haben und hinter den Sie einen Punkt gesetzt haben. So erweitern Sie z.B. die Eingabe »Paintbox.« per Listenauswahl zu »Paintbox.Canvas«. Ein Beispiel zur Anwendung dieser und der folgenden Funktion finden Sie in Kapitel 1.5.4.
- *Code-Parameter:* zur Anzeige, welche Parameter zum Aufruf der Funktion erforderlich sind, deren Namen Sie gerade in den Editor geschrieben haben oder hinter deren Namen Sie die Eingabemarke gesetzt haben.
- *Auswertung durch Kurzhinweis:* zur Anzeige des Wertes einer Variablen beim Debuggen, wenn Sie den Mauszeiger über die Nennung dieser Variablen im Editor halten. Bisher mussten Sie, um einen Variablenwert einmalig abzufragen, ein Dialogfenster öffnen und dort den Namen der Variablen eingeben; jetzt müssen Sie nur noch den Mauszeiger an die richtige Stelle halten.
- *Nicht im Dialog aufgeführt:* Mit Strg+␣ erhalten Sie eine meistens sehr unübersichtliche Liste aller Symbole, die an der aktuellen Eingabeposition verwendet werden können. Die Eingabeposition kann hier nach einem Zuweisungszeichen := sein oder innerhalb einer Parameterliste für eine Funktion.

Bei der Funktion *Auswertung durch Kurzhinweis* müssen Sie bedenken, dass diese Funktion keine *with*-Blöcke berücksichtigt, also nicht immer den Wert des Symbols anzeigt, das mit dem Wort unter dem Mauszeiger wirklich gemeint ist. Wenn Sie z.B. innerhalb einer Methode eines Formulars *Form1* geschrieben haben

```
with Button1 do Width := Width+10;
```

und den Mauszeiger beim Debuggen über *Width* halten, so zeigt Kylix Ihnen den Wert von *Form1.Width* an und nicht etwa den Wert von *Button1.Width*, der an dieser Stelle tatsächlich gemeint ist.

Hinweis: Kleine Experimente mit dem Debugger, wie gerade mit *with Button1 do Width:=Width+10* gezeigt, funktionieren oft nur, wenn Sie die Optimierung des Compilers abschalten, siehe hierzu auch Kapitel 1.7.1.

Abbildung 1.9 zeigt die entsprechenden Einstellungen im Optionsdialog (TOOLS | EDITOR-OPTIONEN). Die ersten vier Funktionen werden per Voreinstellung automatisch aufgerufen. Unter Umständen kann es aber auch bequemer sein, die *Code-Vervollständigung* abzuschalten und sie manuell per Strg+␣ aufzurufen, nachdem Sie einen Qualifizierungspunkt eingegeben haben (z.B. »Button1.«).

Der größte Teil der in Abbildung 1.9 gezeigten Dialogbox soll hier nur der Vollständigkeit halber erwähnt werden, denn er hat nichts mit den Symbolinformationen des Compilers zu tun, sondern ist eine völlig unspektakuläre Funktion, die es schon seit

Jahren in allen Arten von Editiersoftware gibt: Das Einfügen von häufig benötigten extbausteinen per Tastendruck. In Kylix läuft das folgendermaßen ab: Sie geben beispielsweise ein »if« ein, drücken das Tastenkürzel für die Quelltextschablonen (Strg+J) und der Editor fügt dann im günstigsten Fall bereits die restlichen zum bisher eingegebenen Konstrukt gehörenden Wörter ein. Im Falle von »if« gibt es in Object Pascal jedoch mehrere Auswahlmöglichkeiten. Von diesen müssen Sie dann aus einer Liste eine wählen.

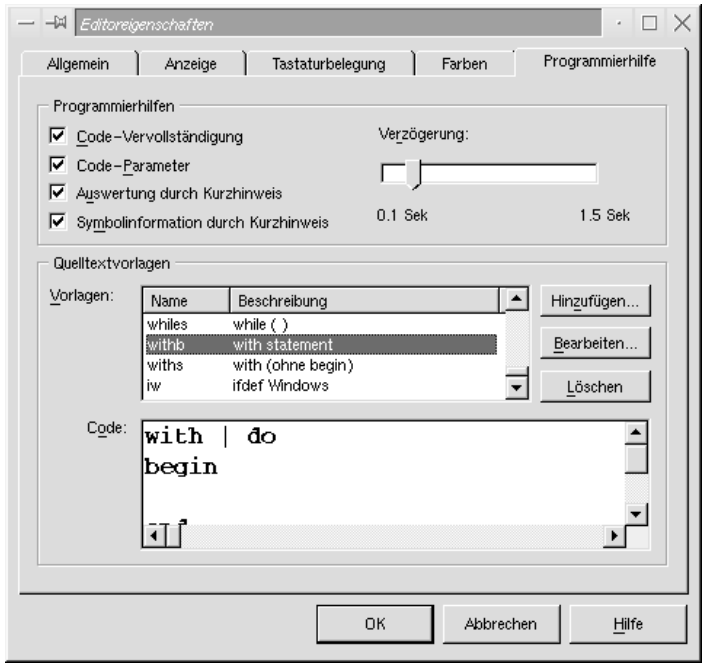


Abbildung 1.9: Die Einstellungen zu den Programmierhilfen des Editors

Jede Quelltextschablone hat auch ein Kürzel, mit dem Sie sich die Auswahl aus einer Liste ersparen (geben Sie z.B. statt *if* das Kürzel *ifeb* ein und drücken dann Strg+J, so erhalten Sie sofort die ausführlichste Version einer *if*-Anweisung in Object Pascal). Schließlich gehört zu jeder Schablone auch noch ein festgelegter Punkt innerhalb der Schablone, auf den der Cursor nach dem Einfügen gesetzt wird, also die erste Lücke, die Sie noch per Hand ergänzen müssen. All diese Optionen stehen Ihnen zur Verfügung, wenn Sie in der Dialogbox aus Abbildung 1.9 eigene neue Schablonen erzeugen. Die genannten Schablonen mit *if* im Namen sind bereits von Borland vordefiniert, wie auch weitere Schablonen für andere Object-Pascal-Konstrukte. Wenn Sie ein Backup Ihrer eigenen Schablonen machen wollen, kopieren Sie die Datei `<home-verzeichnis>/.borland/delphi60dci`.

Wichtige Tastenkürzel

Die folgende Tabelle listet die bis hier erwähnten wichtigen Tastenkürzel auf, denn einige davon sind nicht aus dem Kontextmenü des Editors ersichtlich und können leicht in Vergessenheit geraten:

Tastenkürzel	Wirkung
⇧ + Strg + C	Vervollständigen der Klasse an der Eingabeposition (kann in Interface und in Implementation aufgerufen werden)
⇧ + Strg + ↑	Springen zur Deklaration der gerade bearbeiteten Methode
⇧ + Strg + ↓	Springen zum Rumpf der Methode, deren Deklaration sich an der Eingabeposition befindet
Strg + ␣	Expliziter Aufruf der Programmierhilfen
Strg	Umschalten in den Hyperlink-Modus – Mausklicks auf Symbolnamen im Quelltext bewirken Sprung zur Deklaration des Symbols
Strg + J	Einfügen einer Code-Schablone
⇧ + Strg + E	Umschalten zwischen Code-Editor und Code-Explorer

1.6.2 Aufbau von Projekten

Bei einfachen Projekten mit nur einem Formular brauchen Sie sich noch keine Gedanken um die Verwaltung des Projekts zu machen. Zur Speicherung und zur Benennung von Dateien genügen hier noch die Erklärungen von Kapitel 1.4.5. Sobald weitere Formulare hinzukommen, werden die Eigenschaften der Projektverwaltung interessant. Wir untersuchen zuerst den Aufbau eines Projekts.

Die Dateien eines Projekts

Zur Erstellung einer Anwendung in Kylix sind mehrere Dateien notwendig, die zu einem Projekt zusammengefasst werden. Dieses können Sie dann als Ganzes über das DATEI-Menü laden und speichern. Grundlage für die Aufteilung in Dateien sind die Formulare. Kylix erstellt für jedes Formular, das Sie erzeugen, eine eigene Object-Pascal-Datei. Diese Datei wird auch als *Modul* bezeichnet, welches in Object Pascal den speziellen Namen *Unit* hat. Sie können auch eigene Units erstellen, die nichts mit Formularen zu tun haben, sondern nur dazu dienen, Ihr Programm in überschaubare Einheiten aufzuteilen.

Jedes Kylix-Projekt besitzt, wie herkömmliche Pascal-Projekte, nicht nur eine Vielzahl gleichberechtigter Units, sondern darüber hinaus ein Hauptmodul, das nach dem Programmstart als Erstes die Kontrolle erhält (das Hauptmodul wird nicht als Unit bezeichnet).

In verschiedenen Computersprachen gibt es Projektdateien, die dem Entwickler und dem Entwicklungssystem einen Überblick über alle Module des Projekts geben. In Kylix ist es nicht notwendig, dafür eine eigene Datei anzulegen: Das Hauptmodul *ist* die Projektdatei und trägt daher die Endung `.dpr` (für DelphiPRoject). Sie wird von Kylix automatisch verwaltet, kann aber auch vom Programmierer verändert werden. Einsicht in die Projektdatei erhalten Sie nach Auswahl des Menüpunktes PROJEKT | UELLTEXT ANZEIGEN.

Alle Units tragen die Endung `.pas`. Darüber hinaus muss Kylix den Aufbau der Formulare speichern und verwendet dazu binäre Dateien, die seit dem Erscheinen von Kylix mit zwei verschiedenen Endungen auftreten: Die Endung `.dfm` steht für DelphiForM und wurde bis Delphi 5 ausschließlich verwendet. Die Endung `.xfm` wird standardmäig von Kylix verwendet, wobei das »X« für »Cross Platform« steht. Sie können auch in Kylix die Endung `.dfm` verwenden (dies ergibt sich beispielsweise bei Portierung eines Projekts von Windows von selbst), wobei Kylix unabhängig von der Endung immer denselben Inhalt in die Datei schreibt. In zukünftigen Versionen von Delphi für Windows (ab Delphi 6) erhält die Unterscheidung zwischen den beiden Endungen eine neue Bedeutung: `.xfm` steht dann für eine Datei, die die CLX als Grundlage hat, `.dfm` für ein Formular, das aus VCL-Komponenten besteht. Wenn in diesem Buch die Rede von XFM-Dateien ist, können Sie normalerweise auch .DFM-Dateien dafür einsetzen.

ür jedes Projekt gibt es also eine einzige `.dpr`-Datei und für jedes Formular ein Dateipaar, bestehend aus einer `.xfm`-Datei und einer `.pas`-Unit. Dies sind die Dateien, die standardmäßig im Fenster der Projektverwaltung (Abbildung 1.10) angezeigt werden. Alle anderen Dateien wie Units und Bitmap-Ressourcen werden durch Verweise in der *uses*-Anweisung der Units oder durch einen *$R*-Compilerbefehl in das Projekt eingebunden.

Vorübersetzte Units

Damit Kylix nicht jedes Mal alle Units neu übersetzen muss, legt es für jede Unit eine Datei mit der Endung `.dcu` an, die die zur Unit gehörige Übersetzung enthält und sehr schnell mit anderen `dcu`-Dateien zur ausführbaren Datei zusammengebunden werden kann. `dcu` steht für *Delphi Compiled Unit*. Die `dcu`-Dateien werden im selben Verzeichnis gespeichert wie die ausführbaren Dateien, und zwar in dem Verzeichnis, das Sie in den Projektoptionen auf der Verzeichnisseite unter AUSGABEVERZEICHNIS angeben.

erwandt mit den `dcu`-Dateien sind übrigens die `dpu`-Dateien (Delphi Package Unit), die der Compiler beim Kompilieren eines Packages erzeugt.

Nicht-visuelle Units in der Projektverwaltung

Damit die von Ihnen verwendeten Nicht-Formular-Units ebenfalls in der Projektverwaltung aufgelistet werden, können Sie sie explizit darin aufnehmen. Ein entsprechender Schalter befindet sich in der Standard-Symbolleiste; aber auch im Menü PROJEKT

und im lokalen Menü der Projektverwaltung sind entsprechende Punkte zu finden. Nicht-visuelle Units, die Sie mit DATEI | NEU erzeugen, werden übrigens automatisch in das aktive Projekt eingefügt.

Obwohl Sie auch diese Units im Projektfenster aufnehmen können, ist es nicht der Sinn der Projektverwaltung, den absoluten Überblick über das Projekt zu geben. Für eine vollständige Übersicht über wirklich alle im Programm benutzten Units können Sie den in die IDE integrierten Browser verwenden (siehe Kapitel 1.6.5).

1.6.3 Die Projektverwaltung

Die Projektverwaltung von Kylix kann auch mehrere Projekte zu einer Projektgruppe bündeln, die im Projektmanager als oberster Knoten der hierarchischen Anzeige in Erscheinung tritt (Abbildung 1.10). Eine Gruppe aus mehreren Projekten kann zur Erzeugung verschiedener Zieldateien verwendet werden, z.B. zur Erzeugung von zwei ausführbaren Dateien und einer Shared Library. Auf der Buch-CD finden Sie als Beispiel dafür die fünf Versionen des Beispielprogramms aus Kapitel 1.8 in der Projektgruppe `Kapitel1.8.bpg` (ein Projekt mit fünf ausführbaren Dateien als Zieldateien).

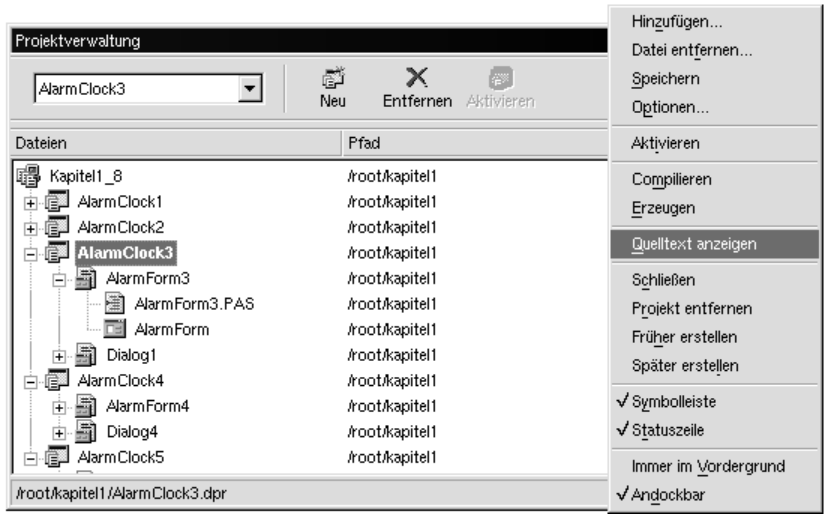


Abbildung 1.10: Die Projektverwaltung

Projektdateien

Um den Projektmanager zum Vorschein zu bringen, verwenden Sie den Menüpunkt ANSICHT | PROJEKTVERWALTUNG. Bei einem einfachen Projekt finden Sie dort unter einem Wurzelknoten, der per Voreinstellung den Namen *ProjektGroup1* trägt, die Zieldatei des Projekts, üblicherweise eine ausführbare Datei. Diesem Eintrag untergeord-

net sind alle Module, die an diesem Projekt beteiligt sind. Für ein Modul, das ein Formular enthält, finden Sie wiederum zwei Untereinträge, einen für die Formulardatei und einen für den Object-Pascal-Quelltext.

Jeder Knoten für eine Zieldatei entspricht einem einzelnen Projekt. Zu jedem Projekt gibt es genau einen Projektquelltext, der die Endung `.dpr` trägt und den Sie über den Menüpunkt PROJEKT | QUELLTEXT ANZEIGEN einsehen können. Ein typischer Projektquelltext ist der des bisherigen Beispielprogramms:

```
program BashRCEditor;

uses
  QForms,
  MiniRCEditor in 'MiniRCEditor.pas' {frmBashEditor};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TfrmBashEditor, frmBashEditor);
  Application.Run;
end.
```

Die Optionen, die in der Projektdatei festgelegt werden, sind:

Mindestens jede Formular-Unit wird per *uses*-Klausel in die DPR-Datei eingebunden und findet sich auch in der Liste des Projektfensters wieder.

Neben einer solchen Unit befindet sich der Name des Formulars, das in dieser Unit enthalten ist, und zwar in Kommentarklammern. Diese Kommentare unterscheiden sich von normalen Kommentaren dadurch, dass die Projektverwaltung sie *auswertet*. Nur über diese Kommentare kann die Projektverwaltung erkennen, dass zu einer Unit auch ein Formular gehört.

- Für jedes Formular, das automatisch erstellt werden soll, ist im Hauptprogramm ein Aufruf von *Application.CreateForm* zu finden. Im Dialog PROJEKT | OPTIONEN können Sie dies auf der Seite *Formulare* verändern, indem Sie die Formulare zwischen den Listen *Autom. Formularerstellung* und *Verfügbare Formulare* austauschen. Standardmäßig wird jedes neue Formular in die Liste der automatisch zu erzeugenden Formulare aufgenommen. Sie können dieses Standardverhalten jedoch auch umkehren, indem Sie die Option FORMULARE AUTOM. ERZEUGEN in den Umgebungsoptionen, Seite PRÄFERENZEN, deaktivieren.

  Für Formulare, die nicht automatisch erstellt werden, müssen Sie den Konstruktor des Formulars selbst aufrufen, bevor Sie ein Exemplar davon benutzen können. Detaillierte Informationen dazu finden Sie in Kapitel 3.4.4 (*Arbeiten mit mehreren Formularen*), ein häufiges Beispiel dafür sind dynamisch erzeugte MDI-Kindfenster (siehe Kapitel 6.7.4).

Hinweis: Damit manuelle Änderungen im Quelltext von Projektdateien in den Projektmanager übernommen werden, müssen Sie das Projekt neu laden.

Befehle für das Projektmanagement

Die mit der Projektverwaltung zusammenhängenden Befehle sind an verschiedenen Orten anzutreffen: im Hauptmenü (DATEI, ANSICHT und PROJEKT), in der Symbolleiste unter dem Hauptmenü, in der Symbolleiste des Projektfensters und im lokalen Menü desselben.

Der Inhalt des lokalen Projektmanager-Menüs hängt davon ab, welche Art von Knoten Sie anklicken. Bei einer Zieldatei (einem »Einzelprojekt«) finden Sie darin z.B. Befehle zum Hinzufügen oder zum Entfernen von Dateien, zum Speichern aller aufgelisteten Dateien und zum Editieren der Projektoptionen, wie etwa zum Festlegen des Icons für die ausführbare Datei.

Projekte erzeugen

Das vollständige Neu-Kompilieren des Projekts inklusive all seiner Units, für das es im Englischen den schönen Fachbegriff *Build* gibt, wird in der deutschsprachigen Version von Kylix als ERZEUGEN bezeichnet, was Sie nicht dazu verleiten sollte, dies mit dem nicht eingedeutschten Menüpunkt für das COMPILIEREN zu verwechseln. Im normalen Entwicklungsalltag wird das Erzeugen eines Projekts meistens dann fällig, wenn Sie etwa eine globale Compiler-Option geändert haben und nun den gesamten Quelltext mit der neuen Option übersetzen wollen. Bei solchen Optionsänderungen führt Kylix nämlich vor dem nächsten Programmstart nicht von sich aus eine automatische Neu-Kompilierung durch.

Es empfiehlt sich, ein Projekt außerdem neu zu erzeugen, bevor Sie die kompilierte Datei weitergeben oder wenn es bei der Programmausführung zu unerklärlichen Fehlern kommen sollte. Beim normalen Kompilieren übersetzt Kylix nämlich nur die Quelltexte, die seit der letzten Übersetzung aktualisiert wurden, sowie die Dateien, die von diesen abhängig sind. Fehler bei solchen Abhängigkeitsprüfungen (die von der aktuellen Kylix-Version zwar noch nicht berichtet wurden, aber nie ganz auszuschließen sind) könnten dazu führen, dass eine veraltete Unit in die neue ausführbare Datei eingebunden wird. Und selbst wenn diese veraltete Unit an sich keine Fehler enthält, können zur Laufzeit die erwähnten rätselhaften Fehler entstehen, wenn die anderen Units mit einer neuen Version der veralteten Unit kompiliert wurden und daher z.B. ein falsches Aufrufformat für Methoden verwenden (dieser Fehler kann dann natürlich auch bei Aufrufen von neuen Units aus der veralteten Unit vorkommen).

Automatisch angezeigte Formulare und das Hauptformular

Zwar werden alle automatisch erzeugten Formulare Ihrer Anwendung beim Programmstart erzeugt, allerdings ist es sinnvoll, dass davon zunächst nur eines sichtbar wird. Sichtbar sind zunächst nur die Formulare, deren *Visible*-Property den Wert *True* aufweist. Formulare, die am Anfang nicht sichtbar sind, können Sie zur Laufzeit des Programms bei geeigneter Gelegenheit von der Unsichtbarkeit befreien, indem Sie *sible* auf *True* setzen.

Schließlich ist es noch von Bedeutung, welches Formular das *Hauptformular* einer Anwendung ist. Wenn Sie nämlich dieses Formular zur Laufzeit schließen, wird die gesamte Anwendung geschlossen. Andere Formulare können Sie jederzeit schließen, ohne dadurch die Anwendung zu beenden.

Standardmäßig ist das erste Formular des Projekts das Hauptformular. Um das später zu ändern, wählen Sie ein anderes Formular aus der Liste *Hauptformular* in den Projektoptionen (Seite *Formulare*) aus. Hinter dieser Dialog-Kulisse verbirgt sich der folgende Mechanismus, den Sie im Projekt-Quelltext sehen können, sobald Sie die Dialogbox wieder schließen:

Die Methode *Application.CreateForm* erkennt es, wenn zum ersten Mal ein Formular erzeugt wird. Dieses erste Formular sieht sie unbeirrbar als Hauptformular an. Später önnen Sie dieses über das Property *Application.MainForm* abfragen, aber nicht mehr ndern. Um das Hauptformular zum Hauptformular zu machen, muss dieses also im ersten Aufruf von *CreateForm* der Projektdatei erzeugt werden. Genau das ist es, was Sie auch mit der Auswahl eines neuen Hauptformulars in der Dialogbox *Projektoptionen* erreichen.

Projektgruppen

Wenn Sie wissen, was ein Projekt ist, können Sie sich leicht in die Projektgruppen einarbeiten. In einer Projektgruppe werden mehrere Einzelprojekte zusammengefasst, das Hinzufügen von Projekten geht denkbar einfach über die entsprechenden Menüpunkte im Hauptmenü unter PROJEKT und im lokalen Menü der Projektgruppe im Projektfenster. Die Reihenfolge der Projekte lässt sich über FRÜHER ERSTELLEN und SPÄTER ERSTELLEN nachträglich ändern.

Innerhalb einer Gruppe können Sie jedes Projekt weiter wie ein einzelnes Projekt behandeln, jedes Projekt verfügt also über einen eigenen Satz an Optionen, die Sie im bekannten Optionsdialog einstellen können. Diesen Dialog können Sie aus dem lokalen Menü des Projekts im Projektmanager aufrufen oder aus dem Hauptmenü, falls das Projekt das *aktive* Projekt ist.

Das Konzept des *aktiven Projekts* ist auch für andere Aufgaben wie etwa das Kompilieen und Starten einer Anwendung von Bedeutung: Aus allen Projekten einer Gruppe

kann immer nur ein Projekt aktiv sein. Welches das gerade ist, können Sie aus dem Namen in der Kylix-Titelleiste, aus der Hervorhebung im Projektfenster und auch aus der Auswahlliste am oberen Ende des Projektfensters entnehmen. Über diese Auswahlliste, über das lokale Menü des Projektmanagers und über dessen Symbolleiste können Sie ein Projekt manuell aktivieren – eine automatische Aktivierung findet statt, wenn Sie aus dem Projektmanager heraus zu einer Datei des Projekts wechseln.

Die meisten Befehle im Menü PROJEKT beziehen sich auf dieses aktive Projekt, ebenso wie der Menüpunkt START | START. Im Projektmenü können Sie schon durch die Namen der Menüpunkte erkennen, ob sie sich auf die ganze Gruppe oder nur auf das aktive Projekt beziehen. So gibt es sowohl den Punkt *[Name des Projekts]* ERZEUGEN als auch ALLE PROJEKTE ERZEUGEN.

Einzelprojekte und »virtuelle Gruppen«

Schon die Tatsache, dass Sie im DATEI | NEU...-Dialog keinen Eintrag für »Neues Projekt«, sondern nur noch für »Neue Projektgruppe« finden, weist darauf hin, dass Sie eigentlich gar nicht ohne Projektgruppen arbeiten können. Auch wenn Sie eine einzelne DPR-Datei öffnen, wird dieses Projekt im Projektmanager immer innerhalb einer neuen Gruppe dargestellt. Solange Sie diese Gruppe nicht speichern, bleibt sie aber »virtuell«, sie existiert also nicht als physikalische Datei auf der Festplatte. *Wenn* Sie eine Gruppe speichern (und das wird spätestens dann erforderlich, wenn Sie mehr als ein Projekt darin aufgenommen haben), erzeugt Kylix dafür eine Datei mit der Endung `.bpg` (Borland Project Group) – eine lesbare Datei, die nach einer einfachen Syntax aufgebaut ist, welche mit Object Pascal keinerlei Ähnlichkeiten aufweist und für die eine genauere Beschreibung in diesem Buch sicher uninteressant wäre. Mit QUELLTEXT DER PROJEKTGRUPPE ANZEIGEN aus dem lokalen Menü des Projektmanagers (nur bei Markierung des Wurzelknotens) können Sie diese Datei im Editor einsehen, unabhängig davon, ob sie auf der Festplatte existiert oder nicht.

1.6.4 Objektablage und Komponentenschablonen

Das Prinzip der Wiederverwendung aus der objektorientierten Programmierung ist in Kylix auch bei der Formulargestaltung anzutreffen. Visuell gestaltete »Bauelemente« können auf die folgenden Arten wiederverwendet werden:

- Wiederverwendung eines einmal angelegten Formulars/Projekts, das Sie in der Objektablage untergebracht haben
- Erweiterung eines einmal angelegten Formulars (hier *erbt* ein neues Formular den Aufbau und die Ereignisbearbeitung von einem in der Objektablage gespeicherten Formular)

Wiederverwendung von Komponenten oder Komponentengruppen als Komponentenschablone (Komponentenvorlage).

- Zusammenfassung von Komponentengruppen in einem *Frame*, der so leicht wiederverwendet werden kann wie eine Komponentenschablone, aber gegenüber dieser erhebliche Vorteile aufweist.

Die folgenden Abschnitte gehen auf den Einsatz der ersten drei Techniken ein. Eine Diskussion der Vor- und Nachteile der einzelnen Techniken sowie Beispiele für die Anwendung von Formularvererbung und Frames finden Sie in Kapitel 3.7.

Zugriff auf die Objektablage

Der übliche Zugriff auf die Objektablage läuft über das Menü DATEI | NEU ab. Sie gelangen dadurch in die in Abbildung 1.11 gezeigte Dialogbox, wählen eines der auf den Seiten enthaltenen Objekte aus, stellen die Art der Nutzung ein (KOPIEREN, VERERBEN oder VERWENDEN, dazu später mehr), wählen *OK* und erhalten ein neues Objekt, das von nun an zu Ihrem Projekt zählt.

Sie können die IDE jedoch auch so einstellen, dass sie bei Auswahl der Menüpunkte ATEI | NEUE ANWENDUNG und DATEI | NEUES FORMULAR kein leeres Projekt bzw. Formular erstellt, sondern eine Kopie eines Projekts/Formulars aus der Objektablage. Hierzu markieren Sie das Projekt/Formular in den Optionen zur Projektablage (TOOLS | OBJEKTABLAGE...) als NEUES FORMULAR bzw. NEUES PROJEKT.

ine dritte Option an dieser Stelle ist HAUPTFORMULAR. Ein damit markiertes Formular der Objektablage wird beim Öffnen eines neuen Projekts automatisch als Hauptformular verwendet.

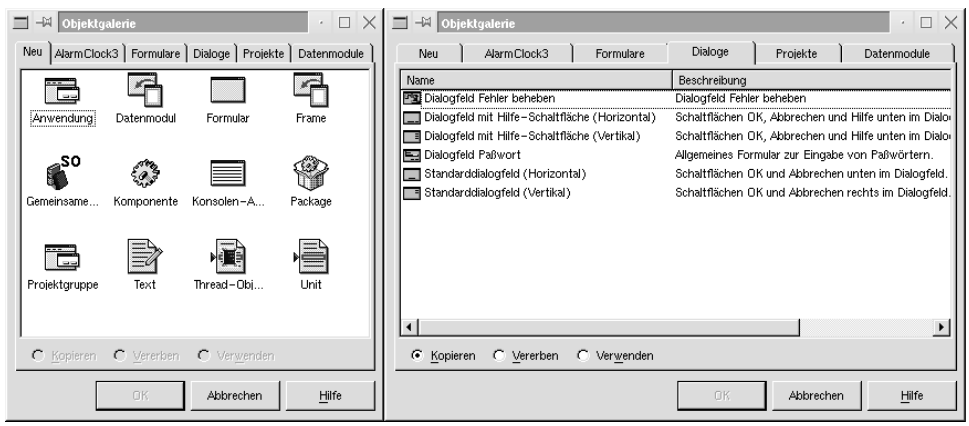


Abbildung 1.11: Die Objektablage befindet sich im Neu-Dialog; rechts die mit Kylix mitgelieferten Dialog-Schablonen

Die drei Arten, ein abgelegtes Objekt zu verwenden

Die Objektablage bietet Ihnen grundsätzlich die drei oben erwähnten Möglichkeiten, die in ihr aufbewahrten Objekte in neuen Projekten wiederzuverwenden, auch wenn nicht für jedes Objekt alle drei Optionen zur Verfügung stehen:

- Wenn Sie KOPIEREN wählen, lädt Kylix alle Dateien, die zum ausgewählten Objekt gehören, und legt sie als unbenannte Dateien im aktuellen Projekt ab. Handelt es sich beispielsweise um ein Formular, so lädt Kylix die Formulardatei und die zugehörige Unit. Beim nächsten Speichern werden Sie dann automatisch nach einem Namen für die noch unbenannten Dateien gefragt. Kylix speichert die Dateien unter dem neuen Namen ab. Diese neuen Dateien haben nichts mehr mit den Objekten der Objektablage zu tun.

- Die beim Erscheinen von Delphi 2 zu Recht viel gepriesene »visuelle Formular-Vererbung« erhalten Sie mit der Option VERERBEN. Wie beim Kopieren des Objekts legt Kylix beim Vererben eine oder mehrere neue unbenannte Dateien an, jedoch werden diese nicht aus den Dateien der Objektablage erzeugt.

  Am Beispiel eines Formulars bedeutet dies: Die neu erzeugte Formular-Unit enthält eine neue Formularklasse, die keinerlei Komponentendeklarationen und keine Ereignisbearbeitungsmethoden enthält, so als handele es sich um ein völlig neues Formular. Einziger Unterschied zu einem solchen völlig neuen Formular ist, dass die Formularklasse nicht von *TForm*, sondern von der Klasse des Formulars der Objektablage abgeleitet ist. Die Unit enthält also statt »*TForm1=class(TForm)*« eine Zeile der Art »*TForm1=class(TFormXYZAusAblage)*«.

  Für Kylix ist es mit diesem kleinen Unterschied natürlich nicht getan; es muss im Hintergrund einigen Verwaltungsaufwand betreiben, um schon zur Entwurfszeit ein Formular darzustellen, das sowohl die Komponenten des Ablageobjekts als auch die Komponenten, die Sie nachträglich hinzufügen, anzeigt. (Dieser hohe Verwaltungsaufwand, der durch einen langsameren Bildaufbau zur Entwurfszeit auffallen kann, ist zur Laufzeit nicht erforderlich. Die Vererbung beeinträchtigt also die Geschwindigkeit Ihrer Anwendung nicht.)

  Jede Ihrer Änderungen des per Vererbung neu erzeugten Formulars wirkt sich nur auf dieses Formular aus. Umgekehrt wirken sich jedoch alle Änderungen, die Sie an einem Formular der Objektablage vornehmen – sowohl Änderungen der Komponenten und Properties als auch Änderungen der Ereignisbearbeitungsmethoden – auf alle Formulare aus, die Sie über die Option *Vererben* von diesem Formular erzeugt haben.

  Außer Formularen (dazu gehören auch die Formulare auf der Objektablage-Seite *Dialoge*) können Sie noch Datenmodule vererben, bei Projekten scheidet diese Möglichkeit aus, da ein Projekt nicht als Klasse vorliegt.

- Schließlich können Sie manche Objekte der Ablage auch noch VERWENDEN. Diese Option bewirkt, dass Kylix keine neue unbenannte Datei anlegt, sondern die Originaldateien aus der Objektablage direkt in Ihr Projekt einbindet. Wenn Sie also Änderungen durchführen, verändern Sie das Objekt, das sich in der Objektablage befindet. Eine solche Veränderung ist besonders dann erwünscht, wenn es sich um eine Formularklasse handelt, die an andere Formulare vererbt wird, und wenn alle diese Formulare gleichzeitig verändert werden sollen. Ändern Sie dann das Formular aus der Objektablage, so wirkt sich das auch auf diese Formulare aus. (Allerdings müssen Sie ein Ablageobjekt nicht über VERWENDEN in das Projekt einbinden, um es direkt zu verändern. Sie können es auch mit DATEI | ÖFFNEN direkt aus der Ablage laden – Kylix speichert die Ablageobjekte im Verzeichnis `<kylix>/objrepos`, für die Aufteilung auf die verschiedenen Seiten der Ablage ist die Datei `<home-Verzeichnis>/.borland/delphi6Odro` zuständig.)

Die Optionen VERERBEN und VERWENDEN stehen übrigens nur zur Verfügung, wenn Sie den Menüpunkt DATEI | NEU verwenden, nicht aber bei der automatischen Auswahl eines Ablageobjekts durch DATEI | NEUE ANWENDUNG und DATEI | NEUES FORMULAR.

Inhalt der Objektablage

Um einen Einblick in den Inhalt der Objektablage zu gewinnen, wählen Sie den Menüpunkt DATEI | NEU. Sie erhalten den in Abbildung 1.11 gezeigten Dialog, der mehrere Seiten mit Icons enthält. Die Icons werden von einem *ListView*-Steuerelement dargestellt, das Sie auch auf andere Darstellungsmodi umstellen können (hier zu finden im lokalen Menü des *ListViews*).

Die Seite NEU enthält Vorlagen für verschiedene neue Objekte wie Units, Formulare, Anwendungen, Threads. Die drei oben beschriebenen Nutzungsoptionen entfallen für diese Objekte. Die zweite Seite ist dem gerade geladenen Projekt gewidmet (in Abbildung 1.11 dem TreeDesigner-Projekt) und listet all seine Formulare auf. Mit Hilfe dieser Seite können auch Formulare innerhalb Ihres Projekts voneinander erben, ohne dass Sie das Formular, von dem geerbt werden soll, zuerst in die Objektablage einfügen müssen.

Die weiteren Seiten stellen den eigentlichen Inhalt der Objektablage dar. Sie enthalten Formulare und Projekte, denen Sie weitere hinzufügen können. Auch so genannte Experten und Wizards zum Anlegen von Formularen und Projekten werden auf diesen Seiten aufgeführt.

Anpassen der Ablage und Hinzufügen von Objekten

Es stellt sich jetzt noch die Frage, wie Sie eigene Objekte in die Objektablage einfügen. Hier gilt es wieder, nach Projekten und Formularen zu unterscheiden:

- Für Formulare finden Sie in deren lokalem Menü den Punkt DER OBJEKTABLAGE HINZUFÜGEN..., der Sie in eine Dialogbox führt, in der Sie unter anderem ein Formular des aktuellen Projekts, eine Beschreibung, ein Icon und die Seite der Objektablage, auf der das Formular erscheinen soll, auswählen können.
- Mit demselben Dialog können Sie auch das Projekt in die Ablage einfügen, wenn Sie den Menüpunkt PROJEKT | DER OBJEKTABLAGE HINZUFÜGEN auswählen.

Rein organisatorischer Natur sind die Einstellmöglichkeiten, die Ihnen der Optionsdialog zur Objektablage bietet (TOOLS | OBJEKTABLAGE...). Neben den Optionen, die bereits im letzten Abschnitt erwähnt wurden, können Sie hier auch Objekte zwischen den einzelnen Seiten verlagern, neue Seiten erstellen und Objekte aus der Ablage entfernen (ähnlich der Anpassung der Komponentenpalette mit TOOLS | UMGEBUNGSOPTIONEN | PALETTE).

Wenn Sie neue Objekte in die Ablage einfügen, die Sie später in der Ablage modifizieren, ist es natürlich auch eine gute Idee, das gesamte Verzeichnis der Objektablage in etwaige Backup-Operationen Ihres Projekts einzubeziehen.

Komponentenschablonen

Wenn Sie viele verschiedene Formulare entwerfen, in denen manche speziell eingestellte Komponenten und Komponentengruppen immer wieder vorkommen, werden Sie wahrscheinlich die Komponentenschablonen schätzen lernen (z.B. für eine Schablone für das Duo aus *Ok* und *Abbruch*-Schalter, bei denen Sie sonst regelmäßig sechs verschiedene Property-Einstellungen im Objektinspektor vornehmen müssten, siehe hierzu Kapitel 1.8.1). Je nachdem für welchen Anwenderkreis Ihr Formular gedacht ist, wollen Sie den *Abbruch*-Schalter vielleicht auch immer mit einer Sicherheitsabfrage verknüpfen. Auch die hierzu notwendige Methode ließe sich in die Schablone aufnehmen.

Um eine einmal entworfene Komponente oder Komponentengruppe zu einer neuen Komponente in der Komponentenpalette zu machen, markieren Sie alle notwendigen Komponenten und führen den Menüpunkt KOMPONENTE | KOMPONENTENVORLAGE ERZEUGEN... aus. In der daraufhin erscheinenden Dialogbox geben Sie der Schablone einen Namen, bestätigen das vorgegebene Icon oder wählen ein anderes und geben an, auf welcher Seite der Komponentenpalette die Schablone angezeigt werden soll.

Hinweis: Intern handelt es sich beim neuen Symbol in der Komponentenpalette nicht um eine *echte* Komponente. »Echte« Komponenten haben immer eine zugehörige Klasse, die Sie bei eigenen Komponenten auch selbst entwickeln müssen (zur Erweiterung oder Veränderung bestehender Komponenten siehe auch Kapitel 8.5.1).

Eine so in die Komponentenpalette eingefügte Schablone können Sie wie eine normale Komponente in jedes Formular einfügen. Kylix stellt dabei nicht nur alle Property-Werte der Komponenten wieder her, sondern übernimmt auch die Ereignisbearbeitungsmethoden, die die Komponenten im Original-Formular hatten, in das neue Formular. Nur die Namen der Komponenten (und folglich auch die Namen der Methoden) ändern sich bei dieser Einfügen-Operation (Kylix verwendet hier wieder die automatisch erzeugten Vorgabenamen wie *Button1* usw.).

Speicherung der Komponentenschablonen

ür den Fall, dass Sie auch die Komponentenschablonen in Ihr Backup einbeziehen wollen (und Sie nicht immer die gesamte Festplatte sichern): Die Komponentenschablonen werden nicht im Verzeichnis der Objektablage, sondern in der Datei `delphi.dct` im Ressourcenverzeichnis `<home>/.borland` gespeichert.

1.6.5 Der Browser

Nachdem wir ein Projekt im letzten Kapitel aus der Sicht seiner einzelnen Dateien untersucht haben, kommen wir nun zu einer anderen Sichtweise des Projekts, wie sie im Browser dargestellt wird (Abbildung 1.12).

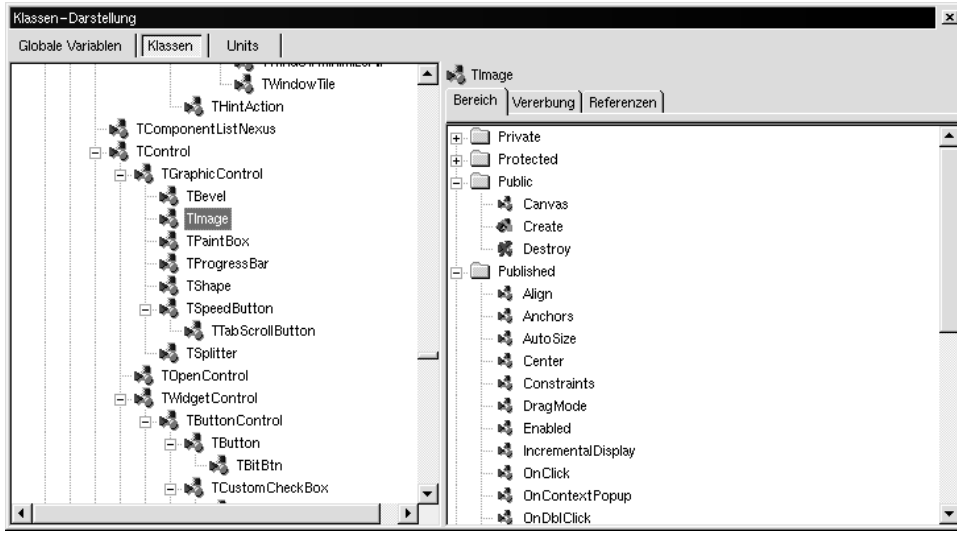


Abbildung 1.12: Der Browser

Der Browser liefert Ihnen Informationen über die Klassenhierarchie Ihres Projekts und der CLX und erlaubt es Ihnen, gezielt nach Klassen, deren Elementen und beliebigen anderen Programmsymbolen zu suchen. Dabei ist der Browser grundsätzlich zuverläs-

siger als die Hilfedatei, da er mit den aktuellsten, vom Compiler bereitgestellten Daten arbeitet, während die Hilfedatei schon nach einer kleinen nachträglichen Änderung der CLX durch Borland nicht mehr auf dem neuesten Stand wäre.

Sie können den Browser auf zwei Arten aufrufen:

- Mit ANSICHT | BROWSER erhalten Sie ein Fenster, in dem alle Symbole einer bestimmten Art und eines bestimmten Bereichs dargestellt werden.
- SUCHEN | SYMBOL ANZEIGEN... fragt Sie zunächst nach dem Namen eines Symbols. Dieses wird daraufhin in einem Browserfenster angezeigt, falls es gefunden werden konnte. Sie können nur aus der globalen Perspektive suchen. Um nicht-globale Symbole finden zu können, müssen Sie vorher das globale Symbol angeben, in dem das gesuchte Symbol enthalten ist; so finden Sie z.B. mit der Eingabe *TForm.Caption* das *Caption*-Property der Formularklasse.

Die Eigenschaften des Browsers befinden sich zusammen mit denen des CodeExplorers auf der schon in Kapitel 1.6.1 beschriebenen Dialogseite (Abbildung 1.8). Zwei exklusiv für den Browser geltende Einstellungen wurden jedoch in Kapitel 1.6.1 noch nicht erläutert:

- die Anfangsansicht, die Sie nach dem Start des Browsers durch ANSICHT | BROWSER erhalten und für die KLASSEN, UNITS und GLOBAL zur Auswahl stehen,
- der Bereich der Symbole, der entweder nur das aktuelle Projekt oder das Projekt mit allen Symbolen der Kylix umfassen kann.

Ansichtsarten und Browserbereich

Die Ansichten, die Sie nachträglich auch über die drei Schalter unterhalb der Titelzeile des Browsers ändern können, haben folgende Bedeutung:

- Die KLASSEN-Ansicht zeigt einen Hierarchiebaum der im gewählten Browserbereich definierten Klassen an. Dieser beginnt in jedem Fall bei der allen Klassen zugrunde liegenden Basisklasse *TObject*. Falls der Browserbereich auch Interfaces umfasst, gibt es neben *TObject* noch eine zweite Wurzel namens *IUnknown*.
- UNITS zeigt eine Liste aller im Programm überhaupt vorkommenden Units an, also auch die Units, die Sie nicht direkt eingebunden haben, sondern die von irgendeiner anderen Unit per *uses*-Klausel geladen wurden. Sollte es sich dabei um Units handeln, die nicht in der Online-Hilfe erwähnt sind, dann sind dies Units, die von der CLX nur intern benötigt werden.
- Mit GLOBALE VARIABLEN erhalten Sie eine sehr lange Liste aller Bezeichner, die global, also weder innerhalb einer Klasse noch innerhalb einer Prozedur/Funktion/Methode definiert sind. Hierzu gehören sämtliche Funktionen der Laufzeitbibliothek, alle Konstanten wie z.B. die Farbkonstanten *cl...* usw. Übersichtlicher ist es,

wenn Sie zuerst die Liste der Units anzeigen lassen und dann eine Unit auswählen. Sie erhalten dann im rechten Teil des Browsers eine Liste aller in dieser Unit definierten globalen Symbole.

Der Browserbereich ist normalerweise auf alle Symbole der CLX eingestellt, Sie können ihn im Eigenschaftsdialog aber auch auf die Symbole des aktuellen Projekts einschränken. Wenn Sie beispielsweise gerade ein neues Projekt mit nur einer Formular-Unit angelegt haben, so wird Ihnen der Browser bei dieser Einschränkung als einzige ariable die automatisch definierte Variable *Form1* zeigen und in der Klassenansicht wird es nur die Klasse *TForm1* geben sowie die Klassen, von denen *TForm1* abstammt.

Hinweis: Unter bestimmten Umständen – z. B. wenn der Quelltext aufgrund eines Fehlers nicht im Hintergrund übersetzt werden konnte – kennt der Browser nur die Projektsymbole, die beim letzten manuellen Kompilieren des Projekts definiert waren.

Bedienung

Die Bedienung eines typischen Browser-Fensters wie in Abbildung 1.12 ist intuitiv echt leicht erfassbar. Der rechte Fensterteil zeigt jeweils die Details zu dem im linken Bereich gewählten Symbol. Mit einem Doppelklick erhalten Sie zu den in beiden Bereichen aufgeführten Elementen weitere Informationen:

Für die auf der Seite REFERENZEN angegebenen Quelltextpositionen, an denen ein bestimmtes Symbol verwendet wird, springen Sie direkt an die entsprechende Position im Quelltext-Editor.

- Bei allen anderen Einträgen erhalten Sie ein neues, kleines Browserfenster ohne Detailansicht. Sofern es sich bei dem Eintrag um eine Variable oder Methode handelt, enthält dieses Fenster lediglich die Seite REFERENZEN. Falls Sie auf eine Klasse doppelgeklickt haben, finden Sie alle drei möglichen Seiten BEREICH, VERERBUNG und REFERENZEN.

1.6.6 To-Do-Listen

In To-Do-Listen können Sie Notizen über alle erdenklichen Arten von offenen Arbeiten an Ihrem Projekt speichern. Aus dem in Abbildung 1.13 gezeigten Fenster (ANSICHT | O-DO-LISTE) einer solchen Liste geht bereits die Struktur eines To-Do-Eintrags hervor: Neben dem eigentlichen Text können Sie jeden Eintrag mit einer Priorität (1 = hoch bis 5 = niedrig oder 0 = ohne Priorität), dem Namen einer zuständigen Person sowie mit einer Kategorie versehen. Optional kann er noch mit einem bestimmten Modul Ihres Projekts verbunden werden. Die Kategorien sind nicht vorgegeben, sondern als freier

Text einzugeben oder aus einer aufklappbaren Liste auszuwählen, die alle bereits in der To-Do-Liste vorkommenden Kategorien zeigt. Ähnlich funktionieren Eingabe und Auswahl des Namens einer zuständigen Person.

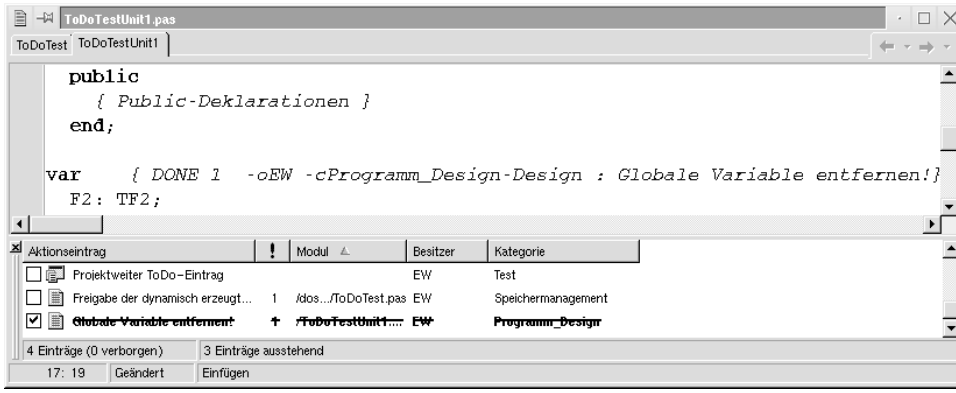


Abbildung 1.13: To-Do-Einträge in Editor und angedockter To-Do-Liste

Die Handhabung des To-Do-Fensters – Einfügen neuer Einträge, Sortieren der Tabelle nach verschiedenen Spalten, Markieren von Einträgen als erledigt (»fertig«), Beschränken der Anzeige auf bestimmte Arten von Einträgen (»filtern«), Bearbeiten und Löschen bestehender Einträge – ist über die Bedienelemente des Fensters selbst (Spaltenüberschriften und Checkbox vor jedem Listeneintrag) sowie über das Popup-Menü des ToDo-Fensters so einfach, dass sie nicht weiter beschrieben werden muss. Erwähnenswert ist noch die Möglichkeit, eine To-Do-Liste als HTML-Tabelle zu exportieren – mit zahlreichen benutzerdefinierbaren Optionen z. B. für Farbe und Tabellenaufbau.

Auch die To-Do-Verwaltung ist ein Two-Way-Tool

Wenn Sie Ihre To-Do-Listen bisher im Quelltext verwaltet haben und sich nun fragen, warum Sie auf eine externe Liste ausweichen sollen, sollten Sie sich genauer ansehen, wie die To-Do-Liste mit dem Quelltext-Editor zusammenarbeitet.

Zunächst einmal haben Sie die Möglichkeit, aus dem Quelltext-Editor heraus neue To-Do-Einträge zu erzeugen (das zugehörige Tastenkürzel ⇧+Strg+T finden Sie auch im Popup-Menü des Editors). In der To-Do-Liste erscheint dann ein neuer Eintrag, in dessen MODUL-Spalte der Name der Unit angezeigt wird, aus der heraus Sie den Eintrag hinzugefügt haben. Die Unit selbst erhält an der aktuellen Eingabeposition einen neuen Kommentar, der den Inhalt des Eintrages mit Priorität, Personenname und Kategorie widerspiegelt, z. B.:

```
{ TODO 2 -oEW -cPhantasieKategorie : Hier das muss noch geändert werden! }
```

Die »Two-Way-Tool-Eigenschaft« ist gewährleistet, da Sie diesen Eintrag sowohl im To-Do-Fenster als auch im Quelltext beliebig ändern können. Sie können sogar von Hand neue To-Do-Einträge in den Quelltext einfügen. Sobald Sie einen Kommentar mit einem *TODO* beginnen wie im obigen Beispiel, fügt Kylix einen neuen Eintrag in die Liste ein. Entfernen Sie den Kommentar, wird er aus der Liste gelöscht.

Dies bedeutet, dass Sie »maschinell unterstützte« To-Do-Einträge genauso flexibel im Quelltext unterbringen können wie früher von Hand verwaltete Einträge. Und Sie önnen sie auch dann lesen, verwenden und editieren, wenn die IDE aus irgendeinem Grund einmal nicht zur Verfügung stehen sollte. Um einen Eintrag beispielsweise in einem externen Editor als erledigt zu markierern, tauschen Sie das *TODO* am Anfang des Kommentars in ein *DONE* aus, Kylix erkennt diesen Eintrag dann beim nächsten Laden als »fertig«.

Projektweite To-Do-Einträge

Einträge, die Sie im To-Do-Fenster selbst mit HINZUFÜGEN einfügen, gelten nicht für eine einzelne Datei, sondern projektweit. Sie werden jedoch nicht etwa direkt in der Projektdatei gespeichert, sondern in einer eigenen Datei mit der Endung `.todo`. Diese Datei wiederum wird über einen Kommentar in die Projektdatei eingebunden, z. B.:

```
{%ToDo 'ToDoTestProject.todo'}
```

Auch die `.todo`-Datei ist eine Textdatei; sie enthält die Einträge im selben »Kommentarformat«, in dem sie auch in den Quelltextdateien vermerkt werden.

Hinweis: Die To-Do-Funktionalität ist nicht auf das aktuelle Projekt beschränkt: Wenn Sie in FILTER | EINTRAGSTYPEN (lokales Menü des To-Do-Fensters) die falsch übersetzte Option QUELLDATEIEN ÖFFNEN (gemeint ist »geöffnete Quelldateien«) wählen, zeigt das To-Do-Fenster sogar die To-Do-Einträge von geöffneten Dateien, die nicht zum Projekt gehören.

1.7 Der Debugger

Im Verlauf des bisherigen Beispielprogramms gab es noch keinen Anlass, den Debugger zu Rate zu ziehen, bei größeren Projekten ist er jedoch ein unverzichtbares Werkzeug, um fehlerhafte Programmteile auf frischer Tat zu ertappen. Neben diesem integrierten Debugger befasst sich dieses Kapitel auch mit einem automatischen Hilfsmittel der Fehlersuche, den Assertions.

Hinweis: Für ein bequemes Arbeiten mit den vielen Arten von Debugger-Fenstern empfiehlt es sich, eine Symbolleiste einzurichten, über die die einzelnen Fenster schnell aufgerufen werden können (siehe etwa Abbildung 1.17), und dann die entsprechenden Tastenkürzel zu lernen, die im Hinweisfenster unter der Maus angezeigt werden.

1.7.1 Übersetzungsoptionen für den Debugger

Nach der Installation ist die Kylix-IDE auf die Verwendung des internen Debuggers voreingestellt, so dass Sie alle ab Kapitel 1.7.2 beschriebenen Funktionen sofort ausprobieren können. Dieses Kapitel fasst die für den Debugger wichtigen Optionen zusammen, damit Sie eventuelle Fehleinstellungen schnell beheben können.

Der Debugger benötigt umfangreiche Informationen über die Symbole (Namen von Objekten, Variablen etc.), über den Aufbau der Komponenten und Klassen und über die Position der Quelltextzeilen im übersetzten Maschinencode. Das einzige Tool der Kylix-Umgebung, das sich damit richtig auskennt, ist der Compiler. Um den Debugger verwenden zu können, müssen Sie diesen also anweisen, die Informationen in einem für den Debugger leicht verständlichen Format abzulegen.

Zuerst einmal müssen diese Informationen in den DCU-Dateien abgelegt werden. Um die maximal mögliche Menge von Informationen zu erhalten, wählen Sie unter den Compileroptionen (in den Projektoptionen enthalten auf der Seite COMPILER) die ersten drei Optionen im Bereich *Debugger*, nicht aber das Markierungsfeld NUR DEFINITIONEN (Abbildung 1.14).

Des Öfteren kann es sich auch als nützlich erweisen, wenn Sie zum Debuggen die Optimierung abschalten, denn durch diese können Variablen und Quelltextzeilen wegoptimiert werden, so dass das Debuggen erschwert werden kann. Wenn Sie z.B. eine neue Variable einführen, die nur zum Debuggen dienen soll, der Sie dann einen Wert zuweisen (den Sie im Debugger untersuchen wollen), die Sie aber im Programmcode nicht weiter verwenden, dann optimiert der Compiler diese Variable standardmäßig weg, und Sie bekommen nur ein entschuldigendes Meldungsfenster, wenn Sie die Variable im Debugger untersuchen wollen.

Zum Abschalten der Optimierung haben Sie zwei Möglichkeiten:

- Schalten Sie die Optimierung global in den Projektoptionen ab (Abbildung 1.14),
- oder steuern Sie die Optimierung gezielt über die Angabe von Compilerbefehlen im Quelltext. So können Sie die Optimierung mit *{$Optimization Off}* z.B. vor einer Methode abschalten und mit *{$Optimization Off}* danach wieder anschalten (Sie können auch die Kürzel *{$O-}* und *{$O+}* verwenden).



Abbildung 1.14: Compileroptionen für Debugger- und Browsereinsatz

1.7.2 Allgemeine Debugger-Fenster

Die folgenden Abschnitte behandeln zunächst die Debugger-Fenster, die nicht im Zusammenhang mit Haltepunkten und dem Untersuchen von Variablen stehen: das Modul-Fenster, das CPU-Fenster und das Ereignisprotokoll.

Das Modul-Fenster

Sobald Sie eine Anwendung gestartet haben, steht Ihnen das Modul-Fenster (Abbildung 1.15) über das Menü ANSICHT | DEBUG-FENSTER | MODULE zur Verfügung. Sinn dieses Fensters ist es, alle Module aufzulisten, die an der Ausführung Ihrer Anwendung beteiligt sind. Dazu gehören die ausführbare Datei, alle direkt und indirekt benötigten Bibliotheksdateien sowie die Packages, falls Sie sich für die Nutzung der Package-Technologie entschieden haben (siehe Kapitel 8.1.3). Beim Mehrprozess-Debuggen zeigt es auch mehrere laufende Prozesse an und markiert den aktiven Prozess mit einem grünen Pfeil (jede ausführbare Datei wird zur Laufzeit zu einem *Prozess*).

Die beiden Bereiche am rechten und unteren Rand des Fensters zeigen die Details zu dem Modul, das im Hauptbereich gewählt ist:

- Der untere Bereich listet, sofern das Modul mit Debugger-Informationen übersetzt wurde, die für das Modul verwendeten Quelltextdateien auf. Über das lokale Menü können Sie natürlich gleich zur entsprechenden Quelltextdatei in den Editor springen.
- Der rechte Bereich listet – ebenfalls für das in der Modulliste gerade ausgewählte Modul – alle Einheiten des Programmcodes auf, die im Debugger verfügbar sind. Wenn Debugger-Informationen für das Modul vorliegen, sind dies alle Prozeduren, Funktionen, Methoden, die *initialization*- und *finalization*-Sektionen von Units sowie der Initialisierungs-Programmcode des Projekts (Hauptprogramm der Projektdatei, unter dem Namen des Projekts am Ende der Liste zu finden). Wie zu erwarten, können Sie über das lokale Menü direkt in den Editor zur entsprechenden Stelle in der Quelltextdatei springen.

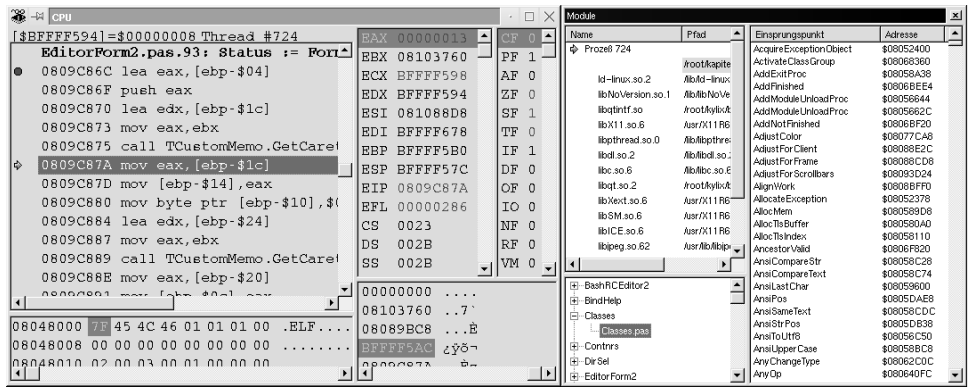


Abbildung 1.15: Zwei systemnahe Fenster des Kylix-Debuggers: CPU- und Modul-Fenster

Eine interessante Funktion des Modulfensters ist, dass Sie sich den Code *aller* Funktionen ansehen können, die in der Liste der Funktionen aufgelistet sind. Wenn es sich um eine Funktion handelt, die nicht im Quelltext vorliegt, wird die Funktion einfach in disassemblierter Form im CPU-Fenster dargestellt.

CPU- und FPU-Fenster

Im CPU-Fenster können Sie die Abarbeitung des Maschinencodes einer Anwendung »live« verfolgen, bei Verwendung von Fließkommazahlen assistiert dabei noch das FPU-Fenster. Während das CPU-Fenster (Abbildung 1.15) den Maschinen- und Assemblercode des Programms zusammen mit den zugehörigen Zeilen des Quelltextes darstellt – sofern dieser vorliegt –, stellt das FPU-Fenster die Register der Fließkomma- bzw. MMX-Einheit des Prozessors dar. Zu den Einsatzgebieten dieser Fenster gehören:

Schon alleine die Anzeige des Assemblercodes hat einen wichtigen informativen Charakter, denn wenn Sie über gute Assemblerkenntnisse verfügen, können Sie sich selbst ein Bild von der Arbeit des Compilers machen und davon, wie effektiv er Ihr Programm in Maschinencode übersetzt hat.

Ähnlich wie beim Debuggen von Quelltext können Sie hier (Assembler-)Anweisungen einzeln ausführen, Haltepunkte setzen und die angezeigten Werte (hier also Register- und Speicherinhalte) testweise ändern.

In die Abarbeitungsreihenfolge des Programms eingreifen: Über den lokalen Menüpunkt NEUER EIP können Sie die aktuelle Programmposition verändern. Stellen Sie sich beispielsweise vor, Sie stehen nach einer langen und komplizierten Debugger-Sitzung kurz vor einer wichtigen Entdeckung, gelangen dann aber an eine an sich unwichtige Anweisung, die Ihr Programm mit Sicherheit in den Absturz führen wird. Statt sich in dieses Schicksal zu fügen, die Absturz-Anweisung aus dem Programm zu entfernen, es neu zu übersetzen und die Debugger-Sitzung von vorne zu beginnen, setzen Sie die Ausführungsposition einfach hinter diese gefährliche Anweisung und lassen das Programm ohne Absturz weiterlaufen.

- Im Datenbereich des CPU-Fensters (links unten) können Sie den Hauptspeicher Ihrer Anwendung in einem maschinennahen Anzeigeformat untersuchen und ändern. Mit dem lokalen Menü können Sie zwischen verschiedenen Formaten wählen und eine Variable angeben, deren Speicherbereich Sie sehen wollen. Auch per Drag&Drop können Sie einen Variablennamen oder Ausdruck aus dem Editorfenster in den Datenbereich des CPU-Fenster ziehen.

Das Ereignisprotokoll

Im Ereignis-Fenster (Abbildung 1.16) können Sie während des Programmlaufs Meldungen über vier verschiedene Arten von Ereignissen sammeln, wenn Sie die entsprechenden Debuggeroptionen einstellen (TOOLS | DEBUGGER-OPTIONEN, Seite *Ereignisprotokoll*). Im Gegensatz zu Meldungsfenstern, bei denen der Benutzer OK drücken muss, ühren diese Meldungen nicht zu einer Unterbrechung des Programmablaufs. Auf diese Weise können in kurzer Zeit hunderte solcher Meldungen gesammelt werden, wobei es dann an Ihnen liegt, sich aus einer langen Meldungsliste die entscheidenden Meldungen herauszusuchen. Die fünf Arten von Meldungen sind:

PROZESS: Diese Option erzeugt zwei unspektakuläre Meldungen beim Starten und Beenden des untersuchten Programms.

THREAD: Hier geht es um die vom Programm gestarteten Threads, wieder werden vor allem Start und Ende protokolliert.

- MODUL: Durch diese Option erhalten Sie Informationen über alle beim Programmstart dynamisch hinzugeladenen Bibliotheken und darüber, ob diese Debug-Informationen enthalten.
- HALTEPUNKT: Wenn das Programm vom Debugger angehalten wird, notiert Kylix die genaueren Umstände dieser Unterbrechung ebenfalls im Protokoll. Als Unterbrechungsursache kommen nicht nur Haltepunkte, sondern auch Ausnahmebedingungen (Exceptions) innerhalb Ihres Programms in Frage.
- Schließlich können Sie über das lokale Menü des Fensters zusätzliche Kommentare in das Protokoll aufnehmen lassen.

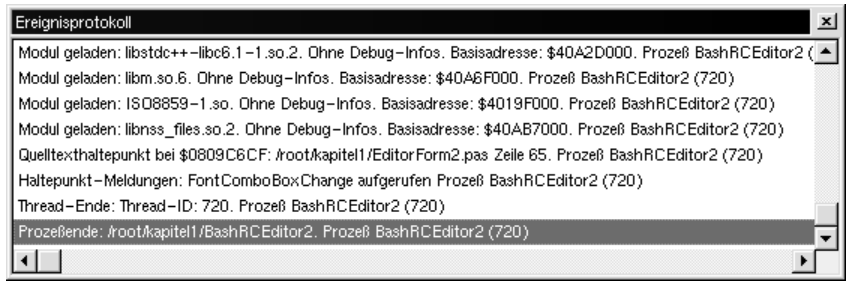


Abbildung 1.16: Das Ereignisprotokoll hält Meldungen über verschiedene Geschehnisse während des Programmablaufs chronologisch fest.

Per Voreinstellung (BEIM START LÖSCHEN in den Debugger-Optionen) wird das Ereignisprotokoll bei jedem Neustart des Projekts geleert. Als sehr nützlich kann sich auch die Möglichkeit erweisen, den gesamten Inhalt in einer Datei zu speichern (lokales Menü) oder einen Teil des Protokolls zu markieren und in die Zwischenablage zu kopieren, denn nur so ist es beispielsweise möglich, nach einem bestimmten Text zu suchen.

Hinweis: Gegenüber dem Ereignisprotokoll von Delphi für Windows fehlen unter Kylix zwei Arten von Ereignissen. Zumindest eine davon, die AUSGABEMELDUNG, die unter Windows mit *OutputDebugString* getätigt werden kann, ist unter Linux leicht zu ersetzen, denn dort können Sie mit *writeln* ebenfalls »Ausgabemeldungen« vornehmen, die dann allerdings in dem Konsolenfenster erscheinen, von dem die Anwendung bzw. die Kylix-IDE gestartet wurde. Eine Alternative dazu wird unten vorgestellt. Das Fehlen der zweiten Option, die der FENSTERMELDUNGEN, kann als Motivation aufgefasst werden, sich unter Linux nicht mit nicht-portablen Ereignissen der GUI-Bibliothek (hier von X Window) zu befassen. Wollen Sie es dennoch tun, benötigen Sie dafür ein externes Tool.

Diagnosemeldungen ausgeben *R162*

Eine wesentlich flexiblere Möglichkeit, Diagnosemeldungen einer Anwendung auszugeben, außerhalb der Anwendung aufzufangen und aufzulisten, besteht in der Log-Funktion des Systems, die Sie auch aus Kylix heraus mit der *Libc*-Funktion *syslog* verwenden können. Um beispielsweise den String *DebugString* auszugeben, könnte der Aufruf wie folgt aussehen:

```
uses Libc, ...; // Libc muss in die uses-Zeile eingebunden werden.
...
syslog(LOG_INFO or LOG_USER, PChar(DebugString));
```

Dadurch wird der String an die Logdatei angehängt, die normalerweise unter `/var/log/messages` zu finden sein sollte. Zu den verschiedenen Optionen von *syslog* sei auf die entsprechende *man*-Hilfeseite verwiesen. Wir benötigen in diesem Beispiel nur noch eine komfortable Möglichkeit, die Log-Datei anzusehen. Hier bietet sich das Kommandozeilen-Programm *tail* an, das die letzten Zeilen einer Datei ausgibt und das Sie in einem Shell-Fenster wie folgt aufrufen können:

```
tail -f /var/log/messages
```

Die Option *-f* sorgt dafür, dass die Ausgabe erneuert wird, wenn die Datei wächst. Im Endeffekt wird Ihr Konsolen-Fenster auf diese Weise zu einem vollwertigen Ersatz für das »Ausgabemeldungen«-Ereignisprotokoll von Delphi für Windows. Für weitere Möglichkeiten von *tail* sei wieder auf die Online-Dokumentation *man tail* verwiesen.

In pattformübergreifenden Anwendungen können Sie *syslog* in eine eigene Ausgabefunktion einbauen, die unter Windows die *OutputDebugString*-Funktion aufruft:

```
procedure MyLogFunction(s: String);
begin
{$ifdef LINUX}
  syslog(LOG_INFO or LOG_USER, PChar(s));
{$endif}
{$ifdef WINDOWS} // Vorsicht: unter Delphi 5 nur Win32 vordefiniert!
  OutputDebugString(PChar(s));
{$endif}
end;
```

Das Beispielprogramm *MessageDemo* im Verzeichnis `Kapitel3` der CD benutzt einen Radioschalter, in dem Sie zur Laufzeit wählen können, ob die Meldungsausgabe per *syslog* oder *writeln* erfolgen soll.

1.7.3 Breakpoints

Kylix-Anwendungen sind aufgrund ihrer ereignisorientierten Struktur unmöglich schrittweise von Anfang bis Ende ablauffähig, denn zwischen den Ereignissen haben entweder X Window, Fenstermanager, Desktop-Umgebung oder die CLX die Kon-

trolle. Sie werden daher wahrscheinlich häufig damit beginnen, einen Haltepunkt (*Breakpoint*) in eine Methode zu setzen, die offensichtlich nicht funktioniert oder in der Sie einen Fehler vermuten.

Kylix unterscheidet zwischen drei Arten von Haltepunkten:

- Quelltext-Haltepunkte, die sich im Quelltexteditor durch eine rot hervorgehobene Zeile widerspiegeln, an deren linkem Ende sich ein Stopschild befindet. Bevor die rot markierte Zeile vom Prozessor ausgeführt wird, hält Kylix das Programm an.
- Adress-Haltepunkte sind vergleichbar mit den Haltepunkten im Quelltext, befinden sich aber quasi im CPU-Fenster. Sie bewirken, dass das Programm vor der Ausführung einer Maschineninstruktion an einer bestimmten *Adresse* angehalten wird.
- Daten-Haltepunkte sind unabhängig von einer bestimmten Stelle im Quelltext, denn sie beziehen sich nicht auf den Code, sondern auf die Daten eines Programms und führen zu einer Unterbrechung, bevor ein Schreibzugriff auf die angegebenen Daten stattfindet.

Unabhängig von der Art erlaubt doch jeder Haltepunkt, nachdem er das Programm einmal angehalten hat, dieselben Untersuchungsmöglichkeiten. Solange das Programm unterbrochen ist, können Sie:

- sich Variablen und Objekte des Programms ansehen, um festzustellen, ob diese die erwarteten Werte aufweisen,
- Ausdrücke berechnen lassen (zu beidem siehe Kapitel 1.7.4),
- jede Anweisung einzeln ausführen lassen,
- den Aufruf-Stack untersuchen (zu beidem siehe Kapitel 1.7.5)
- und natürlich weitere Breakpoints setzen, bestehende löschen oder vorübergehend ausschalten.

Temporäre Breakpoints

Falls Sie nur vorübergehend einen Breakpoint benötigen, ist die Funktion ZU CURSORPOSITION GEHEN einfacher zu handhaben: Kylix setzt, ohne die Zeile zu markieren, einen temporären Breakpoint in die Zeile, in der sich momentan der Cursor befindet. Sobald diese Zeile erreicht wird, stoppt Kylix das Programm und löscht den Haltepunkt wieder. Falls das Programm vor dem Erreichen der gewünschten Zeile durch einen anderen Haltepunkt gestoppt wird, geht dieser temporäre Breakpoint jedoch verloren. Um dennoch bei dieser Zeile anzuhalten, müssen Sie den Cursor erneut darauf positionieren und die Funktion ZU CURSORPOSITION GEHEN ausführen.

Programmausführung fortsetzen

Um ein angehaltenes Programm weiterlaufen zu lassen, verwenden Sie denselben Befehl wie zum Starten des Programms, also START | START, F9, oder den entsprechenden Schalter der Symbolleiste.

Dauerhafte Breakpoints

Kommen wir zu den dauerhaften Haltepunkten, die über einige zusätzliche Merkmale verfügen können, und zwar sollen im Folgenden zunächst nur die Quelltext-Haltepunkte gemeint sein. Die einfachste Möglichkeit, einen solchen Haltepunkt zu setzen bzw. wieder abzuschalten, ist über das klassische Tastenkürzel Strg+F8 bzw. über einen Klick in das Editorfenster links neben der Haltezeile. Alle weitergehenden Optionen finden Sie in einem eigenen Fenster, das alle derzeitigen Unterbrechungspunkte auflistet (Menü: ANSICHT | DEBUG-FENSTER | HALTEPUNKTE) – Abbildung 1.17, Fenster *Haltepunkteliste*.

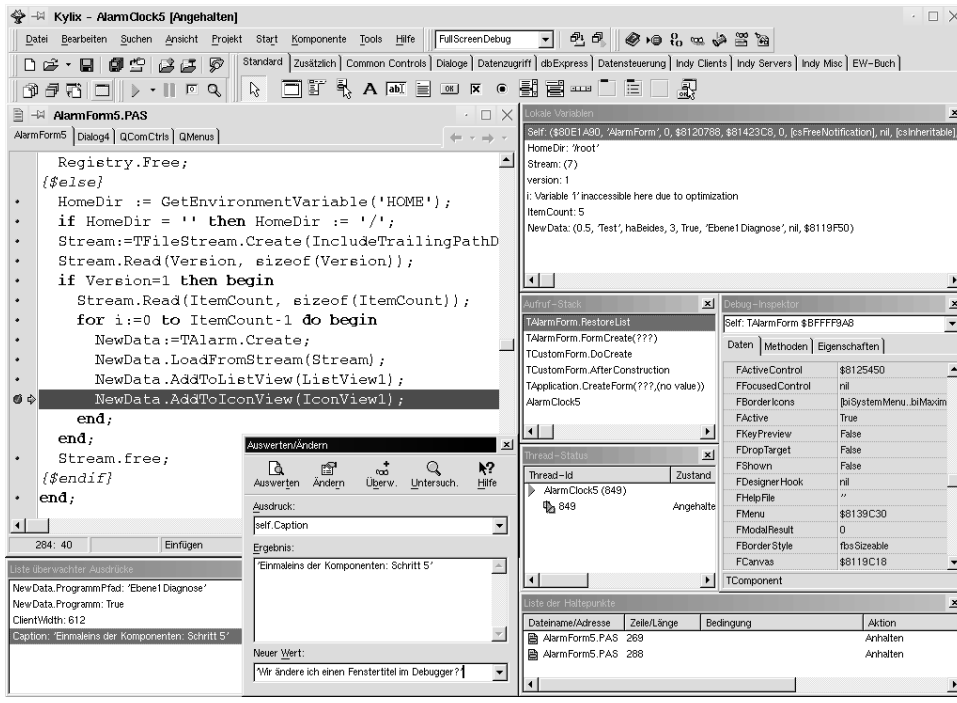


Abbildung 1.17: Übersicht über weitere Fenster des Debuggers

In diesem Fenster können Sie Haltepunkte über das lokale Popup-Menü vorübergehend abschalten, was gegenüber dem Löschen den Vorteil hat, dass Sie sie schnell wie-

der anschalten können, ohne zurück zur Unterbrechungsposition blättern zu müssen. Alle weiteren Merkmale der Haltepunkte werden in einem Dialogfenster bearbeitet, das Sie schnell mit einem Doppelklick oder mit ↵ aufrufen können, oder über das Popup-Menü der Haltepunktmarkierung im Quelltext-Fenster.

Neben den Eigenschaften DATEINAME und ZEILENNUMMER, die ja schon durch das Setzen des Haltepunktes im Editor festgestellt werden, gibt es noch einige weitere:

- BEDINGUNG: Wenn Sie hier einen Ausdruck eintragen, der *True* oder *False* ergibt, z.B. »*StringList.Count<10*«, prüft Kylix jedes Mal, wenn der Haltepunkt erreicht wird, diese Bedingung und hält das Programm nur dann an, wenn sie zutrifft, also den Wert *True* ergibt.
- DURCHLAUFZÄHLER: Soll die Programmzeile erst mehrere Male ohne Unterbrechung durchlaufen werden, tragen Sie die Zahl der Durchläufe in diesem Feld ein. Es werden hierbei nur die Durchläufe gezählt, bei denen auch die Bedingung des vorhergehenden Feldes zutrifft. Wenn die Zahl der Durchläufe erreicht ist, hält Kylix beim nächsten Mal an dieser Stelle an und setzt den internen Durchlaufzähler dieses Haltepunkts auf Null zurück.
- Jedem Haltepunkt können Sie einen Gruppennamen zuweisen. Die Gruppen dienen dazu, mehrere Haltepunkte gemeinsam ein- und ausschalten zu können. Das lokale Menü des Haltepunktfensters enthält zu diesem Zweck die zwei neuen Punkte GRUPPE AKTIVIEREN und GRUPPE DEAKTIVIEREN, unter denen jeweils alle von Ihnen definierten Gruppennamen aufgelistet werden.
- Hinzukommen eine ganze Reihe von Einstellungen für Haltepunkt-Aktionen (im Haltepunkt-Dialog erreichbar über den Schalter WEITERE). Haltepunkt-Aktionen werden nachfolgend in einem eigenen Abschnitt erläutert.

Adress- und Datenhaltepunkte

Die Eigenschaften BEDINGUNG und DURCHLAUFZÄHLER finden Sie auch bei den Adress- und Datenhaltepunkten wieder, hinzu kommt bei den Adresshaltepunkten logischerweise noch eine Adresse, die Sie normalerweise dadurch festlegen, dass Sie einen Haltepunkt auf einer einzelnen Maschinen-Instruktion im CPU-Fenster setzen (per Mausklick an den linken Rand des CPU-Fensters oder über das auch für Quelltext-Haltepunkte verwendete Tastenkürzel).

Datenhaltepunkte werden durch eine Adresse und eine Längenangabe definiert:

- Die Adresse ist zwar hier nichts weiteres als eine achtstellige Hexadezimalzahl, Sie können diese jedoch auch festlegen, indem Sie eine Speicherposition wie in Object Pascal durch einen Variablennamen oder ähnlichen Ausdruck (z.B. *Zahlen[100]*) angeben.

- Die Länge gibt an, wie viele Bytes ab der genannten Adresse überwacht werden sollen. Für *Integer*-Variablen genügen schon die 4 standardmäßig voreingestellten Bytes; wenn Sie 10 Zeichen eines *Char*-Arrays überwachen wollen, geben Sie stattdessen eine 10 ein.

Der Debugger hält daraufhin das Programm an, noch bevor es zu einem Schreibzugriff auf die angegebene Adresse kommt. Falls möglich zeigt er die aktuelle Programmposition im Quelltexteditor an, Sie können daraufhin aber sogar in das CPU-Fenster schalten, um genau die Maschinenanweisung sehen zu können, die im nächsten Taktzyklus der CPU den Speicherzugriff durchgeführt hätte.

Haltepunkt-Aktionen

Die Haltepunkt-Aktionen erlauben Ihnen, beim Eintreten einer Haltepunkt-Bedingung bestimmte Aktionen vom Debugger automatisch ausführen zu lassen. Sie können sogar darauf verzichten, dass der Debugger das Programm anhält und die Kontrolle an Sie weitergibt, so dass Sie aus einem Haltepunkt quasi einen Aktionspunkt machen. Neben der voreingestellten Aktion ANHALTEN stehen folgende Aktionen zur Wahl:

Mit SPÄTERE EXCEPTIONS IGNORIEREN und SPÄTERE EXCEPTIONS BEHANDELN, verteilt auf zwei verschiedene Haltepunkte, können Sie einen bestimmten Programmteil von der standardmäßigen Exception-Behandlung ausklammern – etwa weil Ihnen bereits bekannt ist, dass darin zahlreiche Exceptions auftreten werden, die Sie aber nicht jedes Mal in einem Meldungsfenster angezeigt bekommen wollen. Gegenüber der generellen Abschaltung aller Exception-Reaktionen hat dies den Vorteil, dass Sie von eventuell überraschend auftretenden Exceptions in anderen Programmteilen dennoch informiert werden.

MELDUNG PROTOKOLLIEREN schreibt den Text, den Sie in den Haltepunkt-Eigenschaften eingeben, in das Ereignisfenster.

EVAL-AUSDRUCK wertet einen Ausdruck aus und protokolliert das Ergebnis im Ereignisfenster. Diese Protokollierung können Sie auch ausschalten, was dann Sinn macht, wenn bei der Auswertung des Ausdrucks Seiteneffekte entstehen, weil etwa eine Methode des Programms ausgeführt wird.

- GRUPPE AKTIVIEREN und GRUPPE DEAKTIVIEREN entspricht den gleichnamigen Punkten des Popup-Menüs und kann auf ähnliche Weise verwendet werden wie die anfangs genannten Optionen zum Ein- und Ausschalten der Exception-Behandlung.

1.7.4 Variablen untersuchen

Das häufigste Ziel beim Anhalten des Programms besteht darin, die Werte von Properties, Variablen und Ausdrücken zu untersuchen. Hierfür gibt es je nach Bedürfnis verschiedene Methoden.

Das Variablenfenster

Das Variablenfenster (ANSICHT | DEBUG-FENSTER | LOKALE VARIABLEN) zeigt immer die gerade gültigen lokalen Variablen mitsamt ihren Werten an und erspart Ihnen dadurch eine Menge an manuellen Aufrufen von Variablen-Auswertungsfenstern. Wenn Sie die Optimierung nicht wie in Kapitel 1.7.1 angeraten abgeschaltet haben, kommt es jedoch sehr häufig vor, dass Variablen wegoptimiert wurden und so deren Inhalte nicht angezeigt werden.

Auswertung durch Kurzhinweis

Eine zweite sehr bequeme Möglichkeit der Variablenauswertung sind die ToolTip-Fenster: Wenn Sie bei angehaltenem Programm den Mauszeiger über einen Variablennamen im Quelltexteditor halten, erhalten Sie ein Hinweisfenster mit dem aktuellen Wert dieser Variablen. Diese Debuggerfunktion gehört zu den einzeln an- und abschaltbaren »Programmierhilfen« und verwechselt Variablen in *with*-Blöcken (hierzu und zu den Programmierhilfen siehe Kapitel 1.6.1).

Ausdrücke dauerhaft überwachen

Wollen Sie einen bestimmten Ausdruck über eine längere Zeit beobachten, können Sie ihn in ein so genanntes Watch-Fenster setzen. In der Übersetzung wurde *Watch* zu *Überwachte Ausdrücke*. Dieses Fenster (Menü ANSICHT | DEBUG-FENSTER | ÜBERWACHTE AUSDRÜCKE, in Abbildung 1.17: Fenster *Liste überwachter Ausdrücke*) enthält eine Liste von Ausdrücken, die Sie nach und nach erweitern, verändern und verkleinern können. Neben einem Ausdruck steht jeweils sein aktueller Wert. Jedes Mal wenn das Programm erneut angehalten wird, erneuert Kylix die Anzeige der Werte, da diese sich inzwischen geändert haben können. Es findet jedoch keine Aktualisierung des Fensters in Echtzeit statt, d.h., wenn das Programm einen Wert ändert, erfährt Kylix das erst, wenn das Programm das nächste Mal angehalten wird.

Um die Variable, die sich im Editor an der Cursorposition befindet, im Überwachungsfenster einzutragen, wählen Sie AUSDRUCK AM CURSOR ANZEIGEN... aus dem lokalen Menü des Editors bzw. das zugehörige Tastenkürzel. Wenn das Überwachungsfenster bereits angezeigt wird, können Sie die Maus auch zum Drag&Drop von markierten Ausdrücken vom Editor- in das Überwachungsfenster verwenden (wenn das Programm gerade nicht im Debugger untersucht wird, müssen Sie dabei noch Alt drücken). Im Überwachungsfenster selbst haben Sie eine Reihe weiterer Möglichkeiten, die Ihnen das dort befindliche Popup-Menü zeigt.

Angabe von Ausdrücken

Bei den Ausdrücken kann es sich um einzelne Variablen, Properties und Funktionsaufufe oder um eine Verknüpfung von solchen handeln. Der Ausdruck

```
List1.ItemIndex >= LastSelection+1
```

ist zum Beispiel *True*, wenn das *ItemIndex*-Property einer *ListBox List1* größer dem Wert *LastSelection* plus 1 ist (bei diesem Beispiel wurde kein tieferer Sinn beabsichtigt). Mehr zu Ausdrücken in Object Pascal finden Sie in Kapitel 2.4.2.

ür die Ausdrücke in den Debugger-Fenstern gelten kleine Besonderheiten:

Sie können darin auch Prozessorregister angeben (z. B. *eax*, *ebx*, *esi*).

Hinter dem Ausdruck können Sie ein Komma und eine Formatanweisung folgen lassen. Mit der Angabe »eax, h« können Sie den Wert des eax-Registers beispielsweise in hexadezimaler Form überwachen. Eine Übersicht über die Formatanweisungen erhalten Sie in der Online-Hilfe unter dem Stichwort *Formatbezeichner*. Im Fenster der überwachten Ausdrücke brauchen Sie diese Formatanweisungen allerdings nicht, da Sie das Format dort durch eine Schaltergruppe festlegen können.

- Funktionsaufrufe liefern oft nicht nur ein Ergebnis, sondern führen im Programm eine Aktion aus und beeinflussen dadurch als so genannter *Nebeneffekt* den nachfolgenden Programmablauf. Daher sind Funktionsaufrufe im Überwachungsfenster standardmäßig nicht erlaubt, was Sie aber im Eigenschaftsdialog der einzelnen Ausdrücke ändern können (Dialogfeld FUNKTIONSAUFRUFE GESTATTEN).

Inspektorfenster

Die Inspektorfenster (siehe Abbildung 1.17) sind besonders dazu geeignet, Objekte und Arrays detailliert darzustellen. Sie können schnell mit Alt+F5 oder über das lokale Menü des Editorfensters aufgerufen werden (FEHLERSUCHE | UNTERSUCHEN...). Wenn Sie die Eingabemarkierung vorher auf einen Bezeichner im Quelltexteditor setzen, erhalten Sie sofort das zugehörige Inspektorfenster, ansonsten erfragt Kylix das anzuzeigende Programmobjekt über einen Dialog. Auch hier steht wieder die Möglichkeit des Drag&Drop offen – vom Quelltext-Editor in ein bestehendes Inspektorfenster.

In einem Inspektorfenster wird jedes Objekt- bzw. Array-Element in einer eigenen Zeile dargestellt, Objekt-Inspektorfenster verteilen Daten, Properties und Methoden auf drei verschiedene Seiten. Da es beim Lesen von Properties über Lesemethoden in diesen Methoden zu unerwünschten Nebeneffekten kommen kann, werden solche Properties nicht automatisch vom Debugger ausgelesen und angezeigt. Um die Werte dieser Properties zu sehen, müssen Sie auf den Mini-Schalter neben dem Property drücken. Für die einzelnen Datenelemente können Sie außerdem weitere Inspektorfenster aufrufen und so nur per Mausklicks große verzweigte Zeigerstrukturen durchwandern.

Eine wichtige Option des Popup-Menüs ist die TYPUMWANDLUNG, mit der Sie dem Debugger den »wahren« Typ eines polymorphen Objekts mitteilen können. Inspizieren Sie beispielsweise eines der häufig als Parameter auftretenden *Sender*-Objekte, zeigt Ihnen das Inspektorfenster nur die in *TObject* deklarierten Datenelemente an, da *Sender* als *TObject* deklariert ist. Wenn es sich bei *Sender* aber z.B. um eine *TListBox*-Komponente handelt, müssen Sie die Klasse *TListBox* zuerst im Typumwandlungs-Dialog angeben, um auch die exklusiven Datenelemente von *TListBox* angezeigt zu bekommen.

Ausdrücke auswerten, Objekte untersuchen und Werte ändern

Schließlich gibt es in Kylix noch ein AUSWERTEN/ÄNDERN-Fenster (ebenfalls in Abbildung 1.17 gezeigt). Gegenüber den Kurzhinweisfenstern hat er die folgenden Vorteile anzubieten:

- Sie können wie im Überwachungsfenster auch Ausdrücke auswerten und Formatanweisungen angeben;
- Sie können Variablen angeben, die gerade nicht im Editor genannt werden, vor allem die Variable *self* für das gerade aktive Objekt;
- Sie können Variablenwerte ändern.

Die letzten beiden Vorteile können Sie natürlich auch bei Verwendung der Inspektor-Fenster genießen. Das Auswerten/Ändern-Fenster finden Sie im globalen START-Menü, im lokalen Editormenü und je nach gewählter Tastaturbelegung unter Strg+F7 oder Strg+F4.

1.7.5 Code-Ausführung

Wenn Sie einige Ausdrücke in das Überwachungsfenster geschrieben haben oder wenn Sie nur den schrittweisen Ablauf des Programms verfolgen wollen, können Sie sich der Einzelschrittbefehle des Debuggers bedienen. Davon gibt es zwei verschiedene Typen, deren (englische) Kurzbezeichnung *Step* bzw. *Trace* ist. In der Symbolleiste finden Sie beide Befehle in den beiden Schaltern rechts unten. Im START-Menü haben sie die folgenden Namen:

- Mit GESAMTE ROUTINE (Step) führen Sie den Quelltext zeilenweise aus. Die Übersetzung rührt daher, dass Funktionsaufrufe in diesem Einzelschrittmodus als Ganzes ausgeführt werden.
- Im Gegensatz dazu gelangen Sie mit EINZELNE ANWEISUNG in jede Funktion, die in der aktuellen Zeile aufgerufen wird und zu der der Quelltext vorliegt. Allerdings ist auch hier der bescheidenere Name *Trace* besser, denn wenn sich mehrere einfache Anweisungen ohne Routinenaufruf in einer Zeile befinden, werden diese mit EINZELNE ANWEISUNG eben nicht einzeln, sondern als gesamte Zeile ausgeführt.

Größere Schritte

Ein weiterer nützlicher Eintrag im START-Menü ist AUSFÜHRUNG BIS RÜCKGABE. Er lässt die aktuelle Methode vollständig ablaufen und hält das Programm dann bei ihrem abschließenden *end;* wieder an.

Interessant ist auch ein spezieller Einzelschritt-Befehl, der sich lohnt, wenn Sie sich im CPU-Fenster und in einem Bereich befinden, zu dem es keinen Quelltext gibt: START | NÄCHSTE QUELLTEXTZEILE lässt das Programm dann bis zur nächsten Quelltextzeile weiterlaufen und hält es dort an.

In diesem Zusammenhang sei auch noch einmal die in Kapitel 1.7.3 beschriebene Funktion ZU CURSORPOSITION GEHEN erwähnt, die Sie nicht nur als temporären Breakpoint, sondern auch zum Überspringen mehrerer Anweisungen einsetzen können.

Der Aufruf-Stack

Abbildung 1.17 enthält (außer dem Fenster *Thread-Status*, das wir hier nicht weiter beachten) mit *Aufruf-Stack* noch ein letztes, bisher nicht erwähntes Fenster. Es zeigt die Funktionsaufrufe, die zurzeit aktiv sind, soweit der Debugger den Aufruf-Stack zurückverfolgen kann. So zeigt das Fenster in der Abbildung beispielsweise, dass die Methode *UpdateMemoStatus* von der Methode *FormKeyDown* aufgerufen wurde. Um die Zeile zu finden, in welcher der Aufruf einer Methode stattfand, doppelklicken Sie im Aufruf-Stack-Fenster auf diese Methode, und Sie gelangen zu der Zeile, die dem Aufruf folgt.

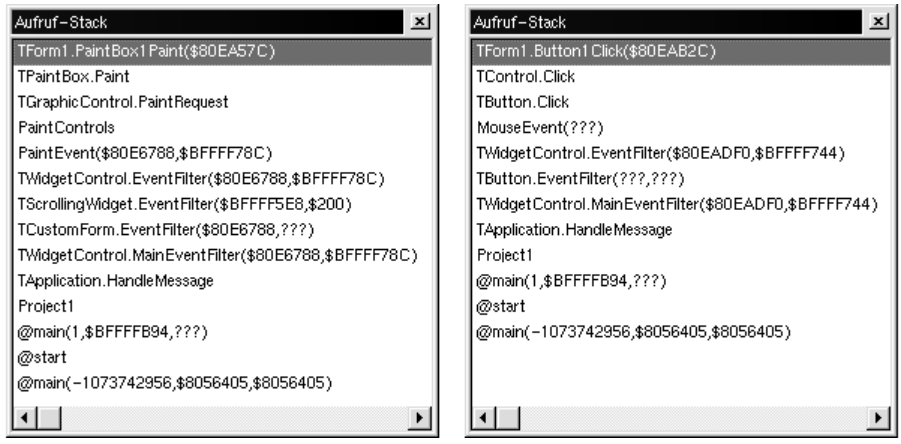


Abbildung 1.18: Diese Stack-Fenster zeigen die Hintergründe zum Aufruf des OnPaint-Ereignisses eines rmulars und des OnClick-Ereignisses eines Schalters.

Mit Hilfe des Quelltextes der CLX können Sie, wenn Sie ihn mit Debug-Informationen in das Programm eingebunden haben, im Stack-Fenster sogar viel über die internen Abläufe der CLX lernen (Abbildung 1.18).

Debuggen mit dem CLX-Quelltext *R161*

Auch, wenn es meistens sehr vorteilhaft ist, dass die CLX unzählige Details vor dem Entwickler versteckt, kann es sehr interessant sein, ihre internen Abläufe per schrittweiser Programmausführung genauestens zu untersuchen. Dies kann großen Nutzen bei der Fehlersuche bringen und vor allem förderlich für das Verständnis der CLX sein (da Sie z.B. Informationen wie in Abbildung 1.18 erhalten können). Damit der integrierte Debugger in Besitz der Debugging-Informationen der CLX kommt, genügt es, in den Projektoptionen auf der Seite COMPILER die Option MIT DEBUG-DCUS zu markieren und das Projekt neu zu erzeugen (PROJEKT | *Projektname ERZEUGEN*).

Eine allgemeine Vorgehensweise, die zum selben Ergebnis führt, aber zusätzlich noch den Vorteil hat, dass eventuelle Änderungen im CLX-Quelltext mitkompiliert werden, ist folgende:

- Zunächst müssen die Optionen aktiviert sein, die sowieso schon zum Debuggen notwendig sind.
- Fügen Sie im Feld *Pfad für Bibliothek* in den Umgebungsoptionen auf der Seite *Bibliothek* den Pfad des CLX-Quelltextes ein (z.B. `/root/kylix/source/clx`). Das CLX-Verzeichnis muss dabei vor dem *Lib*-Verzeichnis stehen und von diesem durch einen Doppelpunkt getrennt werden.
- Übersetzen Sie dann Ihr aktuelles Projekt komplett neu.

Egal, welche Vorgehensweise Sie gewählt haben, ein schneller Test, ob es funktioniert hat, ist folgender: Starten Sie das Projekt schließlich mit START | EINZELNE ANWEISUNG. Wenn die Installation der CLX-Debug-Informationen erfolgreich war, müsste der Debugger Ihr Programm statt in einer Ihrer Dateien im *initialization*-Bereich einer der CLX-Units anhalten.

Hinweis: Auch wenn Sie die CLX-Verzeichnisse in den Projektoptionen unter VERZEICHNISSE/BEDINGUNGEN in den SUCHPFAD eintragen und das Projekt neu erzeugen, kompiliert Kylix die Quelltextdateien der CLX neu und mit Debug-Informationen, falls die entsprechende Option aktiviert ist.

1.7.6 Assertions

Mit den in den bisherigen Abschnitten erläuterten Debugger-Funktionen können Sie sich während des Programmablaufs selbst davon überzeugen, ob alles wie erwartet

abläuft, ob z.B. eine *while*-Schleifen-Variable richtig bis zum Wert 0 »heruntergezählt« wird – kurz: Sie können interaktiv überprüfen, ob bestimmte *Bedingungen* zutreffen.

Kylix unterstützt auch das maschinelle Überprüfen solcher Bedingungen in einer besonderen Weise:

Mit der neuen Standardanweisung *Assert* können Sie eine Bedingung überprüfen, die beim korrekten Ablauf des Programms gelten muss. *Assert* zeigt ein Meldungsfenster an, wenn die Bedingung einmal nicht erfüllt ist. So gesehen ähnelt die Anweisung *Assert(Bedingung, Meldung)* erst einmal nur der Anweisung *if not Bedingung then Meldunganzeigen(Meldung)*.

- Ihren besonderen Status erhält die *Assert*-Anweisung jedoch dadurch, dass Sie in den Compileroptionen einstellen können, ob die *Assert*-Anweisungen im Quelltext beachtet werden sollen oder nicht (siehe auch Abbildung 1.14, Schalter *Assertion*).

Während der Programmentwicklung können Sie also durch großzügigen Gebrauch dieser Bedingungstests möglicherweise Fehler schneller finden. Wenn das Programm dann fertig ist, schalten Sie alle *Assert*-Anweisungen auf einmal aus. Angenommen, dass in Ihrem fertigen Programm alle gewünschten Bedingungen sowieso immer zutreffen, dann hat das Abschalten der *Assert*-Anweisungen noch den Vorteil, dass die Ablaufgeschwindigkeit des Programms erhöht und die Größe der kompilierten Datei reduziert wird. Ein ausführlicheres Beispiel:

```
i:=1;
repeat
  Assert(i<100, 'i<100 gilt nicht');
  i:=i+2;
until i=100;
```

In diesem Programmbeispiel soll sichergestellt werden, dass die Schleife wirklich nur so lange ausgeführt wird, wie *i* kleiner als 100 ist. Da *i* jedoch nur ungerade Werte annimmt und am Ende der Schleife nicht *i>=100* abgefragt wird, erreicht *i* irgendwann den Wert 101, ohne dass die Schleife abgebrochen wird. Sofort schlägt die *Assert*-Anweisung zu und weist auf diesen Fehler hin.

Hinweis: Bei den voreingestellten Umgebungsoptionen für den Debugger unterbricht der integrierte Debugger das Programm an der Stelle der fehlgeschlagenen *Assert*-Anweisung. Die Werte der Variablen sind zu diesem Zeitpunkt jedoch schon nicht mehr definiert. Wenn Sie den Wert von *i* anzeigen lassen, erhalten Sie also in diesem Beispiel mit höchster Wahrscheinlichkeit nicht mehr den Wert 101, sondern beispielsweise 4349104.

Gegenüber den *Assert*-Makros aus C++ hat die Object-Pascal-Anweisung zwar den Nachteil, dass der genaue Wortlaut der Bedingung nicht automatisch im Meldungsfenster angezeigt wird, sondern dass Sie sie in den Meldungsstring kopieren müssen, trotzdem erspart Ihnen eine einzige *Assert*-Anweisung den folgenden Code, der – hundertfach angewendet – Ihre Quelltextdatei ganz schön unübersichtlich machen würde:

```
{$ifdef AktiviereAssertions}
  if not (i<100) then
    // Meldung ausgeben (bzw. Exception erzeugen, siehe Hinweis unten)
{$endif}
```

Hinweis: Genau genommen ist es nicht die Anweisung *Assert*, die das Meldungsfenster öffnet, sondern die automatische Exception-Behandlung der CLX. *Assert* erzeugt lediglich eine *EAssertionFailed*-Exception, durch die die Ausführung des Codes, in der die Bedingung nicht zutrifft, abgebrochen wird. Daraufhin übernimmt die CLX die Kontrolle und gibt das Meldungsfenster aus. Weitere Details zu Exceptions lesen Sie in Kapitel 2.6.

1.8 Komponenten kurz vorgestellt

In den folgenden Abschnitten geht es nicht mehr um grundlegende Abläufe in der Kylix-IDE, sondern um konkrete Komponenten: Eingebunden in eine Übersicht über die Komponentenpalette finden Sie hier eine Reihe weiterer einfacher Beispiele zur Verwendung und Programmierung von Komponenten in Kylix. Die Übersicht orientiert sich an der Funktion der Komponenten und nicht an der Aufteilung der Komponentenpalette. So finden Sie beispielsweise in Kapitel 1.8.4 eine Übersicht über alle Eingabekomponenten von Kylix, egal ob sie sich nun auf der Seite *Standard*, *Zusätzlich* oder *Common Controls* befinden.

Wenn Sie das »Einmaleins« der Komponenten beherrschen, können Sie in Kapitel 3.1 bis 3.4 eine fundierte Einführung in die Grundlagen und Gemeinsamkeiten aller Komponenten mitsamt der dahinter liegenden Klassenhierarchie der CLX lesen.

Die neue Beispielprogramm-Serie

Das in den folgenden Abschnitten verwendete Beispielprogramm basiert auf einer kleinen Uhr mit Alarmfunktion. Die fünf Versionen des Beispielprogramms finden Sie auf der CD unter den Projektnamen *AlarmClock1* bis *AlarmClock5* (siehe Abbildung 1.19). Im Folgenden sollen kurz die Grundprinzipien von Uhr- und Weckfunktion erläutert werden. Diese haben mit dem eigentlichen Thema des Kapitels, der Vorstellung der Komponenten, nur insofern etwas zu tun, als Ihnen dabei die *Timer*-Komponente vorgestellt wird.

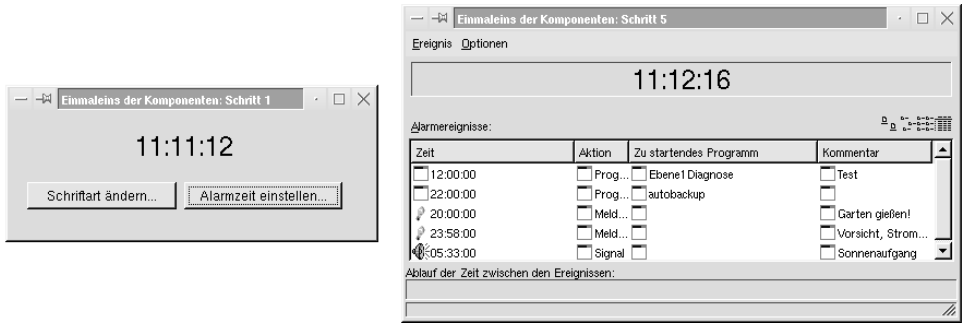


Abbildung 1.19: Das Formular des AlarmClock-Programms nach dem ersten Schritt (links) und nach Abschluss der Entwicklung

Wie schon das Beispielprogramm aus den Kapiteln 1.4 und 1.5 verfügt auch dieses Programm über einen Schriftwahlschalter, der sich hier auf die Anzeige der Uhrzeit bezieht. Er funktioniert genauso wie der Schalter aus dem Bashrc-Editor, braucht also hier nicht weiter erläutert zu werden.

Ermittlung und Anzeige der Uhrzeit

Zunächst ist die Frage zu klären, woher das Programm weiß, wie viel Uhr es ist und wie es die in nackten Zahlen vorliegende Uhrzeit in eine menschenlesbare Ziffernkette im Format 00:00:00 umwandeln kann. Dies geschieht mit den Funktionen der Standard-Unit *SysUtils*: Die Funktion *Time* liefert die aktuelle Zeit in einem Format mit der Bezeichnung *TDateTime*. Dieses brauchen Sie nicht zu kennen (ungewöhnlicherweise handelt es sich übrigens um eine Fließkommazahl), denn andere Funktionen derselben Unit sind in der Lage, die *TDateTime*-Angabe in eine Zeichenkette oder in einzelne Zahlenwerte für Stunden, Minuten und Sekunden (und sogar Hundertstelsekunden) umzuwandeln. Um eine Zeichenkette (*String*) zu erhalten, wenden Sie die Funktion *meToStr* auf das Ergebnis der Funktion *Time* an:

```
UhrzeitAlsString := TimeToStr(Time);
```

Das Timer-Ereignis

Die Anzeige der Uhrzeit im Beispielprogramm soll nun jede Sekunde erneuert werden. Dazu ist ein Timer-Ereignis erforderlich, das mindestens einmal pro Sekunde ausgelöst wird. Hierfür verwendet das Programm eine *Timer*-Komponente, die Sie in der Palette auf der Seite *Zusätzlich* finden und über deren Property *Interval* Sie einstellen, alle wie viel Millisekunden das *OnTimer*-Ereignis der Komponente ausgelöst werden soll). Dieses Ereignis kann wie folgt bearbeitet werden, um die Anzeige der Uhrzeit zu gewährleisten. Dabei ist *TimeText* eine Komponente des bereits bekannten Typs *TLabel*:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
begin
  TimeText.Caption := TimeToStr(Time);
end;
```

Die Alarmfunktion

Zusätzlich zur Zeitanzeige soll das Beispielprogramm mit verschiedenen Alarmfunktionen ausgestattet werden, von der einfachen Meldungsanzeige bis zum automatischen Starten eines Programms. Die Konfiguration der Alarme und die Anzeige der Einstellungen im Fenster wird mit jeder neuen Programmversion weiterentwickelt. Die Grundfunktion, die in allen Schritten gleich ist, besteht darin, jede Sekunde festzustellen, ob jetzt ein Alarm fällig ist. Außerdem ist es dem Benutzer gestattet, die Alarmfunktion zu deaktivieren, weshalb das Programm auch eine Variable *AlarmAktiviert* verwaltet.

Für die Alarmfunktion muss das Programm lediglich die aktuelle Zeit (*Time*) mit der Alarmzeit vergleichen. Ist die aktuelle Zeit größer, kann der Alarm ausgelöst werden. Die Alarmzeit wird dafür in einer Variablen namens *AlarmTime* angegeben, der Vergleich von aktueller Zeit und Alarmzeit ist einfach durch *Time* >= *AlarmTime* möglich.

Zu bedenken ist allerdings noch, dass der Alarm nur einmal ausgelöst werden soll, auch wenn der Vergleich in allen nachfolgenden Sekunden immer noch *Time* >= *AlarmTime* ergibt. Hierzu wird die zuletzt aktivierte Alarmzeit in der Variablen *LastAlarm* gespeichert.

Die Variable *LastAlarm* wird in der Formularklasse deklariert, wie in Kapitel 1.5.5 bereits die Methode *UpdateMemoStatus*. Die Alarmzeit wird entweder in einer zweiten Variablen (*AlarmTime*) oder ab *AlarmClock3* in einer Liste für mehrere Zeiten gespeichert. Der Vergleich von aktueller Zeit und Alarmzeit wird in der Funktion *IsTimeOver* gekapselt, die ebenfalls selbst im Formular deklariert wird:

```
type
  TAlarmClockMainForm = class(TForm)
    ...
    ... (Bereich der von Delphi erzeugten Deklarationen)
    ...
  private
    LastAlarm: TDateTime;
    AlarmTime: TDateTime;
    AlarmAktiviert: Boolean;
    function IsTimeOver(AlarmTime : TDateTime) : boolean;
  end;

implementation

function TAlarmClockMainForm.IsTimeOver(AlarmTime : TDateTime) : boolean;
begin
```

```
  Result := (Time >= AlarmTime) and (AlarmTime > LastAlarm);
end;

...
```

Hinweis: Die obigen Listings sollen nur dazu dienen, die grundsätzliche Arbeitsweise der Alarmfunktion zu illustrieren. Sollten Sie sich die konkreten Beispielprogramme *AlarmClock1* bis *AlarmClock5* auf der CD-ROM ansehen, müssen Sie auf Unterschiede in den Details gefasst sein. Diese haben einen gewichtigen Grund (ab *AlarmClock3* wird nicht mehr nur eine, sondern mehrere Alarmzeiten gespeichert) und einige unwesentliche Gründe (der Autor wollte plötzlich in einer Programmversion etwas ändern, war aber nicht in der Lage, alle anderen vier Programmversionen entsprechend gleichzuschalten).

String-Konvertierung

Fragt sich nur noch, wie wir an den Wert der Variablen *Alarm* kommen, denn dieser muss das Format *TDateTime* haben, während die *TEdit*-Eingabefelder einen String liefern. Für die Konvertierung eines Strings in einen Zeitwert ist aber schon vorgesorgt: Die Unit *SysUtils* liefert dafür die Funktion *StrToTime*. Allerdings kann nicht jeder String in einen Zeitwert konvertiert werden. Wie das Programm die Zeitangaben des Benutzers überprüft, wird im folgenden Kapitel erläutert.

1.8.1 Verwendung von Formularen als modale Dialoge

Da alle folgenden Versionen des Beispielprogramms mit einem Dialog zur Eingabe der Alarmzeiten arbeiten, nutzen wir zunächst diese Gelegenheit, um uns etwas weiter mit der wichtigsten Komponente eines Kylix-Programms zu beschäftigen – mit dem Formular (das zwar nicht in der Komponentenpalette auftaucht, aber in der Vererbungshierarchie der CLX auch als Komponente angesehen wird). Ab Kapitel 1.8.2 geht es dann mit den »normalen« Komponenten weiter.

Das Dialogformular *R35*

Wir nehmen nun an, dass das Hauptformular des Beispielprogramms (in Abbildung 1.20 links) bereits erzeugt wurde, und kümmern uns um die Erzeugung des Dialogformulars. Dies wird ein Beispiel dafür sein, wie Sie Dialogformulare auch zu beliebigen anderen Kylix-Projekten hinzufügen können.

Ein neues Formular erzeugen Sie über das Datei-Menü der Kylix-IDE. Wenn Ihnen ein leeres Formular ausreicht, wählen Sie einfach den Punkt DATEI | NEUES FORMULAR. Zur Erzeugung eines Formulars auf Basis einer Vorlage oder mit Hilfe eines Experten verwenden Sie den Punkt DATEI | NEU... (siehe Kapitel 1.6.4). In beiden Fällen erhält

Ihr Projekt zwei neue Dateien: `form?.xfm` und `form?.pas` (zum Aufbau von Projekten siehe Kapitel 1.6.3). Letztere enthält den Object-Pascal-Code zum Formular und hat nach ihrer Erzeugung den in Kapitel 1.5.3 beschriebenen Standard-Aufbau.

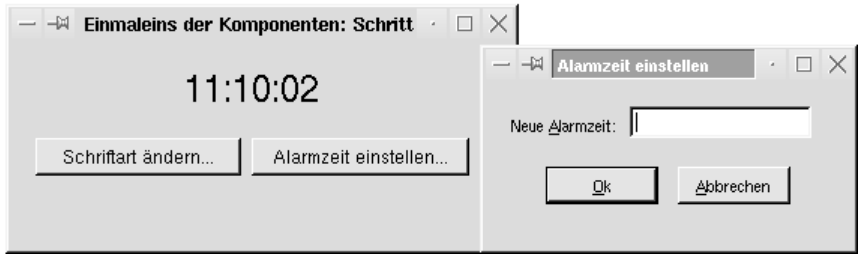


Abbildung 1.20: AlarmClock1 besteht aus zwei sehr einfach gehaltenen Formularen.

Wie die Abbildung zeigt, erhält das neue Formular zunächst ein Editierfeld mit vorangehender Beschriftung sowie zwei Aktionsschalter. Um das neue Formular zu einem funktionsfähigen Dialog zu machen, werden die folgenden Properties gesetzt:

Property-Einstellung	**Wirkung**
Formular: BorderStyle = bsDialog	Zur Laufzeit erhält das Fenster einen nichtverstellbaren Dialograhmen.
Formular: Position = poScreenCenter	Das Fenster wird zur Laufzeit in der Mitte des Bildschirms angezeigt.[a]
Formular: Name = AlarmEditDlg	Bessere Lesbarkeit des Quelltextes.
OK-Schalter: ModalResult = mrOK	Das Drücken dieses Schalters bewirkt das Schließen des Fensters bei Rückgabe des Wertes *mrOK* (wird im Programm abgefragt).
Abbruch-Schalter: ModalResult = mrCancel	Analog zum OK-Schalter ist das Ergebnis hier *mrCancel*.
OK-Schalter: Default = True	Vereinfacht die Tastaturbedienung: Der Schalter ist als Vorgabeschalter per Eingabetaste aktivierbar, und zwar unabhängig davon, welches Dialogelement gerade aktiv ist (mehrzeilige Eingabefelder fangen die Eingabetaste allerdings per Voreinstellung ab).
Abbruch-Schalter: Cancel = True	Ebenso eine Vereinfachung der Tastaturbedienung: Macht den Schalter zum Schalter, der bei Betätigung von Esc automatisch ausgelöst wird.

a. Eine noch bessere Einstellung ist möglicherweise *poMainFormCenter*.

omponentenschablonen *R9*

Die oben gezeigte Kombination aus OK- und Cancel-Schalter wäre ein ideales Beispiel ür eine Komponentenschablone. Markieren Sie die beiden Schalter im Formular-Designer und rufen Sie aus dem lokalen Popup-Menü den Punkt DER OBJEKTABLAGE HINZUFÜGEN... auf, und Sie sparen sich in Zukunft mit zwei Mausklicks die Einstellung von sechs verschiedenen Properties (obige vier plus die beiden *Caption*-Properties mit Unterstreichung der ersten beiden Buchstaben!).

Modale Anzeige des Formulars

Die folgende Methode ist zur Bearbeitung des *OnClick*-Ereignisses des Schalters LARMZEIT EINSTELLEN... des Hauptformulars gedacht:

```
procedure TAlarmForm.Button2Click(Sender: TObject);
begin
  if AlarmEditDlg.ShowModal = mrOk then begin
    AlarmAktiviert := true;
    Alarmzeit := AlarmEditDlg.Edit1.Text;
  end;
end;
```

AlarmEditDlg ist der in der obigen Tabelle festgelegte Name für das von Kylix automatisch angelegte Formular-Objekt (deklariert in der Unit des Dialog-Formulars). Um in der Hauptformular-Unit darauf zugreifen zu können, muss die Unit des zweiten Formulars in die Unit des Hauptformulars eingebunden, also in der *uses*-Zeile aufgeführt werden. Sie brauchen das jedoch nicht manuell zu tun, denn Kylix fragt Sie beim Versuch, das Programm einfach so zu starten, ob es diese Änderung der *uses*-Klausel für Sie durchführen soll.

Die Methode *ShowModal* bewirkt, dass unser Zweitformular (welches wie alle Nicht-Hauptformulare standardmäßig unsichtbar ist, da *Visible=false*) modal angezeigt wird. Modal bedeutet, dass der Benutzer in dieser Anwendung nur mit diesem Fenster arbeiten kann. Alle anderen Fenster in derselben Anwendung werden erst dann wieder zugänglich, wenn er das modale Fenster schließt[1]. Und da wir oben die *ModalResult*-Properties der beiden Schalter entsprechend gesetzt haben, brauchen wir uns bei der Programmierung um dieses Schließen nicht zu kümmern (wir müssen also nicht *AlarmEditDlg.Close* aufrufen).

ShowModal gibt den *ModalResult*-Wert zurück, der zum Schließen des Fensters geführt hat (bzw. *mrNone*, falls wir das Fenster durch Aufruf der *Close*-Methode geschlossen ätten). Da die gerade eingestellte *Alarmzeit* nur bei Verwendung des OK-Schalters

[1] Es ist jedoch möglich, ein zweites modales Fenster zu öffnen. Dann tritt auch das erste modale Fenster in den Wartezustand.

geändert werden soll, muss das Ergebnis von *ShowModal* wie oben gezeigt abgefragt werden. Falls *ModalResult mrOk* zurückliefert, wird außerdem die Alarmfunktion eingeschaltet.

Überprüfen der Dialogeingaben *R30*

Nun soll das Programm die Eingabe des Benutzers noch überprüfen, falls dieser den Dialog mit OK schließt. Falls die Eingabe im Editierfeld nicht einer regelgemäßen Zeitangabe entspricht, wird eine Meldung angezeigt und der Benutzer erhält die Gelegenheit, seine Eingabe zu korrigieren oder ABBRUCH zu drücken.

Ein Fenster zu schließen ist einfach, hierfür gibt es die Methode *Close*. Um aber einen Weg zu finden, wie wir die automatische Schließung des Fensters verhindern können, müssen wir uns noch einmal mit dem *ModalResult*-Property befassen. Nicht nur die Aktionsschalter verfügen über ein solches Property, sondern auch das Formular selbst. Wird das *ModalResult*-Property *des Formulars* auf einen anderen Wert als *mrNone* gesetzt, führt das dazu, dass das Formular geschlossen wird und *ShowModal* das entsprechende Ergebnis zurückliefert.

Wenn ein Aktionsschalter betätigt wird, setzt die CLX den *ModalResult*-Wert des Formulars auf den des Schalters. Dies ist der tiefere Grund dafür, warum der Dialog automatisch geschlossen wird, wenn wir die *ModalResult*-Properties von OK und ABBRUCH setzen.

Um eine Eingabeüberprüfung zu implementieren, haben wir nun zwei Möglichkeiten:

- Wir setzen das *ModalResult*-Property des Formulars manuell zur Laufzeit. Die Einstellung des *ModalResult*-Properties der beiden Buttons zur Entwurfszeit wäre dann unnötig.
- Wir stellen das Property im Objektinspektor auf *mrOk* ein, verhindern aber, wenn die Eingabe des Benutzers nicht korrekt ist, dass auch das *ModalResult*-Property des Formulars gesetzt wird.

Für das Beispielprogramm wurde die zweite Möglichkeit gewählt. Drückt der Benutzer nun OK, so setzt die CLX das *ModalResult*-Property des Formulars auf *mrOk*. Die Schließung des Fensters findet aber erst statt, nachdem die *OnClick*-Methode aufgerufen wurde. Dies gibt uns Gelegenheit, innerhalb dieser Methode das *ModalResult*-Property im Fehlerfall wieder zurückzusetzen, so dass der Dialog doch nicht geschlossen wird:

```
procedure TAlarmEditDlg.Button1Click(Sender: TObject);
var
  TestAlarmTime : TDateTime;
begin
  try
```

```
      if Edit1.Text <> '' then // nur prüfen, wenn keine Leereingabe
        TestAlarmTime := StrToTime(Edit1.Text);
    except
      on E : EConvertError do begin
        ShowMessage('Ungültige Eingabe:' + E.Message);
        ModalResult := mrNone;
      end;
    end;
  end;
```

Weitere Aufgaben für den Dialog

Dies war erst die erste von vielen Methoden, die im weiteren Verlauf des Kapitels 1.8 zu *TAlarmEditDlg* hinzugefügt werden. Die Formularklasse eines Dialogs eignet sich ämlich hervorragend dazu, das Hauptformular zu entlasten:

indem es eine Eingabeüberprüfung selbst durchführt, wie oben gezeigt.

indem es dem Hauptformular eine aus mehreren Feldern bestehende Dialogeingabe in Form einer Object-Pascal-Datenstruktur zurückliefert.

- indem es dem Hauptformular erlaubt, auch mehrere Felder in einem Schritt durch Angabe einer solchen Datenstruktur zu initialisieren.

In der Programmversion *AlarmClock4* in Kapitel 1.8.4 wird das Beispielprogramm erstmals mit einer solchen Datenstruktur arbeiten.

1.8.2 Komponenten zur Programmsteuerung

So wie die Funktionsvielfalt moderner Software ständig ansteigt, erweitert sich auch das Arsenal der Bedienungselemente, durch die diese Funktionen gestartet werden önnen. Zwar basieren diese Elemente auf nur zwei Grundbausteinen, den Schaltern und Menüs, aber zum einen gibt es davon verschiedene Variationen, zum anderen wächst mit zunehmender Zahl der Schalter auch der Bedarf an Komponenten, durch die sich die Schalterflut besser organisieren lässt.

In der Kylix-Komponentenpalette befinden sich für Menüs, Schalter und Schalter-Organisation sowie für weitere Steuerungsaufgaben die folgenden Komponenten:



TMainMenu stellt eine Menüleiste dar, die Sie zum Hauptmenü eines Formulars machen können, indem Sie sie in dessen *Menu*-Property angeben. In Kapitel 1.4.8 wurde bereits beschrieben, wie Sie eine solche Menüleiste im Menüeditor von Kylix entwerfen können. Die nächste Version des Beispielprogramms demonstriert auch den praktischen Einsatz einer solchen Menüleiste.



Ein *TPopupMenu* ist ein Popup-Menü, wie Sie es auch in der Kylix-IDE überall durch Klicken der rechten Maustaste aufrufen können. Es wird ebenfalls im Menüeditor entworfen. Damit es zur Laufzeit vom Benutzer aufgerufen werden kann, müssen Sie lediglich festlegen, für welche Komponente des Formulars es gelten soll, wo der Benutzer also mit der rechten Maustaste klicken muss, um dieses Menü zu erhalten. Die Verbindung zwischen visueller Komponente und Popup-Menü geschieht über das *PopupMenu*-Property der visuellen Komponente. Mehr zum praktischen Einsatz von Popup-Menüs erfahren Sie in Kapitel 6.2.4.



TMenuItem ist die Klasse für einen einzelnen Menüpunkt. Sowohl *MainMenu*- als auch *PopupMenu*-Komponenten sind aus solchen *MenuItem*-Komponenten zusammengestellt. Die Properties von *TMenuItem* wurden bereits in Kapitel 1.4.8 beschrieben.



Die Klasse *TButton* stellt *den* Standard-Befehlsschalter zur Verfügung. Das bisherige Beispielprogramm ist nicht um das wichtigste Property (*Caption*) und das wichtigste Ereignis (*OnClick*) dieser Komponente herumgekommen. Und in Kapitel 1.8.1 sind uns auch die Properties *Default*, *ModalResult* und *Cancel* begegnet, die eng mit der Logik von modalen Dialogen verknüpft sind. Damit ist zu dieser Komponente bereits alles Wichtige gesagt.



Die Klasse *TBitBtn* auf der Seite ZUSÄTZLICH der Komponentenpalette ist von *TButton* abgeleitet und erweitert letztere durch die Möglichkeit, Bitmaps auf der Schaltfläche darzustellen. Hierzu verfügt sie über das Property *Glyph*, in das Sie eine Bitmap laden können, in der bis zu drei Bilder aneinander gereiht sind. Der Schalter stellt in jedem seiner drei möglichen Zustände eine dieser Bitmaps dar. Die drei Zustände sind der Normalzustand, der deaktivierte Zustand (*Enabled=False*) und der gedrückte Zustand. Für die weitere Gestaltung benötigen Sie auch die Properties *Layout*, *Margin*, *Spacing* und *NumGlyphs*, zu denen hier auf die Hilfedatei verwiesen sei. Wir benötigen *TBitBtn* nämlich in diesem Buch nur zur Anzeige der Standardbitmaps, aus denen Sie über das *Kind*-Property wählen können. Für selbst definierte Bitmaps verwenden die Beispielprogramme des Buchs Schalter der Klassen *TToolButton* und *TSpeedButton*.



TSpeedButton (Seite ZUSÄTZLICH) stellt die gleichen Properties für Bitmaps zur Verfügung, wie sie eben für *TBitBtn* genannt wurden. *TSpeedButton* kennt noch einen vierten Zustand, den Zustand, in dem der Schalter dauerhaft gedrückt ist (*Down=True*). Dieser gedrückte Zustand hängt mit der Fähigkeit dieser Schalter zusammen, in Gruppen aufzutreten, mit der sich Kapitel 6.2.1 genauer befassen wird.



TToolBar (Seite COMMON CONTROLS) stellt eine Werkzeug- bzw. Symbolleiste dar, die, im Gegensatz zu *TPanel*, ihre Kindelemente automatisch anordnet und die Icons effektiv in einer Bilderliste (*TImageList*) verwaltet.



TToolButton ist keine unabhängige Komponentenklasse, daher finden Sie sie auch nicht in der Komponentenpalette, sondern nur im lokalen Menü von *TToolBar*-Komponenten (NEUER SCHALTER). *TToolButton* bietet mehr Variationsmöglichkeiten als *TSpeedButton* und kann beispielsweise für DropDown-Schalter oder Textschalter verwendet werden.



TControlBar (Seite ZUSÄTZLICH) stellt eine Möglichkeit zur Verwaltung mehrerer Symbolleisten der *TToolBar*-Klasse dar. Der Benutzer kann diese Leisten zur Laufzeit des Programms verschieben und umordnen. Allgemein kann eine *ControlBar* auch beliebige andere Typen von Steuerelementen aufnehmen, allerdings ist sie für kleinere Elemente kaum geeignet, da sie für jedes Element ein eigenes, relativ aufwändig gestaltetes »Band« anlegt.



Auch die unscheinbare *TLabel*-Komponente besitzt eine wichtige Steuerungsfunktion: Wenn Sie durch Voranstellung eines »&« ein Zeichen in der Beschriftung des Labels unterstreichen, dient dieses Zeichen (eventuell in Verbindung mit Alt) als Tastenkürzel zum Anspringen eines Eingabeelementes. Damit dies funktioniert, müssen Sie ein *TLabel* über das *FocusControl*-Property mit einer Eingabekomponente verknüpfen. Im Beispielprogramm wurde dies bereits demonstriert (Kapitel 1.4.6, *Rücksichtnahme auf die Tastaturbedienung*).

Vom Schalter zum Menü

Nachdem das Entwerfen von Menüs mit Hilfe des Menüeditors bereits in Kapitel 1.4.8 zur Sprache gekommen ist, sei hier nur kurz demonstriert, wie das Beispielprogramm sich der Menüs bedient. In der Programmversion *AlarmClock2* (Abbildung 1.21) wurden zunächst die beiden *TButtons* im Dialogformular durch *TBitBtn*-Komponenten mit verändertem *Kind*-Property ersetzt, was aber nur äußerlich eine Änderung bedeutet. Dann wurden die beiden bisher als Schalter zugänglichen Aktionen des Hauptformulars ins Hauptmenü verlegt. Wir haben es hier also nicht mehr mit *TButton*-, sondern mit *TMenuItem*-Komponenten zu tun, die im Formular nicht durch separate Bausteine repräsentiert werden. Um ihre Properties im Objektinspektor editieren zu können, müssen Sie den entsprechenden Menüpunkt im Menüeditor auswählen.

Wir benötigen hier zunächst das *OnClick*-Ereignis, für das wir *keinen* neuen Code zu schreiben brauchen, denn wir können die Methoden wiederverwenden, die bisher beim Betätigen der Schalter Anwendung gefunden haben. Beide Methoden können Sie im Objektinspektor beim *OnClick*-Ereignis aus der aufklappbaren Liste auswählen. Die Methode *Button1Click* wird so auf den Schriftwahl-Menüpunkt umgeleitet, *Button2Click* auf den Menüpunkt zur Einstellung der Alarmzeit, alles durch ein paar Mausklicks. (Allerdings würde sich jetzt eine Umbenennung der Methoden anbieten, was jedoch mit der Maus alleine nur auf eine sehr umständliche Weise möglich wäre.)

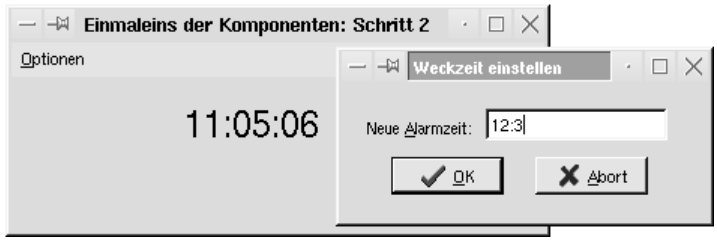


Abbildung 1.21: AlarmClock2 wurde mit einem Hauptmenü versehen.

Um das so gestaltete und mit Aktionen verknüpfte Menü auch anzuzeigen, muss außerdem das *Menu*-Property des Formulars auf die *TMainMenu*-Komponente gesetzt werden (dies geht durch bequemes Anklicken des Namens der *TMainMenu*-Komponente im Objektinspektor beim *Menu*-Property).

Hinweis: Ein weiterführendes Konzept zur Ausführung von Befehlen stellt das der Aktionslisten dar. Wenn Sie ein und dieselbe Aktion sowohl von einem Schalter als auch von einem Menüpunkt (oder irgendeiner anderen Steuerkomponente) aus starten wollen, bieten diese Aktionslisten eine erhebliche Vereinfachung (siehe Kapitel 4.6.4).

Markierte Menüpunkte

Das Beispielprogramm besitzt mit OPTIONEN | ALARMFUNKTION AKTIVIERT noch einen dritten Menüpunkt, mit dem die Alarmfunktion an- und ausgeschaltet werden kann. Eine Markierung neben dem Menüpunkt soll ähnlich einer CheckBox anzeigen, ob die Funktion gerade aktiviert oder deaktiviert ist. Im Gegensatz zu einem Markierungsschalter (*TCheckBox*) wird das *Checked*-Property eines Menüpunktes aber nicht automatisch umgeschaltet, wenn der Menüpunkt angeklickt wird. Daher muss dies manuell in der zum Menüpunkt gehörigen *OnClick*-Methode erledigt werden:

```
procedure TAlarmForm.Alarmaktiviert1Click(Sender: TObject);
begin
  Alarmaktiviert1.Checked := not Alarmaktiviert1.Checked;
end;
```

Da es in dieser Version des Beispielprogramms keine Variable *AlarmAktiviert* gibt, sondern bei Bedarf einfach das oben gesetzte *Checked*-Property abgefragt wird, sind keine weiteren Aktionen erforderlich.

1.8.3 Listenelemente und die Sicherung ihres Inhalts

Auch zur Darstellung von mehreren Datenelementen bietet die CLX eine Reihe von Alternativen:



TListBox ist quasi der Urahn aller Listboxen, eine Standard-Liste, die nicht nur für den Benutzer einfach handzuhaben ist, sondern auch für den Softwareentwickler – solange es nur darum geht, Texteinträge zu präsentieren und nicht etwa Grafiken.



TComboBox ist eine Kombination aus einem einzeiligen Eingabefeld und einer aufklappbaren Listbox (die Größe im aufgeklappten Zustand wird dann durch *DropDownCount* festgelegt). Bezüglich der Werte, die Sie im Editierfeld eingeben können, gibt es zwei Variationen, die Sie im Property *Style* auswählen können: Im Stil *csDropDown* können Sie Werte der Liste auswählen *und* beliebig im Editierfeld editieren oder ganz andere Werte dort eingeben. Mit dem Stil *csDropDownList* erhalten Sie eine Liste wie mit *csDropDown*, jedoch ist das Editierfeld nicht editierbar, sondern kann nur Werte enthalten, die auch in der Liste vorkommen (daher die Betonung des Namens »csDropDown*List*«). In seinen weiteren Eigenschaften (auch Ereignissen) ist *TComboBox* eine Kombination aus *TEdit* und *TListBox*. (Wenn Sie die *TComboBox* bereits von Delphi für Windows sehr gut kennen: Der dort bekannte Stil *csSimple* für ständig aufgeklappte Listen wird von der CLX nicht unterstützt.)



TCheckListBox befindet sich auf der Seite ZUSÄTZLICH der Komponentenpalette. Bei ihr ist jeder Listeneintrag mit einem Markierungsfeld versehen, wodurch dem Benutzer die Auswahl mehrerer Einträge besonders einfach gemacht wird. Auch der Programmierer ist wieder gut bedient, denn er kann den Markierungsstatus aller Einträge einfach über das Array-Property *Checked* abfragen bzw. über das *State*-Array, wenn auch der dritte Zustand *cbGrayed* erlaubt ist (dieser wird in Kapitel 1.8.4 im Zusammenhang mit der *TCheckBox*-Komponente erläutert).



TListView (Seite COMMON CONTROLS) stellt die moderne Variante einer Listbox dar. Jeder Texteintrag kann mit einem Icon versehen und in mehrere Spalten aufgeteilt werden. Wenn Sie eine Idee haben, wie Sie verschiedene Typen von Texteinträgen einer Listbox durch kleine grafische Symbole unterscheiden können, und wenn es vielleicht zu jedem Eintrag Detailinformationen gibt, die sich in einer Tabelle darstellen lassen, eignet sich diese Komponente hervorragend zur Realisierung. Ein erstes Anwendungsbeispiel finden Sie in Kapitel 1.8.5.

Wenn Sie bereits die *TListView*-Komponente von Delphi für Windows kennen, haben Sie hier die Chance, eine neue Gewohnheit zu erlernen: In Kylix enthält *TListView* nur die Darstellungsvarianten, die den *vsReport*- und *vsList*-Darstellungen unter Windows entsprechen. Für die Icon-Darstellungen ist unter Kylix eine eigene Komponente zuständig:



TIconView (Seite COMMON CONTROLS) stellt eine Liste von Einträgen als eine Sammlung von Icons dar. Der Text wird als Beschriftung der Icons angezeigt, die Icons sind frei verschiebbar. Auch für diese Komponente finden Sie in Kapitel 1.8.5 ein Anwendungsbeispiel.

Windows-Programmierer kennen diese Darstellung von Windows-ListView, bei dem diese Darstellung die Bezeichnung *vsIcon* bzw. *vsSmallIcon* hat. Auch bei *TIconView* können Sie zwischen verschiedenen Icon-Größen wechseln, indem Sie das *IconView* mit einer anderen Bilderliste verknüpfen.



TTreeView (Seite COMMON CONTROLS) ist sowohl von der internen Organisation als auch von seinem Äußeren her verwandt mit *TListView*, ist aber mit seinen Baumstrukturen eigentlich schon zu komplex, um noch als »Listenelement« gezählt werden zu können. Die Beispielprogramme dieses Buchs machen des Öfteren Gebrauch von dieser Komponente.



Für den Bereich der Datenbanken gibt es noch spezielle Variationen von Combo- und Listboxen: *TDBListBox*, *TDBComboBox*, *TDBLookupListBox* und *TDBLookupComboBox*, die an entsprechender Stelle in Kapitel 7 erläutert werden.

Bedienfunktionen für eine TListbox

Der Rest dieses Kapitels beschäftigt sich in der Praxis nur mit der Komponente *TListBox*, erwähnt jedoch auch, wie Sie die hier vorgestellten Techniken einfach auf die anderen Listenkomponenten übertragen können.

Das »Problem« der *TListBox*-Komponente ist, dass sie außer für die Darstellung der Listeneinträge nur für die Auswahl eines oder mehrerer Einträge automatisch sorgt, viele weitere Funktionen, die der Benutzer oft von einer Liste erwartet, müssen Sie von Hand hinzufügen. So möchte der Benutzer wahrscheinlich:

einen oder mehrere Einträge mit der Entf-Taste löschen

Einträge mit der Maus innerhalb der Liste verschieben können

- nicht mehrmals dieselben Eingaben machen: Wenn er den Inhalt der Liste selbst eingeben muss, so erwartet er sicher, dass das Programm diese Eingaben bis zum nächsten Programmstart nicht wieder vergisst.

Insbesondere diese drei Funktionen werden im Beispielprogramm *AlarmClock3* (Abbildung 1.22) demonstriert, das im Gegensatz zu *AlarmClock2* mehrere Alarmzeiten verwalten kann. Der schon aus *AlarmClock1/2* bekannte Dialog zur Eingabe der Zeit wird unverändert in das neue Projekt übernommen.

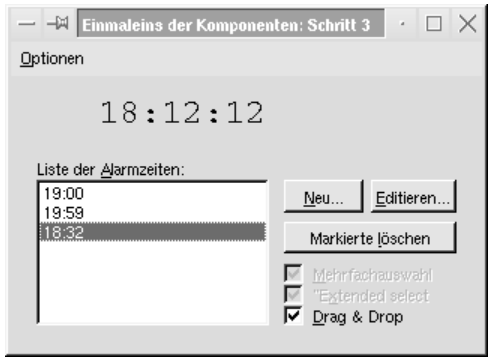


Abbildung 1.22: Die Listbox von AlarmClock3 bietet eine Reihe von Komfort-Funktionen.

TListBox-Inhalt

Der Inhalt einer Listbox wird unter Object Pascal als Stringliste dargestellt, die Ihnen über das Property *Items* zugänglich ist (intern verwendet *TListBox* hierfür den Typ *TStringList*, nach außen hin erscheint dieses Property aber im Gewand des abstrakten Typs *TStrings*). Sie können *Items* schon zur Entwurfszeit festlegen, wenn Sie im Objektinspektor den Stringlisten-Editor dieses Properties aufrufen; meist wird der Inhalt einer Listbox jedoch zur Laufzeit dynamisch aufgebaut. Im Beispielprogramm muss beispielsweise jedes Mal, wenn der Dialog zum Eingeben einer neuen Zeit mit OK bestätigt wurde, ein neues Element zur Liste hinzugefügt werden:

```
procedure TAlarmForm.AddButtonClick(Sender: TObject);
begin
  with AlarmEditDlg do
    if ShowModal = mrOk then
      ListBox1.Items.Append(AlarmEditDlg.Edit1.Text);
end;
```

Die Methode *Append* ist eine von vielen Methoden und Eigenschaften, die Sie auf entsprechende Weise für das *Items*-Property ansprechen können und die durch die schon erwähnte Klasse *TStrings* definiert werden. Die Zahl der Einträge erhalten Sie beispielsweise über *Items.Count*. Und um den Inhalt einer Listbox programmgesteuert abzufragen, können Sie das *Items*-Property wie ein Array-Property verwenden, unabhängig vom Beispielprogramm etwa so:

```
// Suchen des Eintrages '1984' (erster Eintrag an Position [0]!)
for i := 0 to ListBox.Items.Count - 1 do
  if ListBox.Items[i] = '1984' then Gefunden;
```

Eine allgemeine Besprechung der *TStrings*-Eigenschaften und -Methoden finden Sie in Kapitel 4.1.1.

Hinweis: Vielleicht ist Ihnen aufgefallen, dass die Methode *AddButtonClick* die Alarmzeit aus dem Dialog nicht mehr mit *StrToTime* in einen *TDateTime*-Wert konvertiert. Dies ist einer der angekündigten nebensächlichen Unterschiede zwischen den Programmversionen. Er liegt darin begründet, dass eine Listbox für das Speichern von Strings und nicht von *TDateTime*-Werten vorgesehen ist. Das Beispielprogramm stört sich nicht daran und speichert die Alarmzeiten eben als String und wandelt sie dann vor jedem Vergleich mit der aktuellen Zeit erneut in einen *TDateTime*-Wert um (wie in der letzten Programmversion wird aber im Dialog weiterhin geprüft, ob der Benutzer eine zulässige Zeitangabe gemacht hat).

TListBox-Auswahl und Fokus R58

Die wohl zweitwichtigste Funktion einer Listbox ist, dass der Benutzer einen Eintrag auswählen kann. Wenn Sie das *MultiSelect*-Property der Listbox einschalten, ist sogar

eine Mehrfachauswahl erlaubt. Im Beispielprogramm kann der Benutzer zur Laufzeit ber einen Schalter zwischen Einfach- und Mehrfachauswahl umschalten. Die aktuelle Auswahl in der Liste spielt in *AlarmClock3* für die Schalter EDITIEREN und MARKIERTE ÖSCHEN eine Rolle.

ber das Property *ExtendedSelect* können Sie zwischen zwei Arten der Mehrfachauswahl wählen. Bei Einstellung von *False* können Sie ohne zusätzliche Tasten mit der Maus aufeinander folgende Listeneinträge markieren (indem Sie die Maus bei gedrückter Taste bewegen), bei *True* müssen Sie ⇧ (für zusammenhängende Bereiche) und Strg (für mehrere einzelne Einträge) zu Hilfe nehmen, um mehrere Einträge zu markieren.

ür den Schalter EDITIEREN ist es egal, ob eine Mehrfachauswahl vorliegt, denn es soll immer nur ein Eintrag editiert werden können, und zwar der *fokussierte* Eintrag. Dies ist immer der Eintrag, der zuletzt ausgewählt wurde, er ist an einer gepunkteten Umrandung zu erkennen. Im Falle der Einfachauswahl ist der fokussierte gleichzeitig auch der gewählte Eintrag. In *TListBox* können Sie den fokussierten Eintrag über das Property *ItemIndex* abfragen und ändern. Drückt der Benutzer in *AlarmClock3* auf den DITIEREN-Schalter, wird die folgende Methode aufgerufen:

```
procedure TAlarmForm.EditButtonClick(Sender: TObject);
var
  Index: Integer;
begin
  // wenn kein Eintrag gewählt/fokussiert ist, ist
  // ItemIndex -1:
  if ListBox1.ItemIndex <> -1 then begin
    with AlarmEditDlg do begin
      Index := ListBox1.ItemIndex;
      Edit1.Text := ListBox1.Items[Index];
      if ShowModal = mrOk then
        ListBox1.Items[Index] := AlarmEditDlg.Edit1.Text;
    end;
  end else
    ShowMessage('Kein Eintrag ausgewählt.');
end;
```

Wird der Dialog mit OK beendet, überschreibt das Programm einfach den Text des gewählten Eintrags mit dem Inhalt des Editierfelds im Dialog.

Im Falle des Schalters MARKIERTE LÖSCHEN muss auch eine eventuelle Mehrfachauswahl berücksichtigt werden. Hier kommt ein neues *TListBox*-Property ins Spiel: Das Array-Property *Selected*, das wie das *Items*-Array indiziert wird und zu jedem Eintrag angibt, ob er gewählt ist. Bei eingeschalteter Mehrfachauswahl (*MultiSelect* = *True*) öscht die Ereignismethode für MARKIERTE LÖSCHEN alle Einträge, bei denen sie *Selected* = *True* vorfindet, im Falle der Einfachauswahl wird lediglich der durch *ItemIndex* angegebene aktuelle Eintrag gelöscht:

```
procedure TAlarmForm.DelButtonClick(Sender: TObject);
var
  i: Integer;
begin
(*  with ListBox1 do begin //  Der Klarheit zuliebe hier
                               keine with-Anweisung *)
  if ListBox1.MultiSelect then begin
    for i := ListBox1.Items.Count-1 downto 0 do
      if ListBox1.Selected[i] then ListBox1.Items.Delete(i);
  end else begin
    if ListBox1.ItemIndex <> -1 then
      ListBox1.Items.Delete(ListBox1.ItemIndex)
    else ShowMessage('Kein Eintrag gewählt.');
  end;
end;
```

Wichtig ist auch, dass die Löschmethode die Liste vom letzten Eintrag beginnend durchläuft. Eine einfache, nach oben zählende *for*-Schleife würde zu Fehlern führen, da sie nicht beachtet, dass sich die obere Grenze *(Items.Count*) während der Ausführung verringert und hintere Einträge automatisch in vordere Positionen aufrücken, wenn ein Eintrag gelöscht wird.

Eingaben in die Listbox *R80*

Die beiden bisher gezeigten Methoden wurden über separate Schalter gestartet und insbesondere der LÖSCHEN-Schalter überlässt es ganz der Listbox-Komponente, dass diese die Eingaben des Benutzers zur Auswahl von einem oder mehreren Einträgen richtig bearbeitet. Sie können jedoch auch selbst bei der Verarbeitung der Eingaben in die Listbox mitwirken:

- Eine besonders einfache Möglichkeit stellt die Bearbeitung des *OnClick*-Ereignisses dar, das nicht nur bei einem Klick mit der Maus ausgelöst wird, sondern auch bei jeder Änderung der Auswahl über die Tastatur. In einer Methode für dieses Ereignis könnten Sie beispielsweise Details zum gewählten Eintrag in einem Detailfenster anzeigen.
- Das Beispielprogramm bearbeitet das Ereignis *OnDblClick*, um den Editieren-Dialog aufzurufen, wenn der Benutzer auf einen Eintrag doppelklickt. Standardmäßig wird durch einen Doppelklick nämlich nur ein neuer Eintrag gewählt (wobei dies schon beim ersten der beiden Klicks geschieht). Um zusätzlich diesen Eintrag noch editieren zu können, braucht das Programm keine neue Methode, denn *OnDblClick* kann einfach mit der oben schon besprochenen *EditButtonClick*-Methode verknüpft werden.
- Auch für den Schalter MARKIERTE LÖSCHEN bietet es sich an, eine Alternative zur Verfügung zu stellen. Denn wozu gibt es auf der Tastatur die Taste Entf? Um Tastatureingaben in die Listbox zu bearbeiten (also Eingaben, die der Benutzer macht,

während die Listbox oder einer ihrer Einträge fokussiert ist), schreiben Sie eine Methode für *OnKeyDown* oder *OnKeyPress*. Im Beispiel muss nur abgefragt werden, ob es sich um die Löschtaste handelt, den Rest der Arbeit kann die bereits gezeigte Methode des Löschen-Schalters übernehmen:

```
procedure TAlarmForm.ListBox1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = Key_Delete then
    DelButtonClick(nil);
end;
```

Um Tastencodes wie *Key_Delete* verwenden zu können, müssen Sie die Unit *Qt* in Ihre Formular-Unit einbinden. Eine Auflistung aller Tastencodes finden Sie in der Online-Hilfe von Kylix unter dem Stichwort »Tastencodes« oder ganz authentisch und vollständig natürlich im Quelltext von `Qt.pas`. Zu weiteren Details bezüglich *OnKeyDown* und *OnKeyPress* sei hier auf Kapitel 3.3.2 verwiesen.

Speichern des ListBox-Inhaltes *R124*

Damit der Inhalt der Liste zwischen den Programmläufen nicht verloren geht, ruft *AlarmClock3* beim Start (*OnCreate*-Ereignis des Formulars) eine Methode namens *RestoList* und am Ende (*OnClose*-Ereignis des Formulars) eine *SaveList*-Methode auf, die im Folgenden kurz besprochen werden sollen.

Unter Linux gibt es verschiedene Orte, an denen Anwendungen ihre Einstellungen ablegen, wie etwa das Home-Verzeichnis des Benutzers, das Verzeichnis `/etc` oder das erzeichnis, in dem die Anwendung installiert ist. Für das Beispielprogramm wurde das Home-Verzeichnis des aktuellen Benutzers gewählt, das von Linux in der Umgebungsvariablen `HOME` angegeben wird. Diese Umgebungsvariable können Sie in Kylix mit der Funktion *GetEnvironmentVariable* abfragen.

Wenn das Verzeichnis feststeht, stellt sich die Frage nach dem Dateiformat. Menschenlesbare Textdateien sind ja unter Linux sehr beliebt zur Speicherung von Einstellungen und *TListBox.Items* besitzt sogar mit *SaveToFile* schon eine Methode, mit der die gesamten Listeneinträge in einer Textdatei gespeichert werden können. In einer späteren Programmversion, in der nicht mehr nur der Text der Einträge gespeichert werden soll, wird das nicht mehr ausreichen und wir werden eine Datei im INI-Format erzeugen, aber an dieser Stelle genügt es noch, *SaveToFile* aufzurufen:

```
const
  ConfFileName = 'AlarmDemo.conf.list';

procedure TAlarmForm.SaveList;
var
  HomeDir: String;
begin
```

```
  HomeDir := GetEnvironmentVariable('HOME');
  // Falls kein Heimatverzeichnis eingestellt, wird das
  // Wurzelverzeichnis verwendet:
  if HomeDir = '' then HomeDir := '/';
  ListBox1.Items.SaveToFile(IncludeTrailingPathDelimiter(HomeDir)+
    ConfFileName);
end;
```

Die Funktion *IncludeTrailingPathDelimiter* ist eine von Kylix neu eingeführte Funktion, die für plattformunabhängige Dateioperationen benötigt wird. Sie hängt – falls noch nicht vorhanden – ein Pfad-Trennzeichen (unter Linux »/«, unter Windows »\«) an den im Parameter angegebenen Pfad an und liefert als Ergebnis einen Pfad zurück, der auf jeden Fall mit dem Pfad-Trennzeichen endet. Da oben der Dateiname *ConfFileName* noch an den Pfad anschließt, wird in jedem Fall ein solches Pfad-Trennzeichen benötigt. Selbst wenn der in der `HOME`-Umgebungsvariablen angegebene Pfad standardmäßig mit einem »/« enden würde, würde man erst durch den Aufruf von *IncludeTrailingPathDelimiter* die Sicherheit erhalten, dass das Zeichen auch wirklich zur Stelle ist.

Das Wiederherstellen der Liste findet spiegelbildlich mit *LoadFromFile* statt. Als zusätzliche, bisher nicht verwendete Funktion wird noch mit *FileExists* abgefragt, ob überhaupt schon eine Konfigurationsdatei existiert. Ist das nicht der Fall (mindestens beim ersten Programmstart), wird auch *LoadFromFile* nicht aufgerufen (*LoadFormFile* würde sonst eine Exception erzeugen, da eine nicht existierende Datei als Parameter übergeben wird):

```
procedure TAlarmForm.RestoreList;
var
  HomeDir, FileName: String;
  s: String;
begin
  HomeDir := GetEnvironmentVariable('HOME');
  if HomeDir = '' then HomeDir := '/';
  FileName := IncludeTrailingPathDelimiter(HomeDir)+ConfFileName;
  if FileExists(FileName) then
    ListBox1.Items.LoadFromFile(FileName);
  if ListBox1.Items.Count > 0 then
    ListBox1.ItemIndex := 0; (* den ersten Eintrag der Liste auswählen *)
end;
```

Wie bereits erwähnt, wird *SaveList* beim *OnClose*-Ereignis, *RestoreList* im *OnCreate*-Ereignis des Formulars aufgerufen:

```
procedure TAlarmForm.FormCreate(Sender: TObject);
begin
  (* Alarmfunktion per Voreinstellung eingeschaltet: *)
  Alarmaktiviert1.Checked := True;
  (* Keinen Alarm mehr für gespeicherte Zeiten
     geben, die schon beim Start des Programms abgelaufen sind: *)
```

```
  LastAlarm := Time; // zu LastAlarm siehe IsTimeOver in der
                     // Einleitung zu Kapitel 1.8.
  TimerTimer(nil); (* Zeitanzeige initialisieren *)
  RestoreList; (* Vom letzten Programmlauf gespeicherte Alarme laden *)
end;

procedure TAlarmForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  SaveList;
end;
```

Datenobjekte in einer ListBox speichern

Zur Laufzeit des Programms werden also alle wichtigen Alarmdaten in einer *TListBox*-Komponente aufbewahrt. Da es sich bei jedem Alarm um einen einzelnen String handelt, bereitet dies auch noch keine Probleme. Angenommen aber, zu jedem Alarmereignis gäbe es neben der Alarmzeit noch eine Reihe weiterer Daten wie etwa einen Kommentar oder zusätzliche Angaben dazu, was bei Eintreten der Alarmzeit getan werden soll, dann wird es etwas schwieriger, auch diese Daten im Speicher zu halten.

TListBox ist aber schon auf diese Situation vorbereitet, denn dank der Klasse *TStrings* önnen Sie zu jedem Eintrag zusätzlich auch ein Objekt speichern – so befindet sich das Objekt zum i-ten Eintrag in *ListBox.Items.Objects[i]*. In Kapitel 1.8.4 wird das Programm auch diese Listbox-Eigenschaft nutzen.

Drag&Drop in einer ListBox *R50*

Eingebaut in die Listbox von *AlarmClock3* ist auch eine Drag&Drop-Funktion, die den Drag&Drop-Möglichkeiten einiger Listboxen der Kylix-IDE entspricht (z.B. der Liste der Palettenseiten im Dialog zur Konfiguration der Komponentenpalette und in einigen Property-Editoren wie etwa dem Felder-Editor oder dem Befehlslisten-Editor).

Da die Grundlagen des Drag&Drop ausführlich in Kapitel 6.8.3 beschrieben werden, wird sich dieses Kapitel auf die für *TListBox* relevanten Details beschränken.

Zunächst ist es empfehlenswert, die Listbox in den *MultiSelect*-Modus zu schalten, denn nur dann kann während des Drag&Drop unter der Maus ein dezentes Fokus-Rechteck gezeichnet werden, ohne dass dadurch die Markierung des aktuellen Eintrags geändert wird.

- Dann ist auch der Modus *ExtendedSelect* erforderlich, damit der Drag&Drop-Vorgang nicht zu einer Mehrfachauswahl führt.

Das Beispielprogramm trifft alle notwendigen Vorbereitungen, sobald Sie zur Laufzeit den Drag&Drop-Markierungsschalter aktivieren. Es schaltet eine eventuelle Sortierung der Liste aus und die genannten Optionen an, deaktiviert die Markierungsschalter

(*Enabled:=False*), damit der Benutzer gleich sieht, dass er diese Optionen bei Verwendung von Drag&Drop nicht verändern kann, und schaltet den Drag&Drop-Modus der CLX an (*DragMode:=dmAutomatic*, dies wird wieder in Kapitel 6.8.3 ausführlich besprochen):

```
procedure TAlarmForm.CBDragDropClick(Sender: TObject);
begin
  if CBDragDrop.Checked then begin
    // Drag&Drop-Checkbox wurde eingeschaltet
    SortCheck.Checked := False; SortCheck.Enabled := False;
    CBMultiSel.Checked := True; CBMultiSel.Enabled := False;
    CBExtendedSel.Checked := True; CBExtendedSel.Enabled := False;
    Listbox1.DragMode:=dmAutomatic;
  end else begin
    // Das Gegenteil, wenn DragDrop-Schalter abgeschaltet wird:
    SortCheck.Enabled := True;
    CBMultiSel.Enabled := True;
    CBExtendedSel.Enabled := CBMultiSel.Checked;
    Listbox1.dragMode := dmManual;
  end;
end;
```

Der eigentliche Drag&Drop-Vorgang basiert auf den Ereignissen *OnDragOver* und *OnDragDrop* (Kapitel 6.8.3!). Mit Hilfe der Methode *ItemAtPos* kann das Programm herausfinden, über welchem Eintrag sich die Maus gerade befindet (die Koordinaten *X* und *Y* werden von den *OnDrag*-Ereignissen übermittelt):

```
DragOverIndex := Listbox1.ItemAtPos(Point(X, Y), true);
```

Mit Hilfe der so gewonnenen Eintrags-Indizes ist es ein Leichtes, den gezogenen Eintrag an eine andere Stelle zu versetzen (*Items.Delete* für die alte Position, *Items.Insert* an der neuen Position, siehe CD).

Optimal ist diese Implementierung noch nicht, denn ein automatisches Scrollen findet nicht statt, falls Sie versuchen, einen Eintrag der Liste nach oben oder nach unten über die Listbox hinauszuziehen; aber wie Sie an der Kylix-IDE sehen können, ist diese Funktion nicht so wichtig. Wenn Sie einen Listeneintrag an eine nicht in der Liste sichtbare Stelle verschieben wollen, können Sie in der Kylix-IDE jedoch Schalter verwenden, die den Eintrag schrittweise nach oben oder nach unten befördern. Um solche Schalter könnte auch *AlarmClock3* sehr leicht nachgerüstet werden.

ListBox-Alternativen

Die folgende Tabelle dient gleichzeitig als Zusammenfassung der vorherigen Abschnitte und als Hinweis, wo Sie die verschiedenen gezeigten *TListBox*-Merkmale bei der modernen Variante der ListBox, dem *ListView*, finden. Zuerst noch zwei Hinweise auf die näheren Verwandten der ListBox:

TComboBox besitzen bis auf *MultiSelect*, *SelCount* und *ItemAtIndex* ebenfalls alle in der Tabelle für *TListBox* aufgelisteten Elemente.

- Wenn Sie von Delphi für Windows die Komponenten *TFileListBox* und *TDirectoryListBox* gewöhnt sind, werden Sie diese in Kylix vielleicht vermissen. Die Komponentenbibliothek dieses Buchs enthält jedoch Ersatzkomponenten (siehe R53 in Kapitel 8.8).

	TListBox	**TListView**
Einträge	Items	Items
Zahl der Einträge	ItemCount	Items.Count
i-ter Eintrag	Items[i]	Items[i]
Eintrag hinzufügen	Items.Append/Add	Items.Add
Eintrag&Objekt hinzufügen	Items.AddObject	Objekt muss extra zugewiesen werden.
Eintrag löschen	Items.Delete	Items.Delete
okussierter Index	ItemIndex	Items[ItemFocused].Index
okussierter Eintrag	Items[ItemIndex]	ItemFocused
Mehrfachauswahl	MultiSelect	MultiSelect
Zahl gewählter Einträge	SelCount	SelCount
Eintrag i gewählt?	Selected[i]	Items[i].Selected
Eintrag an (Maus-)Position	ItemAtIndex(x, y)	GetItemAt(x, y)
eignis bei Auswahländerung	OnChange	OnSelectItem
eignis bei Doppelklick	OnDblClick	OnDblClick / OnItemDoubleClick
eignisse bei Tastatureingabe	OnKeyDown, OnKeyPress,...	OnKeyDown, OnKeyPress,...
rknüpftes Objekt	Items.Objects[i]	Items[i].Data

1.8.4 Eingabekomponenten

In diesem Kapitel geht es um die zahlreichen Steuerelemente, in die der Benutzer Tastatur- oder Mauseingaben machen kann, die Sie im Programmcode bequem über Properties abfragen können; das Beispielprogramm zeigt auch die mögliche Verarbeitung dieser Eingaben in einer Kylix-Anwendung.

Zwischen Ein- und Ausgabekomponenten lässt sich keine perfekte Trennungslinie ziehen: Wenn Sie einen Mausklick als Eingabe ansehen, können Sie beispielsweise jede Komponente zu einer Eingabekomponente machen, indem Sie ihr *OnMouseDown*-Ereignis bearbeiten. Daher soll die folgende Aufstellung diejenigen Komponenten nennen, die *üblicherweise* zur Eingabe benutzt werden. Manchmal ist es auch wünschenswert, eine Komponente gleichzeitig zur Eingabe und zur Ausgabe zu verwenden. Dies

geschieht im TreeDesigner-Projekt z. B. in der Zeichenfläche (die hierfür zugrunde liegende Komponente *TPaintBox* wird in Kapitel 1.8.5 unter Ausgabekomponenten aufgezählt).

Soweit nicht anders angegeben, finden Sie die Komponenten auf der *Standard*-Seite der Komponentenpalette.



TCheckBox-Markierungsschalter haben zwei mögliche Zustände: nicht markiert und markiert (Property *Checked*). Optional können Sie noch einen dritten Zustand annehmen, den Sie nur über das Property *State* abfragen können. In diesem Zustand erscheint das Markierungsfeld des Schalters grau (nicht aber der Schalter selber, der erst bei *Enabled=False* grau dargestellt wird). In diesem dritten Zustand hat *State* den Wert *cbGrayed* und *Checked* ist *False*. Dieser Zustand ist nur erreichbar, wenn Sie das *AllowGrayed*-Property des Schalters auf *True* setzen.



TRadioButton-Komponenten können ebenfalls zwei Zustände annehmen und darüber hinaus von benachbarten Radioschaltern abhängig sein: In einer Gruppe von mehreren Radioschaltern kann immer nur einer eingeschaltet bzw. markiert sein. Schaltet der Benutzer einen anderen Schalter der Gruppe ein, wird der bisher markierte Schalter automatisch ausgeschaltet. Eine *Gruppe* bilden dabei alle Schalter, die sich innerhalb einer übergeordneten Komponente wie etwa einer *GroupBox*, einem *Panel* oder auch dem Formular selbst befinden. Besonders einfach lassen sich Gruppen von Radioschaltern mit der Komponente *TRadioGroup* erstellen.



Bei *TRadioGroup* brauchen Sie keine einzelnen Radioschalter in das Formular einzuzeichnen, denn diese Komponente erzeugt die Radioschalter automatisch. Sie müssen lediglich das Property *Items* mit den Texten der einzelnen Schalter füllen. Klicken Sie dazu im Objektinspektor auf den Schalter rechts neben *Items*. Es erscheint ein Stringlisten-Editor, in dem Sie die Texte für die einzelnen Radioschalter eingeben können.

TRadioGroup erzeugt für jeden String der Liste einen Radioschalter und verteilt sämtliche Schalter gleichmäßig in vertikaler Reihenfolge (bei Verwendung des Properties *Columns* auch auf mehrere Spalten). Dies ist eine besondere Erleichterung, wenn Sie nachträglich einen Schalter hinzufügen oder entfernen. Auch die Abfrage der eingeschalteten Option gestaltet sich mit *TRadioGroup* einfacher als mit einzel-

nen Radioschaltern, müssen Sie bei *TRadioGroup* doch nur das *ItemIndex*-Property auslesen (für den ersten Schalter nimmt es den Wert 0 an, ist kein Schalter gewählt, gibt es den Wert -1 an).



TEdit ist eine Komponente für ein einzeiliges Editierfeld, bei der Sie normalerweise nicht mehr als das *Text*-Property zur Abfrage des Textes und das *OnChange*-Ereignis zur eventuellen Überwachung von Änderungen abfragen müssen. Über die Vererbungshierarchie ist *TEdit* eng mit *TMemo* verwandt und stimmt auch in einigen weiteren Fähigkeiten mit diesem überein (z.B. bezüglich Zwischenablageoperationen und markiertem Text).



TMemo ist ein mehrzeiliges Editierfeld, das einen automatischen Zeilenumbruch durchführen (Property *WordWrap*) oder/und den Text mit Scrollbars scrollen kann (Property *Scrollbars*). Über das *Lines*-Property haben Sie Zugriff auf jede einzelne Zeile des Textes. *TMemo* hat eine ausführlichere Erläuterung in Kapitel 3.6.1 verdient.



TSpinEdit (Seite COMMON CONTROLS) ist ein um zwei kleine Schalter erweitertes Editierfeld zur Eingabe einer Zahl. Der Textbereich des Feldes ist auf Zahlen beschränkt und die Schalter dienen der schrittweisen Vergrößerung/Verkleinerung der Zahl. Programmtechnisch ist diese Zahl über das Property *Value* ansprechbar.



TMaskEdit (Seite ZUSÄTZLICH) ist ein weiterer Typ von Editierfeldern, der sich gegenüber *TEdit* durch das Property *EditMask* auszeichnet, in dem Sie festlegen, welchem Muster die Eingabe des Benutzers entsprechen muss. So bewirkt beispielsweise die Maske »90:00«, dass das Feld sich dem Benutzer mit der Angabe von »__:__« präsentiert. Der Benutzer kann nur Ziffern eingeben, darf aber die erste Ziffer weglassen. Das *EditMask*-Property verfügt sogar über einen eigenen Property-Editor, in dem Sie weitere Beispielmasken finden, Hilfe abrufen und sogar Ihre Maske testen können.



Ein *TrackBar* (Seite COMMON CONTROLS) ist ein Regler zum Einstellen eines Zahlenwertes, der im Property *Position* zur Verfügung steht. Sehr viele seiner exklusiven Properties befassen sich mit reinen Äußerlichkeiten (*Frequency*, *LineSize*, *Orientation*,

TickMarks, *TickStyle*), einige betreffen aber auch die Funktionsweise (*Min*, *Max*, *PageSize*). Allen Properties ist gemein, dass sie sich schnell durch ihren Namen oder die Online-Hilfe erklären lassen.



TStringGrid (Seite *Zusätzlich*) ist eine ganz in Object Pascal geschriebene (also nicht als Qt-Steuerelement existierende) editierbare Stringtabelle mit frei einstellbaren Zeilen- und Spaltengrößen. Ähnlich wie *TStringGrid* kann übrigens auch die später bei den Ausgabekomponenten aufgelistete *TDrawGrid*-Komponente zur Eingabe verwendet werden.

Neue Dialogelemente für das Beispielprogramm

Abbildung 1.23 zeigt die neue Version des Beispielprogramms, die sich von der letzten Version zur Entwurfszeit nur im Alarmeingabe-Dialog unterscheidet, in dem die Komponentenklassen *TEdit*, *TSpinEdit*, *TTrackBar*, *TMemo*, *TRadioButton* und *TCheckBox* zum Einsatz kommen. Die Änderungen an der Listbox fallen nur zur Laufzeit auf: Sie zeigt nun einige der zusätzlichen Attribute an, die im Dialog eingegeben wurden.

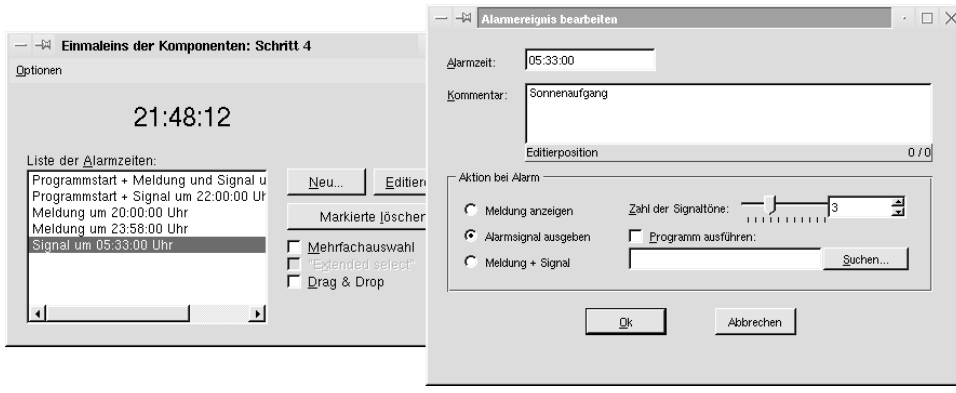


Abbildung 1.23: AlarmClock4 – die vierte Version des Beispielprogramms

Der Umgang mit den Eingabekomponenten im Programmcode braucht nicht groß erläutert zu werden – jede Komponentenklasse enthält ein Property, aus dem die Eingabedaten schnell abgefragt werden können, so wie in der folgenden Methode (die besagten Properties sind in Fettschrift wiedergegeben, Bezeichner des Programms, die später erläutert werden, in Kursivschrift):

```
procedure TAlarmEditDlg.GetAlarmData(Ziel: TAlarm);
begin
  with Ziel do begin
```

```
      Alarmzeit := StrToTime(DateTimeEdit.Text);
      Kommentar := KommentarMemo.Text;
      if RadioButton1.Checked then HinweisArt := haMeldung;
      if RadioButton2.Checked then HinweisArt := haSignal;
      if RadioButton3.Checked then HinweisArt := haBeides;
      AnzahlToene := AnzahlToeneEdit.Value;
      Programm := ProgrammCheckBox.Checked;
      ProgrammPfad := ProgrammPfadEdit.Text;
    end;
  end;
```

ür jede der verwendeten Klassen finden Sie außerdem ein Ereignis, das bei Änderung des Inhalts durch den Benutzer eintritt. Dieses heißt entweder *OnClick* bei Schaltern oder *OnChange* bei den anderen Komponentenklassen. Im Beispielprogramm wird dieses Ereignis für den *TrackBar* verwendet, um bei jeder Änderung des Reglers den neuen Wert in das nebenstehende SpinEdit-Feld zu schreiben:

```
procedure TAlarmEditDlg.AnzahlToeneTrackBarChange(Sender: TObject);
begin
  AnzahlToeneEdit.Value := AnzahlToeneTrackBar.Position;
end;
```

Hinweis: Zur Aktualisierung der Positionsangabe unter dem Memo-Feld bearbeitet *AlarmClock4* das *OnKeyUp*-Ereignis. Die aktuelle Eingabeposition finden Sie im *TMemo*-Property *CaretPos*. Weitere Details zu *TMemo* und *TEdit* finden Sie in Kapitel 3.6.1.

Eine Klasse zur Speicherung der Dialogeingaben

Der Sinn des Beispielprogramms liegt jedoch weniger in der Vorführung dieser einfachen Eingabekomponenten als darin,

- die Definition einer neuen Klasse für die Daten
- den Austausch der Daten zwischen dem Dialog und dem Hauptformular
- und die Speicherung der Daten in der ListBox und in einer Ini-Datei

zu demonstrieren.

Eine aufwändigere Lösung ist hier kaum zu vermeiden, da Kylix für die genannten einfachen Zwecke keine Automatismen zur Verfügung stellt. Wir können die Daten ja nicht einfach in den Komponenten des Dialogs stehen lassen, denn dort ist nur Platz ür die Angaben zu einem einzigen Alarm, das Programm muss aber in der Lage sein, mehrere Alarme speichern zu können. Und in der bisher zur Aufbewahrung der Daten verwendeten Listbox gibt es für jedes Element nur einen einfachen String, der den vielen Alarmparametern kaum gerecht werden kann.

Theoretisch wäre dieses Problem einfach dadurch zu lösen, die Alarmeinträge in einer Datenbank abzulegen und somit die Verwaltung der Daten an das Datenbanksystem zu delegieren. Wenn man sich aber überlegt, dass dann ein an sich winziges Utility wie dieses Beispielprogramm von umfangreichen Bibliotheken und Programmen abhängig wäre, wird schnell klar, dass auch ein Kylix-Programm kleine Datentabellen »von Hand« verwalten darf.

Hinweis: Diese Version des Beispielprogramms macht bereits von der Möglichkeit Gebrauch, eigene Properties zu definieren. Dieses Object-Pascal-Sprachmerkmal wird in Kapitel 2.2.5 beschrieben, wo Sie auch nachlesen sollten, falls hierzu an dieser Stelle noch Fragen offen bleiben sollten.

AlarmClock4, genauer die Unit des Dialogformulars, deklariert also eine neue Klasse, die alle zu einem Alarm gehörigen Daten umfasst und zusätzlich einige nützliche Hilfsmethoden bereitstellt. Eigentlich handelt es sich um eine Mischung zwischen Record und Klasse, denn *TAlarm* deklariert die einzelnen Datenelemente als öffentlich, so dass man sie wie die Elemente eines Records ansprechen kann. Normalerweise sollte eine Klasse keinen direkten Schreibzugriff auf ihre Daten von außen erlauben, doch die Erfüllung dieser Forderung würde das Programm noch aufwändiger machen.

```
TAlarm = class
  function GetEreignisText: String; // intern (d.h. von den unten
  function GetListText: String;   // deklarierten Methoden) verwendete
  function DataSize: Integer;     // Hilfsfunktionen
public
  (* Datenelemente - Zugriff wie bei einem Record *)
  Alarmzeit: TDateTime;
  Kommentar: String[255];
  HinweisArt: THinweisArt;
  AnzahlToene: Integer;
  Programm: Boolean;
  ProgrammPfad: String[255];
  (* Methoden und Properties: *)
  constructor Create; // Initialisierung eines neuen TAlarm-Objekts
  // Alle Daten in einem String angeben (für die Listbox im Hauptfenster):
  property ListText: String read GetListText;
  // zur Speicherung der Daten bei Programmende:
  procedure SaveToStream(S: TStream);
  procedure LoadFromStream(S: TStream);
  procedure UpdateListItem(item: TListItem); // in Kapitel 1.8.5 benötigt
  procedure AddToListView(lv: TListView); // in Kapitel 1.8.5 benötigt
end;
```

Initialisierung und Rückgabe der Dialogeingaben *R29*

Auf Grundlage der Klasse *TAlarm* kann die Dialogformular-Unit dem Hauptformular nun eine einfache Schnittstelle zur Verfügung stellen, die wir uns zunächst aus der Sicht des Hauptformulars ansehen. Wird der Schalter HINZUFÜGEN betätigt, so erzeugt dieses ein neues Objekt der Klasse *TAlarm*, indem es den *Create*-Konstruktor dieser Klasse aufruft. Dieser sorgt auch dafür, dass die einzelnen Datenelemente mit Vorgabewerten belegt werden:

```
procedure TAlarmForm.AddButtonClick(Sender: TObject);
var
  NewData: TAlarm;
begin
  NewData := TAlarm.Create;
```

Um den Dialog mit diesen Vorgabewerten zu initialisieren, genügt die Zuweisung des *TAlarm*-Objekts an ein selbst definiertes Property des Dialogformulars:

```
  AlarmEditDlg.AlarmData := NewData;
```

Der darauf folgende Aufruf des Dialogs findet weiterhin mit *ShowModal* statt. Im Erfolgsfall werden die geänderten Daten mit der Funktion *GetAlarmData* abgefragt:

```
  if AlarmEditDlg.ShowModal = mrOk then begin
    AlarmEditDlg.GetAlarmData(NewData);
```

GetAlarmData erwartet als Parameter ein bereits initialisiertes *TAlarm*-Objekt, in das es die Daten hineinschreiben kann. Nach diesem Aufruf gilt es, das neue Alarmereignis in der Listbox anzuzeigen.

Datenobjekte mit den Listeneinträgen verbinden *R51*

Hier kommt die Möglichkeit von *TListBox*, mit jedem String auch ein zugehöriges Objekt zu speichern, ins Spiel. Statt *Items.Add* oder *Items.Append* dient jetzt *AddObject* dazu, den Listentext (erster Parameter) zusammen mit dem Objekt (zweiter Parameter) zu speichern:

```
    ListBox1.Items.AddObject(NewData.ListText, NewData)
  end else
    NewData.Free; // das TAlarm-Objekt freigeben
end;
```

Dabei ist *ListText* eine Methode von *TAlarm*, die einige der Alarmparameter zu einem String zusammenfasst, den das Beispielprogramm in der Listbox statt der reinen Alarmzeit anzeigt.

Die Methode des Schalters EDITIEREN verläuft ähnlich. Es gibt vor allem eine interessante Zeile in der Methode: Wenn ein bereits vorhandener Eintrag der Liste editiert werden soll, muss der Alarm-Dialog mit den Werten dieses Eintrages initialisiert wer-

den, denn der Benutzer will ja die schon eingegebenen Werte ändern. Also muss das Programm das *TAlarm*-Objekt wiederfinden, das vorher einmal wie oben gezeigt mit *AddObject* zur Liste hinzugefügt wurde. Dafür bedient es sich des Properties *Items.Objects*, in dem zu jedem String der Liste an gleicher Position das zugehörige Objekt gespeichert ist:

```
ListBox1.Items.Objects[Listbox1.ItemIndex] // bezeichnet das Objekt für
                          // den gegenwärtig fokussierten Listeneintrag
```

Eine direkte Zuweisung dieses Objekts an eine *TAlarm*-Variable ist allerdings nicht möglich, da *Items.Objects[Index]* in der CLX als Typ *TObject* deklariert ist (die CLX kennt ja unseren Typ *TAlarm* noch nicht). Um dem Compiler mitzuteilen, dass es sich bei dem Objekt in Wirklichkeit um ein *TAlarm*-Objekt handelt, ist die folgende Typumwandlung notwendig.

```
ChangedAlarm := TAlarm(ListBox1.Items.Objects[Listbox1.ItemIndex]);
AlarmEditDlg.AlarmData := ChangedAlarm; // Initialisierung des Dialogs
  if AlarmEditDlg.ShowModal = mrOk then begin // Anzeige des Dialogs
```

Damit das Hauptformular wie gezeigt mit dem Dialogformular kommunizieren kann, muss Letzteres als Schnittstelle das Property *AlarmData* (mit der Schreibmethode *SetAlarmData*) und die Methode *GetAlarmData* implementieren. Da die beiden sich fast spiegelbildlich zueinander verhalten, genügt hier die oben bereits abgedruckte Methode *AlarmEditDlg.GetAlarmData*.

Hinweis: *GetAlarmData* hätte auch als Funktion geschrieben werden können, die ein *TAlarm*-Objekt zurückgibt, und diese Funktion hätte dann sogar als Lesemethode für das oben zum Schreiben verwendete *AlarmData*-Property dienen können, aber in diesen beiden Fällen wäre unklar gewesen, wer für die Initialisierung des *TAlarm*-Objektes zuständig gewesen wäre. Im Beispielprogramm gilt die Regel, dass ausschließlich das Hauptfenster die *TAlarm*-Objekte mit *Create* erzeugen und mit *Free* freigeben darf.

Zum Abschluss hier noch die vollständige Bearbeitungsmethode für den EDITIEREN-Schalter:

```
procedure TAlarmForm.EditButtonClick(Sender: TObject);
var
  Index: Integer;
  ChangedAlarm: TAlarm;
begin
  if ListBox1.ItemIndex <> -1 then begin
    Index := ListBox1.ItemIndex;
    ChangedAlarm := TAlarm(ListBox1.Items.Objects[Index]);
    AlarmEditDlg.AlarmData := ChangedAlarm;
    if AlarmEditDlg.ShowModal = mrOk then begin
      ListBox1.Items.Delete(Index);
```

```
        AlarmEditDlg.GetAlarmData(ChangedAlarm);
        ListBox1.Items.InsertObject(Index, ChangedAlarm.GetListText,
                                    ChangedAlarm);
        (* Die Markierung des Eintrages ging beim Löschen verloren
           und wird jetzt wiederhergestellt: *)
        ListBox1.ItemIndex := Index;
      end;
    end else
      ShowMessage('Kein Eintrag ausgewählt.');
  end;
```

Speicherung der Daten

ür die Speicherung der Daten genügt es jetzt nicht mehr, einfach den Inhalt der ListBox per *SaveToFile* in eine Datei zu schreiben, sondern es kommt darauf an, dass die einzelnen Felder jedes einzelnen *TAlarm*-Objekts berücksichtigt werden. *AlarmClock4* erzeugt nun zur Abwechslung einmal eine binäre Datei, die wieder unter dem in der Konstanten *ConfFileName* festgelegten Namen im Home-Verzeichnis gespeichert wird.

ür die Verwaltung der Datei auf der Seite des Programms ist die Klasse *TFileStream* zuständig, die zur Laufzeitbibliothek von Kylix gehört und der das Kapitel 4.3.1 sich unter anderem näher widmet. Als erstes Element dieser Datei wird eine »1« als eine Kennung für die Version des Dateiformats geschrieben, dann die Zahl der Alarmeinäge und schließlich werden alle Einträge der ListBox durchlaufen und die mit ihnen verknüpften *TAlarm*-Objekte ermittelt, für die dann *SaveToStream* aufgerufen wird:

```
procedure TAlarmForm.SaveList;
(* Alarmdaten schreiben *)
var
  HomeDir: String;
  Stream: TFileStream;
  i: Integer;
  HelpInt: Integer;
begin
  HomeDir := GetEnvironmentVariable('HOME');
  if HomeDir = '' then HomeDir := '/';
  Stream := TFileStream.Create(IncludeTrailingPathDelimiter(HomeDir)
                               + ConfFileName, fmCreate);
  HelpInt := 1;
  Stream.Write(HelpInt, sizeof(HelpInt)); // Format-Version
  HelpInt := ListBox1.Items.Count;
  Stream.Write(HelpInt, sizeof(HelpInt));
  for i := 0 to ListBox1.Items.Count-1 do
    TAlarm(ListBox1.Items.Objects[i]).SaveToStream(Stream);
  Stream.Free;
end;
```

Alarm.SaveToStream ist eine selbst definierte Methode, deren Code in der Unit von *TAlarm* zu finden ist. Hier werden die eigentlichen Daten geschrieben:

```
procedure TAlarm.SaveToStream(S: TStream);
begin
  S.Write(Alarmzeit, sizeof(Alarmzeit));
  WriteStringToStream(S, Kommentar);
  S.Write(HinweisArt, sizeof(HinweisArt));
  S.Write(AnzahlToene, sizeof(AnzahlToene));
  S.Write(Programm, sizeof(Programm));
  WriteStringToStream(S, ProgrammPfad);
end;
```

Freigabe der Daten

Mit der Verwendung einer neuen Klasse zur Speicherung der Alarmdaten geht natürlich auch die »moralische« Verpflichtung einher, die neu konstruierten Datenobjekte auch wieder freizugeben. In den oben gezeigten Zeilen findet zum Beispiel eine solche Freigabe statt, wenn der Benutzer den Dialog zum Einfügen eines neuen Elements abbricht. Das Programm verwirft das vorher neu erzeugte Objekt dann wieder mit *Free*. Auch wenn der Benutzer Objekte aus der Liste löscht, ruft *AlarmClock4* die Methode *Free* für die entsprechenden *TAlarm*-Objekte auf. Am Ende des Programms, also beim Ereignis *OnDestroy* des Hauptformulars, werden schließlich alle noch in der Liste befindlichen Alarm-Objekte freigegeben. Weitere Informationen zu *Free* und zur Freigabe von Objekten finden Sie in Kapitel 2.3.2.

1.8.5 Ausgabekomponenten

Mit den schon am Anfang von Kapitel 1.8.4 genannten Einschränkungen und Ausnahmen lassen sich die folgenden Komponenten als Ausgabekomponenten zusammenfassen:



Mit einer *TImage*-Komponente (Seite ZUSÄTZLICH) können Sie auf einfache Weise Grafiken in Ihre Formulare aufnehmen. Diese Bilder können schon zur Entwurfszeit festgelegt werden (Property *Image*), sie können aber auch zur Laufzeit aus Dateien oder der Zwischenablage geladen werden, und zwar über die in Kapitel 4.5.3 beschriebene *TPicture*-Schnittstelle (das Property *Image* hat den Typ *TPicture*).



TShape (Seite ZUSÄTZLICH): Wenn Sie in einem Formular eine einfache grafische Form wie einen Kreis oder ein Rechteck benötigen, könnten Sie diese Komponente benutzen. Möglicherweise ist jedoch der sinnvollste Einsatzzweck dieser Komponente, die Attribute der Pinsel und Stifte kennen zu lernen, mit denen Sie auf einem *Canvas*-Objekt zeichnen können (diese Attribute sind in *TShape* in den Properties *Pen* und *Brush* einstellbar und können schon zur Entwurfszeit getestet werden).



Die schon bekannte *TLabel*-Komponente sei hier der Vollständigkeit halber noch einmal mit aufgeführt, denn sie wird häufig nicht nur zur Beschriftung und Bereitstellung eines Tastenkürzels, sondern auch zur Ausgabe von Text verwendet (*Caption*-Property, eventuell auch mehrzeilig, wenn *WordWrap=True*).



TProgressBar (Seite COMMON CONTROLS) ist eine Klasse für Fortschrittsbalken, die von Installationsprozeduren und anderen zeitaufwändigen Vorgängen bekannt sind. Normalerweise haben Sie bei dieser Komponente nicht mehr zu tun, als ein paar Properties für Äußerlichkeiten einzustellen und zur Laufzeit das *Position*-Property gemäß dem zu beobachtenden Fortschritt zu setzen.



TPaintBox (Seite SYSTEM) definiert eine leere Zeichenfläche, die Ihnen im *OnPaint*-Ereignis die Gelegenheit gibt, Grafik über das *Canvas*-Property der Paintbox auszugeben. Die *PaintBox* ist immer dann ein guter Ersatz für die Zeichenfläche des gesamten Formulars, wenn die grafische Ausgabe nur in einem Teil dieses Formulars stattfinden soll. Der TreeDesigner in Kapitel 6 nutzt eine solche *TPaintBox* für seine Grafikausgabe.



TListView (Seite Common Controls) wurde bereits unter den Listenkomponenten in 1.8.3 vorgestellt. Die nächste Version des Beispielprogramms wird Ihnen diese Komponente genauer vorstellen.



Ebenfalls aus Kapitel 1.8.3 bekannt ist *TIconView*, ein Anwendungsbeispiel gibt das Beispielprogramm.



TTreeView (Seite Common Controls) ist eine Komponente zur Anzeige hierarchischer Strukturen nach der Art der Verzeichnisbäume in modernen Datei-Browsern wie dem KDE Dateimanager. Ihr ist das Kapitel 3.6.3 gewidmet.



THeaderControl (Seite Common Controls) stellt einen Tabellenkopf dar, dessen Spalten Sie über das Property *Sections* schon zur Entwurfszeit betiteln und über weitere Properties einstellen können. Zur Laufzeit kann der Benutzer die Spaltenbreiten

dann mit der Maus verändern und die Spaltenüberschriften als Schaltflächen bedienen (wodurch er die Ereignisse *OnSectionClick*, *OnSectionMoved* und *OnSectionResize* auslöst).

In der Komponente *TListView* ist ein Header bereits eingebaut, so dass Sie diese Komponente wahrscheinlich selten einzeln benötigen. Diese Komponente kommt im vorliegenden Buch und in den zugehörigen Beispielprogrammen nicht einzeln (d.h. außerhalb eines ListViews) vor.



Bei *TStatusBar* (Seite COMMON CONTROLS) handelt sich um eine typische Statuszeile, die in einzelne Abschnitte aufgeteilt werden kann – so wie die Statuszeile des Kylix-Editors. Zwar können Sie Statuszeilen auch einfach mit gewöhnlichen Panels aufbauen, *TStatusBar* bietet jedoch erheblich mehr Komfort bei der Unterteilung der Leiste (siehe den Editor des Properties *TStatusBar.Panels*).



TImageList (Seite COMMON CONTROLS) ist eine nicht-visuelle Komponente, die in *ToolBars*, *ListViews* und *TreeViews* zur Verwaltung der Icons verwendet wird. Die nächste Version des Beispielprogramms wird eine solche Komponente für sein ListView verwenden.



TLCDNumber (Seite COMMON CONTROLS) kapselt die von Qt bereitgestellte Komponente zur Anzeige von Zahlen in LCD-Optik (mit 7 Segmenten pro Ziffer). Sie können dem *Value*-Property ganze Zahlen und Fließkommazahlen zuweisen, in jedem Fall ist die Anzeige auf eine feste Zahl von Stellen begrenzt, die Sie im Property *Digits* angeben. Das Property *SegmentStyle* gibt Ihnen drei optische Darstellungsvarianten für die Ziffern, das Property *Mode* vier verschiedene Zahlensysteme (dezimal, binär, hexadezimal, oktal) zur Auswahl.



TTextViewer (Seite COMMON CONTROLS) ist ein Anzeigeelement für HTML-Dokumente, das ebenfalls von Qt zur Verfügung gestellt und von der CLX gekapselt wird. Neben der anzuzeigenden Datei (Property *FileName*) können Sie wie in einem richtigen Browser die Farbe von normalem Text und der Hyperlinks (*TextColor*, *LinkColor*) einstellen, die Unterstreichung der Links lässt sich durch *UnderlineLink* auch abschalten.



TTextBrowser (Seite COMMON CONTROLS) stellt eine Erweiterung von *TTextViewer* dar, die auch Fähigkeiten zum Browsen einschließt. Während Hyperlinks im Text*Viewer* lediglich hervorgehoben dargestellt werden können, reagieren sie im Text*Browser* auch auf Mausklicks, der Browser lädt dann die mit den Links verbundenen Dokumente. Auch die Erweiterungen des Browsers gegenüber des Viewers entstammen bereits der Qt-Bibliothek.



TDrawGrid (Seite ZUSÄTZLICH) stellt die gleiche Tabellenstruktur zur Verfügung wie *TStringGrid*, überlässt es aber Ihrer Verantwortung, die Zellen mit Inhalt zu füllen. *TDrawGrid* und *TStringGrid* sind über die gemeinsame Elternklasse *TCustomGrid* eng verwandt. Ein Beispiel zur Grafikausgabe in einem *TDrawGrid* finden Sie am Ende von Kapitel 4.4.3.

Die ListView-Version des Beispielprogramms

Die letzte Version des Beispielprogramms, *AlarmClock5* (Abbildung 1.24), verwendet vier der genannten Ausgabekomponenten: *TListView*, *TIconView*, *TStatusBar* und *TProgressBar*: Das ListView ersetzt die bisher verwendete Listbox und bietet eine mehrspaltige Darstellung verschiedener Alarmdaten sowie die Darstellung von Symbolen für verschiedene Alarmtypen. Das IconView nimmt die gleiche Fläche wie das ListView ein und stellt eine rein symbolische Darstellung der Alarmdaten (nur Icons + Titel) zur Verfügung. Die Statuszeile dient dazu, längere Hinweistexte zu der Komponente anzugeben, über der sich die Maus gerade befindet. Der Fortschrittsbalken soll anzeigen, wie viel der Zeit zwischen dem letzten und dem nächsten Alarmereignis bereits abgelaufen ist.

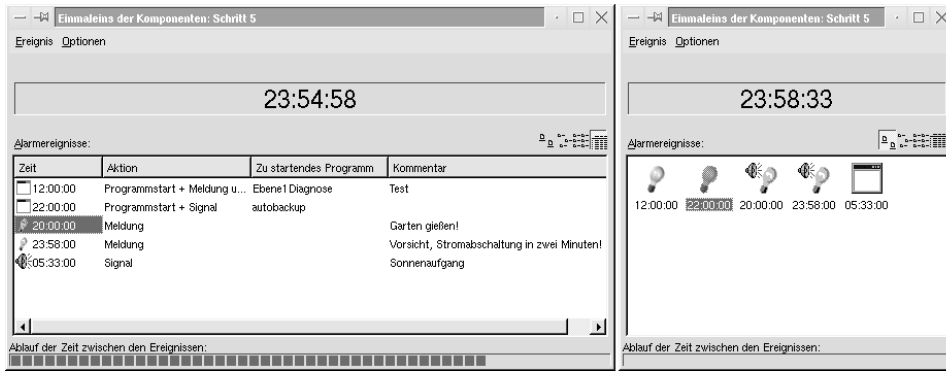


Abbildung 1.24: Zwei der vier möglichen Ansichten der abschließenden Programmversion

TStatusBar

Um das Aussehen einer Statuszeile festzulegen, müssen Sie im Wesentlichen die Properties *SizeGrip* und *Panels* setzen. Wenn *SizeGrip* eingeschaltet ist, wird die rechte untere Ecke mit einer Hervorhebung versehen, die an eine Vergrößerbarkeit des Fensters erinnert, selbst wenn die Statuszeile gar nicht zur Größeneinstellung dient. *Panels* ist ein strukturiertes Property, das Sie über einen Listeneditor editieren, in dem Sie die Bereiche definieren können, in welche die Statuszeile unterteilt werden soll. Wenn Sie Bereiche definieren, können Sie den Text jedes Bereichs über das Property *TStatusBar.Panels[x].Text* setzen.

Um einen möglichst breiten Platz zur Anzeige von Hinweisen zu erhalten, definiert *AlarmClock5 keine* Bereiche, sondern verwendet eine aus einem einzigen Bereich bestehende Statuszeile, deren Text sich über das Property *SimpleText* setzen lässt. Um dieses wirksam zu machen, muss allerdings vorher noch das *SimplePanel*-Property auf *True* gesetzt werden.

Ein Beispiel für die Anwendung der Statuszeile: Wenn der Benutzer die Alarmfunktion manuell deaktiviert, zeigt das Programm eine Rückmeldung in der Statuszeile an:

```
StatusBar1.SimpleText := 'Alarmfunktion deaktiviert.';
```

Hinweistexte in der Statuszeile anzeigen R59

Der Haupteinsatzzweck der Statuszeile in *AlarmClock5* liegt jedoch im Anzeigen der Hinweise für die einzelnen Komponenten. Es gibt sogar ein Ereignis, das jedes Mal auftritt, wenn die Maus über eine beliebige Komponente Ihrer Anwendung bewegt wird. Dieses Ereignis heißt *OnHint*, befindet sich aber im *Application*-Objekt, so dass Sie das Ereignis nicht über den Objektinspektor mit einer Methode verknüpfen können. In *AlarmClock5* wird diese Verknüpfung stattdessen in der *OnCreate*-Methode programmgesteuert durchgeführt:

```
procedure TAlarmForm.FormCreate(Sender: TObject);
begin
  ...
  Application.OnHint := ApplicationHint;
end;
```

Die Methode *ApplicationHint* muss nun lediglich den Hinweistext der aktuellen Komponente, den die CLX im Property *Application.Hint* bereitstellt, in die Statuszeile schreiben:

```
procedure TAlarmForm.ApplicationHint(Sender: TObject);
begin
  StatusBar1.SimpleText := Application.Hint;
end;
```

Hinweisfenster allgemein

Normalerweise zeigt die CLX die Hinweistexte automatisch in den bekannten gelben Hinweisfenstern an, wenn Sie das *ShowHint*-Property des Formulars auf *True* setzen. Alle Komponenten dieses Formulars »erben« dann standardmäßig dieses *ShowHint*-erhalten. (Diese Vererbung können Sie bei jeder Komponente unterbinden, indem Sie ihr *ParentShowHint*-Property auf *False* setzen.) In diesem Beispielprogramm wurde *ShowHint* abgeschaltet, damit die Hinweise nicht gleich doppelt angezeigt werden.

Optional können Sie im *Hint*-Property sogar zwei Versionen des Hinweistextes angeben. Sie trennen beide Versionen durch das Zeichen »|« (ANSI-Code 124, üblicherweise mit »<« und »>« auf einer Taste). Kylix nimmt an, dass Sie vor diesem Zeichen eine kurze Version des Hinweistextes angeben, die sich sehr gut für die Hinweisfenster eignet, und danach eine ausführlichere Version, die beispielsweise in einer Statuszeile angezeigt werden könnte und die in *AlarmClock5* über das Property *Application.Hint* abgerufen wird. Geben Sie nur eine Hinweis-Version ein, so verwendet die CLX diese sowohl für die automatischen Hinweisfenster als auch für *Application.Hint*.

Hinweis: Sie können das *Hint*-Property auch dynamisch zur Laufzeit ändern. So können Sie beispielsweise im *OnMouseMove*-Ereignis einer Zeichenfläche das *Hint*-Property auf die aktuelle Mausposition setzen, damit in einer Statuszeile immer die aktuelle Mausposition angezeigt wird. Die TreeDesigner-Beispielanwendung aus Kapitel 6 gibt mit ihrer Statuszeile ein praktisches Beispiel hierfür.

TProgressBar

Das Beispielprogramm übernimmt das voreingestellte Aussehen des Fortschrittsbalkens. Veränderungen an Properties werden daher nur zur Laufzeit vorgenommen. Der durch den Balken zu beobachtende »Fortschritt« ist der Verlauf der Zeit zwischen zwei Alarmereignissen. Der kleinste vom Programm messbare Fortschritt ist damit die Zeit zwischen zwei *OnTimer*-Ereignissen. So verwendet das Programm auch die *OnTimer*-Methode, um den Balken zu aktualisieren. Der aktuelle Fortschritt wird im Property *Position* angegeben und versteht sich relativ zu Start- und Endwert, die in den Properties *Min* und *Max* angegeben werden können. Da im Beispiel *Min* und *Max* nicht geändert wurden, stehen die Werte zwischen 0 und 100 zur Verfügung, um den Fortschritt darzustellen. Die *OnTimer*-Methode muss also die bisher abgelaufene Zeit lediglich in Prozent umrechnen, den dabei herauskommenden Fließkommawert mit der Standardfunktion *trunc* in eine ganze Zahl runden und dem *Position*-Property zuweisen:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
var
  NextAlarm: TDateTime;
begin
  ...
```

```
NextAlarm:=GetNextAlarm;
if NextAlarm > -0.1 then
  ProgressBar1.Position :=
    trunc ( (Time-LastAlarm) /
            (NextAlarm-LastAlarm) * 100
          )
else ProgressBar1.Position:=0;
```

TListView und TIconView

TListView verfügt von Haus aus über zwei Darstellungsmodi, die Sie über das Property *ViewStyle* auswählen können:

- Im Stil *vsRecord* werden mehrere Spalten von Informationen angezeigt, in der ersten Spalte wird für jeden Eintrag ein Symbol dargestellt. Diese Spalten werden nach Art der Komponente *THeaderControl* beschriftet. Im Dateimanager von KDE 2 taucht diese Ansichtsart unter dem Namen »Detailliert« auf und auch vom Windows-Explorer ist diese Darstellungsweise bereits bekannt. Selbst in der Kylix-IDE gibt es Anwendungsbeispiele für diese Darstellungsweise: Der DATEI-ÖFFNEN-Dialog lässt sich durch einen kleinen Schalter oben rechts in diese Darstellungsweise bringen und im Dialog DATEI | NEU können Sie aus dem Kontextmenü des Symbolbereichs den Punkt DETAILS ANZEIGEN auswählen, um eine ähnliche Tabelle zu erhalten.
- Bei der Einstellung *vsList* werden nur noch die Symbole und deren Beschriftung angezeigt, die weiteren vom Stil *vsRecord* bekannten Spalten fallen weg. Dafür bildet das ListView nun mehrere Spalten von Symbolen (je nach vorhandenem Platz), so wie sie standardmäßig in den Dateiwahldialogen für die Liste der Dateien angezeigt werden.

Wenn wir die Komponenten aus der Sicht des Benutzers betrachten, fehlen dem ListView noch einige Darstellungsarten. Sowohl der Windows-Explorer als auch der Konqueror von KDE 2 bieten neben den beiden erwähnten Ansichtsarten noch weitere, für deren Nachbildung man unter Kylix andere Komponenten benötigt. Der Konqueror kann seine Dateilisten auch als TreeView (die entsprechende Kylix-Komponente ist *TTreeView*) oder IconView darstellen. Von der ListView-Komponente von Delphi für Windows sind zwei Darstellungensmodi bekannt, die dem IconView entsprechen, und zwar entweder mit großen oder mit kleinen Icons.

Das Beispielprogramm unterstützt auch die beiden Darstellungsmodi des IconViews, jedoch würde es den Rahmen dieses Kapitels sprengen, wenn dies alles hier erläutert werden sollte. Die folgenden Absätze befassen sich daher nur mit dem ListView und überlassen es dem Kapitel 3.6.2, die Kombination von ListView und IconView in *AlarmClock5* näher zu untersuchen.

TListView im Beispielprogramm

In Kapitel 3.6.2 werden Sie eine ausführlichere Besprechung dieser Komponente finden, an dieser Stelle geht es um einen ersten Einstieg in die Praxis mit dieser Komponente, wobei noch einige Merkmale von *TListView* außer Acht gelassen werden.

or dem Schreiben von Code müssen die folgenden Vorbereitungen getroffen werden, um ein ListView zu erhalten, das wie im Beispielprogramm funktioniert:

Zuerst gilt es, ein paar passende Bildsymbole zu finden. Wenn diese nicht aus der MDI-Schablone von Kylix (erzeugen Sie mit DATEI | NEU ein neues MDI-Projekt von der Dialogseite PROJEKTE) oder anderen frei verwendbaren Vorlagen wie etwa denen aus dem Bilderordner von Kylix (`kylix/images`) übernommen werden können, können Sie mit einem Bildeditor eigene Bitmaps herstellen. Da Icons für die Funktion eines ListViews nicht unbedingt erforderlich sind, können Sie diesen Schritt natürlich auch überspringen oder zumindest auf später verschieben.

Diese Icons werden dann unter Kylix in einer *TImageList*-Komponente zusammengefasst. Nachdem Sie eine solche Komponente in das Formular eingefügt haben, rufen Sie mit einem Doppelklick darauf einen Dialog auf, in dem Sie die Bilddateien laden können.

Nun können Sie die *TListView*-Komponente in das Formular einfügen und über das Property *Images* mit der Bilderliste verknüpfen.

Auch die Tabellenspalten für die Report-Ansicht lassen sich bequem schon zur Entwurfszeit einfügen. Rufen Sie dazu im Objektinspektor beim *Columns*-Property des ListViews den Spalteneditor auf. Für das Beispielprogramm wurden die vier Spalten *Zeit*, *Aktion*, *Zu startendes Programm* und *Kommentar* erzeugt. Für jede Spalte lassen sich neben der Beschriftung einige weitere Properties einstellen, was im Beispielprogramm zur Einstellung der Spaltenbreite genutzt wurde.

- Um die Tabellenspalten schon zur Entwurfszeit überprüfen zu können, können Sie das ListView-Property *ViewStyle* auf *vsReport* setzen.

Elemente hinzufügen

Die Verwaltung der einzelnen Listeneinträge läuft in einem ListView grundlegend verschieden von der Verwaltung der Strings in einer ListBox ab. Im ListView wird jeder Eintrag durch ein eigenes Objekt repräsentiert, in dem alle wichtigen Daten zusammengefasst sind, die in einer Listbox noch vereinzelt vorliegen (so müssen Sie sich in einer ListBox beispielsweise die zum *i*-ten Eintrag gehörenden Daten aus folgenden Quellen zusammensuchen: *Selected[i]*, *Items[i]* und *Items.Objects[i]*, in *TListView* finden Sie sie als *Items[i].Selected*, *Items[i].Caption* und *Items[i].Data*).

Ähnlich wie bei *TListBox.Items.Append* erzeugen Sie mit *TListView.Items.Add* einen neuen Eintrag. Der wesentliche Unterschied ist, dass *Add* noch nicht den Inhalt des Eintrages festlegt, sondern nur ein *TListItem*-Objekt zurückliefert:

```
// siehe Beispielprogramm: Methode TAlarm.AddToListView
var
  NewItem: TListItem;
begin
  NewItem := ListView.Items.Add;
```

Dieses *TListItem*-Objekt verfügt nun seinerseits wieder über eine Reihe von Properties, über die Sie den Inhalt des Eintrags festlegen können, der nicht nur aus dem Beschriftungstext besteht, sondern auch aus dem Bildindex (die Bilderlisten werden ja schon in *TListView*-Properties angegeben, müssen hier also nicht nochmals gesetzt werden) und den Informationen, die in den Spalten der Report-Ansicht angezeigt werden sollen:

```
// siehe Beispielprogramm: Methode TAlarm.UpdateListItem
  NewItem.Caption := TimeToStr(Alarmzeit); // Text des Eintrages
  if Programm then // Index des Bildsymbols
    item.ImageIndex := 0
  else item.ImageIndex := 1;
  NewItem.SubItems.Clear; // Text für die einzelnen Spalten
  NewItem.SubItems.Add(GetEreignisText);
  NewItem.SubItems.Add(ProgrammPfad);
  NewItem.SubItems.Add(Kommentar);
```

Um den Zusammenhang zum Beispielprogramm darzustellen, hier noch ein Ausschnitt aus der *OnClick*-Methode für den Schalter EDITIEREN:

```
NewData := TAlarm.Create;
...
if AlarmEditDlg.ShowModal = mrOk then begin
  AlarmEditDlg.GetAlarmData(NewData);
  NewData.AddToListView(ListView1)
end else
  NewData.Free;
```

Um diese Zeilen vollständig verstehen zu können, sollten Sie die *TAlarm*-Klasse kennen, die in Kapitel 1.8.4 eingeführt wurde. Sie eignet sich hervorragend, um die Initialisierung eines Listeneintrages durchzuführen. *TAlarm.UpdateListItem* schreibt alle Alarmdaten in einen bestehenden *TListItem* und *TAlarm.AddToListView* erzeugt zusätzlich einen neuen Eintrag. Die entscheidenden Ausschnitte aus diesen beiden Methoden wurden oben abgedruckt.

Verknüpfung der Einträge mit Programmobjekten

Wie in der letzten Programmversion sollen auch in *AlarmClock5* die einzelnen *TAlarm*-Objekte bei den zugehörigen Listeneinträgen gespeichert werden, damit sich die Daten später wieder so leicht in die Dialogbox zum Editieren kopieren lassen wie in Kapitel

1.8.4 gezeigt. Zu diesem Zweck kommt noch eine weitere Zuweisung an das *TListItem*-Objekt hinzu:

```
// siehe Beispielprogramm: Methode TAlarm.AddToListView
  NewItem.Data := self;
```

Diese Verknüpfung kann unter anderem dazu verwendet werden, um in der *OnTimer*-Methode die einzelnen Alarmdaten abzufragen:

```
with ListView1 do begin
  // Eine neu erreichte Alarmzeit suchen:
  for i := 0 to Items.Count-1 do begin
    TestAlarm := TAlarm(Items[i].Data).Alarmzeit;
    if IsTimeOver(TestAlarm) then begin
      LastAlarm := TestAlarm;
      AlarmAction(TAlarm(Items[i].Data));
    end;
  end;
```

Die Typumwandlung *TAlarm(Items[i].Data)* ist ebenfalls schon aus *AlarmClock4* bekannt. Auch die Methoden zum Editieren eines Elements und zum Speichern der Daten in der Konfigurationsdatei müssen bei einer Umstellung von *TListBox* auf *TListView* nur geringfügig angepasst werden. Wenn Sie eine solche Umstellung in einer Ihrer Anwendungen durchführen wollen, können Sie auch die Tabelle in Kapitel 1.8.3 zu Hilfe nehmen.

Hinweis: Während das *Data*-Property der ListView-Einträge schon der Kompatibilität mit der VCL wegen erforderlich ist, bietet die CLX noch eine zweite Möglichkeit, Datenobjekte im ListView zu speichern. Und zwar können Sie eine eigene Klasse von *TListItemClass* ableiten und ListView mitteilen, dass es von nun anstatt der vordefinierten *TListItemClass* Ihre neue Klasse verwenden soll. In Kapitel 3.6.2 wird gezeigt, wie Sie eigene Klassen in einem *IconView* verwenden können. In einem ListView funktioniert dies völlig analog dazu.

Editieren der Einträge *R55*

Ein ListView kann auch als Eingabekomponente verwendet werden. Solange Sie sein *ReadOnly*-Property nicht auf *True* setzen, wird die Editierfunktion automatisch von der Komponente bereitgestellt: Um einen Eintrag zu editieren, muss der Benutzer ihn mit der rechten Maustaste anklicken.

Hiermit ist es aber nicht getan, denn die Beschriftung der Elemente im ListView des Beispielprogramms gibt ja die Alarmzeit an, die jedoch eigentlich an einer anderen Stelle gespeichert ist, und zwar in einem *TAlarm*-Objekt. Damit das Editieren im ListView Sinn macht, muss das Programm also nach jeder Umbenennung auch die mit dem umbenannten Eintrag gespeicherte Alarmzeit aktualisieren.

Natürlich gibt es für das Umbenennen eines Eintrages ein spezielles Ereignis, es heißt *OnEdited* und wird wie folgt bearbeitet:

```
procedure TAlarmForm.ListView1Edited(Sender: TObject; Item: TListItem;
  var S: WideString);
begin
  try
    TAlarm(Item.Data).Alarmzeit := StrToTime(S);
  except
    on E: EConvertError do
      S := TimeToStr(TAlarm(Item.Data)).Alarmzeit;
  end;
end;
```

Der editierte Eintrag muss nicht aus der *Items*-Liste herausgesucht werden, denn er wird der Methode direkt im Parameter *Item* übergeben. *S* enthält nahe liegenderweise den neuen Text für den Eintrag. Er ist als *var*-Parameter gehalten, so dass das Programm ihn noch einmal verändern kann.

In diesem Fall wird einfach das mit dem *Item* verknüpfte *TAlarm*-Objekt erfragt und dessen *Alarmzeit*-Variable auf den neuen Wert gesetzt. Dafür muss allerdings der editierte String in eine Zeitangabe umgewandelt werden, was wieder zu einem Konvertierungsfehler führen kann. Damit nun eine mögliche *EConvertError*-Exception nicht den Prozess des Editierens stört, fängt das Programm diese Exception ab. Es zeigt aber keine Fehlermeldung an, sondern sorgt dafür, dass die Umbenennung des Eintrags nicht durchgeführt wird. Dazu weist es dem *var*-Parameter *S* einfach wieder die bisherige Alarmzeit zu (umgewandelt in einen String).

1.8.6 Komponenten zu Gestaltung und Strukturierung

Viele Komponenten dienen weder zur Ein- noch zur Ausgabe, noch dazu, eine Aktion auszulösen wie die Aktionsschalter, sondern sie haben eine rein gestalterische Funktion oder sind dazu gedacht, den Formularaufbau zu strukturieren:



TGroupBox hat die Aufgabe, mehrere Komponenten in einer Gruppe zusammenzufassen. Dabei bewirkt sie sowohl eine optische Gliederung, die dem Benutzer des Programms zugute kommt, als auch eine technische Gliederung, die dem Entwickler die Übersicht zu wahren hilft. Tiefer gehende Details zu gruppierten Steuerelementen finden Sie in Kapitel 3.5.1.



TPanel ist eine von Borland programmierte Komponente, die Ihnen in den Properties *BevelInner*, *BevelOuter* und *BevelWidth* erlaubt, eine Umrandung ein- und abzuschalten sowie die Art und Dicke der Umrandung zu beeinflussen. Dafür gibt es bei einem *Panel*-Element nicht die von der *GroupBox* bekannte Möglichkeit, eine Überschrift anzugeben (die Beschriftung eines Panels wird vertikal zentriert ausgegeben). Ansonsten hat ein *Panel* die gleiche Funktion wie eine *GroupBox*.



TBevel (auf der Seite ZUSÄTZLICH) gibt Ihnen ähnliche Gestaltungsfunktionen wie *TPanel* (allerdings werden diese hier mit den Properties *Shape* und *Style* gesteuert) und kann zum Zeichnen von Trennlinien im Formular verwendet werden. *TBevel* gliedert das Formular aber *nur* optisch, eine Gruppierung von mehreren Steuerelementen zu einer im Programm besser handzuhabenden Einheit findet nicht statt. Der Vorteil davon ist, dass ein *TBevel* weniger Ressourcen des Betriebssystems verbraucht.



TTabControl (Seite COMMON CONTROLS) ist eine vielseitig verwendbare Komponente zum Umschalten zwischen Dateien, Dateibereichen, Seiten, zwischen verschiedenen Ansichten einer Seite oder was auch immer Sie über die angedeuteten Registerzungen dieser Komponente zugänglich machen wollen. Sie wird zusammen mit verwandten Komponenten in Kapitel 3.5.4 ausführlich behandelt.



TPageControl (in der Palette neben *TTabControl*) ermöglicht Ihnen die Gestaltung mehrseitiger Dialoge und entspricht einem *TTabControl* mit zusätzlicher Unterstützung mehrerer Seiten. Das Hinzufügen und Editieren von neuen Seiten ist zur Entwurfszeit auf eine sehr komfortable Weise möglich (siehe Kapitel 3.5.4).



TScrollBox (Seite STANDARD) dient dazu, eines oder mehrere Kindelemente, die eine größere Fläche beanspruchen, als der ScrollBox auf dem Bildschirm zur Verfügung steht, zu verwalten und zu scrollen. Die Komponente erzeugt automatisch Bildlaufleisten, falls die Kindelemente in horizontaler bzw. vertikaler Richtung mehr Platz benötigen. Auch das Scrollen wird von dieser Komponente automatisch abgewickelt. Außerdem können Sie zwischen normalen dreidimensionalen Bildlaufleisten und solchen im allerneuesten flachen Design wählen.



Mit der *TScrollBar*-Komponente (Seite STANDARD) können Sie auch eine einzelne Bildlaufleiste in Ihr Formular aufnehmen und selbst festlegen, was durch sie gesteuert werden soll. Früher wurden solche Komponenten häufig zur Einstellung von Werten verwendet, für diesen Zweck ist heute aber die *TTrackBar*-Komponente eher zu empfehlen.



Wenn Fenster aus mehreren Bereichen bestehen, werden diese Bereiche häufig durch verschiebbare Balken getrennt, wie zum Beispiel in der Kylix-IDE die Fenster, die aneinander angedockt werden. Für solche Zwecke ist die *TSplitter*-Komponente vorgesehen (Seite ZUSÄTZLICH), allerdings sind bei ihrer Verwendung Besonderheiten zu beachten, die in Kapitel 3.3.1, R3 näher beschrieben sind.

Diese Übersicht soll an dieser Stelle genügen. Das Beispielprogramm *AlarmClock5* und seine Vorgänger haben ja bereits Komponenten der Klassen *TPanel* und *TGroupBox* verwendet, bieten jedoch noch kaum ein Betätigungsfeld für die anderen genannten Gestaltungskomponenten. Daher werden diese im Falle der Seitenauswahlkomponenten an einem eigenen Beispielprogramm im bereits erwähnten Kapitel 3.5.4 behandelt. Weitere Details zu *TScrollBox* finden Sie in Kapitel 3.4.1, Kapitel 6.5.4 beschreibt technische Details zum Scrolling der TreeDesigner-Beispielanwendung, und in Kapitel 8.5.2 wird *TScrollBox* als Basis für eine selbst definierte Komponentenklasse verwendet.

2 Die Sprache Object Pascal

Dieses Kapitel beschäftigt sich mit der Sprache Object Pascal, auf der Kylix aufbaut. Es legt besonderen Wert auf die neueren und objektorientierten Aspekte dieser Sprache, behandelt aber auch alle wichtigen anderen Eigenschaften, insbesondere natürlich alle Eigenschaften, die in diesem Buch verwendet werden.

Object Pascal ist eine Nachfolgesprache von »Borland Pascal mit Objekten«, die 1989 mit Turbo-Pascal 5.5 eingeführt wurde, einer Entwicklungsumgebung, die noch unter DOS lief. Wenig später machte die Sprache in »Turbo Pascal für Windows« Bekanntschaft mit einem grafischen Desktop und 1995 schließlich wurde sie durch umfangreiche Erweiterungen und auch Änderungen fit für ein neues Produkt namens »Delphi« gemacht. Bis Delphi 4 fügte Borland mit jeder Delphi-Version weitere interessante Merkmale zu dieser Sprache hinzu:

- Delphi 1 brachte die bisher grundlegendsten Veränderungen, denn es führte das derzeit vorherrschende Objektmodell (erkennbar am Schlüsselwort *class* im Gegensatz zu *object*, siehe Kapitel 2.2 und 2.3) sowie die Properties und Exceptions ein (Kapitel 2.2.5 bzw. 2.6). Der Name der Sprache wurde von Borland Pascal with Objects auf Object Pascal geändert.
- Delphi 2 stand ganz im Zeichen der Umstellung des Compilers von 16- auf 32-Bit, die meisten Änderungen fanden im Bereich der Typen statt (Kapitel 2.4). Außerdem »legalisierte« es initialisierte Variablen (Abschnitt 2.1.5) und führte neue Aufrufkonventionen für Prozeduren und Funktionen ein (Abschnitt 2.5.2).
- Die wichtigste Neuerung in Delphi 3 waren die Interfaces (Kapitel 2.7), ergänzt von den für die Fehlersuche vorgesehenen Assertions (Kapitel 1.7.6) und weiteren Details wie neuen Aufrufkonventionen und Typen.
- Zu den Spracherweiterungen von Delphi 4 gehören das Überladen von Funktionen, Prozeduren und Methoden, die Definition von Standardwerten für ihre Parameter (zu beidem siehe Kapitel 2.5.4) und die dynamischen Arrays (Kapitel 2.4.3). Auch die schon von Delphi 3 bekannte Unterstützung der COM-Interfaces wurde noch weiter verfeinert (siehe *Delegierte Implementierung* in Kapitel 2.7.5). Schließlich lässt es sich auch Delphi 4 nicht nehmen, ein paar neue Typen einzuführen (Kapitel 2.4).

- Delphi 5 bestätigte den hohen Entwicklungsstand, den Object Pascal in Delphi 4 erreicht hat, indem es nicht schon wieder neue Sprachmerkmale einführte.
- In Kylix stellt sich Object Pascal weiter als ein Hort der Stabilität dar; Erweiterungen waren angesichts der Plattform-übergreifenden Programmierung nur in Randbezirken wie der bedingten Kompilierung und einiger zusätzlicher Deklarationsvarianten (*library, platform*, *deprecated* und *varargs*) erforderlich.

Wir beginnen in diesem Kapitel nicht im Kleinen bei den Typen, Variablen und Anweisungen von Object Pascal, sondern im Großen bei der objektorientierten Sprachstruktur, mit der Sie bei der Verwendung von Kylix schon von Anfang an konfrontiert werden, zu allererst jedoch mit einem Überblick.

2.1 Überblick

Bevor wir richtig in die Welt der Objekte einsteigen, soll dieses Kapitel einen Überblick ber das Modulkonzept und über einige Kleinigkeiten wie Schlüsselwörter, Kommentare und andere lexikalische Elemente geben.

2.1.1 Object Pascal für Umsteiger

Dieses Kapitel enthält kurze Vergleiche von Object Pascal und Sprachen, die mächtiger sind als Object Pascal (C++) oder weniger mächtig (Basic-, Datenbank- und Makrosprachen). Für Umsteiger von Sprachen der letztgenannten Gruppe beschreibt es kurz die Konzepte, die Object Pascal diesen Sprachen voraus hat. Falls Sie einen informellen berblick über die Möglichkeiten suchen, die Sie innerhalb einer Ereignisbearbeitungsmethode haben, lesen Sie bitte das Kapitel 1.5.2.

Wenn Sie sich bereits mit der objektorientierten Programmierung in Turbo- oder Borland Pascal auskennen, benötigen Sie von diesem zweiten Kapitel wahrscheinlich nur die Abschnitte, in denen es um Neuerungen geht. Im folgenden Vergleich zwischen Object Pascal und C++ werden zahlreiche dieser Neuerungen erwähnt. Einige Neue ungen, die speziell für die Programmierung von Komponenten gedacht sind, werden erst in Kapitel 8 besprochen.

Object Pascal und C++

Object Pascal enthält gegenüber Borland Pascal einige Neuerungen, die speziell für C++-Programmierer (und aufgrund der Ähnlichkeit der Sprache zu C++ natürlich auch für Java-Programmierer) interessant sind:

- virtuelle Konstruktoren, mit denen Object Pascal gegenüber C++ auf dem Gebiet der Polymorphie einen interessanten Vorteil besitzt (Kapitel 2.3.4),

- Methodenzeiger, die praktischer und effektiver sind als die Methodenzeiger in C++ (Kapitel 2.5.5),
- Exceptions, die den Exceptions im alten C-Stil entsprechen – ein Vorteil gegenüber den C++-Exceptions ist die *finally*-Anweisung (die in C++ wegen der automatischen Destruktoraufrufe natürlich nicht wirklich benötigt wird) (Kapitel 2.6),
- Typinformationen, die über die RTTI von C++ hinausgehen (Kapitel 2.3.6),
- offene Array-Konstruktoren, mit denen Sie praktisch variable Parameterlisten erhalten (Kapitel 2.5.3),
- Neben den Schutzebenen *private* und *public* kennt Object Pascal nun auch die *protected*-Ebene. Sogar zur *friend*-Deklaration von C++ gibt es eine ähnliche Alternative: Alle Klassen innerhalb einer Unit sind »friends«, die Schutzbestimmungen werden vom Compiler nur zwischen verschiedenen Modulen kontrolliert (Kapitel 2.2.6).
- C++-Programmierer, die schlechte Erfahrungen mit der Mehrfachvererbung gemacht haben, finden vielleicht in den Interfaces von Object Pascal eine saubere Lösung für ihr Programmierproblem. Wie die Interfaces der Sprache Java erreichen sie viele Ziele, die man in C++ nur mit Mehrfachvererbung erreichen konnte, sind aber frei von den großen Risiken dieser C++-Spracheigenschaft (Kapitel 2.7).

Auch wenn Object Pascal einiges zu C++ aufgeholt hat, bleiben einige grundsätzliche Vorteile von C++, die wahrscheinlich auch in Zukunft nicht in Object Pascal ausgeglichen werden können, da dazu die Grundfesten von Object Pascal erschüttert werden müssten. So müssen Sie in Object Pascal des Öfteren zu umständlichen Ausdrucksweisen greifen. Object Pascal enthält manche Schlüsselwörter, die in C++ gar nicht benötigt werden, da sich der Sinn dort von alleine ergibt (z.B. *then*, *var*, *type*, *to* in einer *for*-Schleife), und Operatoren und andere in C++ aus ein oder zwei Zeichen bestehende Symbole beanspruchen in Pascal ein ganzes Schlüsselwort (z.B. *begin*, *end*, *div*, *mod*).

Um die Verzichtsliste noch eine Weile fortzusetzen: In Object Pascal gibt es außerdem keinen Zuweisungsoperator, der flexibel wie jeder andere Operator einsetzbar ist, keinen »?:«-Operator, keinen Methoden-Code in der Klassendeklaration (Inline-Methoden), keine automatischen Konstruktoren/Destruktoren, keine Makros, keine selbst überladbaren Operatoren, keine Mehrfachvererbung von Klassen und auch keine Templates.

Neben den am Anfang schon genannten Highlights von Object Pascal gibt es übrigens noch einige weitere Vorteile, die schon etwas älter sind: Mengen (Kapitel 2.4.5), offene Arrays (Kapitel 2.5.3) und der in diesem Buch wahrscheinlich am häufigsten verwendete Vorteil von Pascal gegenüber C++, die *with*-Anweisung (siehe Kapitel 2.5.1).

Umsteiger von Basic- und Makrosprachen

Wenn Sie bisher mit Visual Basic, einem anderen Basic oder einer Makrosprache gearbeitet haben, werden Sie in Kylix auf viele konzeptuelle Unterschiede treffen, die im Prinzip immer dieselben Ziele haben: die Erstellung und Wartung größerer Programme zu vereinfachen sowie deren Lesbarkeit und Überschaubarkeit zu verbessern. Dabei spielen verschiedene Konzepte wie das Aufteilen einer Programmieraufgabe in berschaubare und möglichst unabhängige Einheiten, das Verbergen von Informationen (Kapselung, Information Hiding) und das Wiederverwenden von Code eine Rolle, die hier jedoch nicht in Einzelheiten erläutert werden können.

Module

Object-Pascal-Module (*Units*) unterscheiden sich von den Modulen vieler anderer Sprachen durch ihre besondere Eigenständigkeit. Während Module in Visual Basic mit dem Formular verschmelzen und Dateien in C++ auf die verschiedensten Arten zusammengebunden werden können, ist eine Unit in Object Pascal ein klar definierter und von Formularen unabhängig gespeicherter Baustein. Der gesamte Quelltext, den Sie in Kylix schreiben, inklusive aller Ereignisbearbeitungsmethoden, wird in Units gespeichert, die im normalen Textfformat vorliegen und daher nach außen völlig offen sind. Der Aufbau eines Formulars wird in einer eigenständigen, wahlweise ebenfalls im Textformat speicherbaren `.xfm`-Datei abgelegt, die Sie auch in lesbarer Form in den Quelltexteditor laden können.

Ein weiterer Vorteil von Units ist ihre klare Schnittstellendefinition. Im *interface*-Teil einer Unit werden alle nach außen hin sichtbaren Elemente der Unit deklariert, alles andere bleibt in der Unit verborgen. Die Einbindung einer Unit findet lokal statt, d.h. eingebundene Units sind nur in den Units sichtbar, in denen sie auch eingebunden werden.

pen

Besonders wichtig ist, dass Pascal-Variablen immer zuerst deklariert werden müssen, bevor sie verwendet werden. Bei dieser Deklaration müssen Sie sich auf einen bestimmten Variablentyp festlegen, allerdings haben Sie mit untypisierten Variablen- arametern oder untypisierten Zeigern auch in Object Pascal sehr große Flexibilität. Schließlich findet die Deklaration von Variable in einem (oder mehreren) eigenen *var*-Blöcken statt, die klar vom Programmcode abgetrennt sind, was zwar unbequem sein kann, aber sehr zur Übersichtlichkeit beiträgt.

Unterschiede im Quelltext

Die Übersichtlichkeit der Pascal-Quelltexte erleichtert das Erlernen der Sprache in besonderer Weise: Zwischen verschiedenen Programmteilen findet eine klare Tren-

nung statt. So besteht eine Unit aus den beiden Teilen *Interface* und *Implementation*, innerhalb derer sich wieder kleinere Blöcke befinden, die mit eigenen Überschriften versehen sind (*var*, *type*, *const*). Im Implementationsteil kommen schließlich die Methoden (Prozeduren und Funktionen) hinzu, deren Aufbau wieder ähnlich strukturiert ist: Zuerst folgen *var*-, *type*- und *const*-Abschnitte, dann der in *begin* und *end* eingeschlossene Programmcode.

Auch innerhalb des Quelltextes ist (Object) Pascal sehr strukturiert. Anweisungsblöcke von Funktionen, Prozeduren und Methoden werden immer durch *begin* und *end* umgeben, bei vielen Kontrollstrukturen wie *if...then*, *while*- und *for*-Schleifen usw. verwenden Sie ebenfalls *begin* und *end*, um mehrere Anweisungen zusammenzufassen. Schließlich gibt es im Aufruf von Prozeduren und Funktionen keinen syntaktischen Unterschied – bei beiden werden die Parameter in Klammern angegeben (was doch eigentlich einfacher ist als die Anweisungs-/Funktions-Unterscheidung im angeblich so einfachen Basic).

Für einen groben praktischen Überblick über einfachen Object-Pascal-Quelltext sei hier noch einmal auf das Kapitel 1.5.2 verwiesen.

OOP

Um mehr über die objektorientierte Programmierung in Object Pascal zu erfahren, sollten Sie das Kapitel 2.2 und folgende lesen. Die Unterschiede zwischen der objektorientierten Programmierung in Object Pascal und den Ansätzen zur objektorientierten Programmierung, die vielleicht in einigen Makrosprachen zu finden sind, sind zu groß, als dass sie einzeln beschrieben werden sollten.

2.1.2 Lexikalische Elemente

Der gesamte Quelltext eines Object Pascal-Programms darf nur aus folgenden Elementen bestehen:

- aus den von der Sprache vorgegebenen Schlüsselwörtern, Operatoren und Interpunktionszeichen
- direkter Angabe von Werten (Konstanten) in verschiedenen Typen (Zahlen, Zeichen und Strings)
- Bezeichnern
- Kommentaren

Wir gehen diese Punkte in umgekehrter Reihenfolge durch.

Kommentare dürfen überall zwischen den anderen syntaktischen Elementen stehen und dürfen alle Arten von Zeichen enthalten, außer natürlich der Zeichenfolge, die als Kommentarende-Markierung verwendet wird. Sie haben die Wahl zwischen drei Begrenzungsarten:

```
{Kommentare können so eingeklammert sein, }
(* oder so. Eine Vermischung der Begrenzer ist nicht erlaubt*)
  x := 0; // Dieser Kommentar endet "automatisch" am Zeilenende
```

Es ist möglich, Kommentare der ersten beiden Formen in zwei Ebenen zu schachteln, sofern Sie für jede Ebene eine andere der beiden Arten von Begrenzungszeichen verwenden.

Bezeichner sind aus den Buchstaben des Alphabets, aus Ziffern und aus dem Unterstrich aufgebaut, dürfen aber aus Gründen der Eindeutigkeit nicht mit einer Ziffer beginnen. Im folgenden Beispiel-Bezeichner ist eine weitere Regel versteckt:

```
Zwischen_Gross_und_Kleinschreibung_wird_nicht_unterschieden (* . *)
```

Die verschiedenen Typen konstanter Werte von Object Pascal werden im folgenden zusammengefasst.

Art der Konstantenangabe	Aufbau	Beispiel
ganze Zahlen	Folge aus 0...9	9876543210
Hexadezimalzahlen	Hexadezimalzahl mit vorangestelltem '$'-Zeichen	$F0D9
Fließkommazahlen	Kommazahl+Exponent	3.4e5 – 1.4e+100 – 49e-4
einzelne Zeichen	Zeichen, eingeschlossen wie folgt:	'X'
einzelne Zeichen	#+Zahlenwert	#65 (entspricht in ASCII- und ANSI-Code dem 'A')
einzelne Steuerzeichen	^A	entspricht dem Zeichen #1
Strings	Zeichenfolgen, eingeschlossen wie im Beispiel	"XYZ"
Mengen	Liste von Elementen, eingeschlossen in eckigen Klammern	[biSystemMenu, biMinimize, biMaximize]

Sonderfälle sind, wenn Sie Begrenzungszeichen selbst als Zeichen oder String angeben sollen: Sie werden dann einfach zweimal hintereinander geschrieben (ein Hochkomma als Zeichen wird geschrieben als `''''` und ein einzelnes Anführungszeichen als String sieht so aus: `""`).

Schlüsselwörter: Die mittlerweile schon 64 Schlüsselwörter von Object Pascal, die Sie nicht für eigene Bezeichnernamen verwenden dürfen, sind:

and	array	as	asm	begin	class
case	const	constructor	destructor	dispinterface	div
do	downto	else	end	except	exports
file	finalization	finally	for	function	goto
if	implementation	in	inherited	initialization	inline
interface	is	label	library	mod	nil
not	object	of	on	or	packed
procedure	program	property	raise	record	repeat
set	shl	shr	string	then	threadvar
to	try	type	unit	until	uses
var	while	with	xor		

Ebenfalls zur Sprache Object Pascal gehören die folgenden Standarddirektiven:

absolute	abstract	assembler	automated	cdecl	default
deprecated	dispid	dynamic	export	external	forward
implements	index	library	message	name	nodefault
overload	override	pascal	platform	private	protected
public	published	read	readonly	register	reintroduce
resident	safecall	stdcall	stored	virtual	write
writeonly					

Die Besonderheit der Standarddirektiven gegenüber den Schlüsselwörtern ist, dass sie nicht ausschließlich für den Compiler reserviert sind. Sie dürfen diese Direktiven ohne Einschränkung für eigene Bezeichner verwenden, weil keine Zweideutigkeiten entstehen können: Da, wo die meisten der Standarddirektiven stehen – am Ende von Funktionsköpfen –, können keine Bezeichner stehen und umgekehrt kommen die Standarddirektiven nie in Funktionsrümpfen vor. Die Worte *private*, *protected*, *public* und *published* sind allerdings innerhalb einer Objekttyp-Deklaration reserviert, können also nicht als Name von Methoden oder Variablen einer Objektklasse verwendet werden, weil der Compiler dann wieder mehr überlegen müsste, ob nun ein Bezeichner oder die Direktive gemeint ist.

Bei Standarddirektiven und Schlüsselwörtern unterscheidet Object Pascal ebenso wenig zwischen Groß- und Kleinschreibung wie bei Bezeichnern.

Semikolons

Wie schon in der Schnellübersicht aus Kapitel 1.5.2 gezeigt, besteht der Programmcode auf höherer Ebene aus Zuweisungen, Aufrufen von Methoden/Funktionen/Prozeduren und Kontrollstrukturen. Jeder dieser drei Anweisungstypen wird an geeigneter Stelle beschrieben (wobei die Zuweisung kaum über das Kapitel 1.5.2 hinauskommt), so dass hier nur eine grundsätzliche Bemerkung zu den Semikolons übrig bleibt.

Semikolons werden in Pascal grundsätzlich dazu verwendet, Anweisungen zu trennen, das heißt, dass Sie die letzte Anweisung eines Anweisungsblocks nicht durch ein Semikolon abschließen müssen. Da Pascal jedoch eine leere Anweisung akzeptiert, können Sie nach der letzten Anweisung (und an anderen Stellen) beliebig viele weitere Semikolons setzen. In den Programmen dieses Buchs werden der Einfachheit halber alle Anweisungen durch ein Semikolon abgeschlossen.

Zur Verwendung der Semikolons in Konstanten-, Typen- und Variablendeklarationen und an anderen Stellen nehmen die entsprechenden Abschnitte dieses Kapitels Stellung.

2.1.3 Compileranweisungen

Innerhalb eines Quelltextes können Sie nicht nur Anweisungen für das Programm, sondern auch für den Compiler unterbringen. Da diese nicht zur eigentlichen Programmiersprache gehören, haben sie sich ein Versteck gesucht, wo sie nicht mit der Sprache verwechselt werden können: die Kommentare. Zur Unterscheidung von normalen Kommentaren beginnen sie mit einem »$«-Zeichen. Um also beispielsweise den Compiler-Schalter *$B* einzuschalten, genügt es, ein »*{$B+}*« in den Quelltext zu schreiben.

Schalter

Die einfachste Anweisung an den Compiler ist es, eine der Optionen zu setzen, die Sie auch in der Dialogbox PROJEKT | OPTIONEN finden. Zu jeder Dialogboxoption gibt es eine entsprechende Compileranweisung und darüber hinaus einige weitere Compilerschalter. Manche davon dürfen nur am Anfang einer Quelltextdatei gesetzt werden. Genauere Informationen zu den einzelnen Optionen finden Sie in der Online-Hilfe. Einige wichtige Schalter, die auch in diesem Buch verwendet oder an anderer Stelle erwähnt werden, sind:

- $B: Auswertung von booleschen Ausdrücken, siehe *Auswertung von booleschen Ausdrücken* in Kapitel 2.4.2
- $O: Optimierung an/aus (siehe Kapitel 1.7.1)
- $C: Assertion-Auswertung an/aus (siehe Kapitel 1.7.6)

- $H: steuert Schalter die Verwendung der langen Strings (siehe Kapitel 2.4.4).
- $I: I/O-Fehlerbehandlung mit Exceptions oder mit *IoResult* (siehe Kapitel 2.6.5)
- $J: macht typisierte Konstanten zu echten Konstanten (siehe Kapitel 2.1.5).
- $R: Bereichsüberprüfung (siehe Kapitel 2.6.5)
- $S: Stacküberprüfung, bricht das Programm bei einem Stack-Überlauf ab.
- $D, $L, $Y: Informationen für Browser und Debugger in die DCU-Datei aufnehmen (Kapitel 1.7.1)
- $P: Aktivierung von offenen Arrays (Kapitel 2.5.3)
- $N: Einschalten der erweiterten Fließkommatypen (Kapitel 2.4.1)

Alternativ zu diesen Kürzeln können Sie auch eine ausführlichere Schreibweise wählen, z.B. *$BooleVal On* statt *$B+* und *$Assertions Off* statt *$C-*. Eine Übersicht finden Sie in der Online-Hilfe.

Neben den Schaltern gibt es einige Anweisungen, von denen hier nur die *$I-* und die *$*R-Anweisung erwähnt werden sollen.

Ressourcen und Include-Dateien

Mit der folgenden Compileranweisung können Sie den Compiler veranlassen, temporär zum Zeitpunkt der Übersetzung eine andere Datei in den Quelltext einzufügen, im folgenden Fall `incl.pas`:

```
(*$I incl.pas*)
```

Dies war früher eine Möglichkeit, sich gegen das Fehlen des Unit-Konzepts zu wehren und einen Quelltext auf mehrere Dateien zu verteilen.

Ähnlich und zurzeit noch häufiger verwendet wird die Anweisung *$R Datei*. Eine solche finden Sie in jeder Formular- und in jeder Projektdatei. Sie bindet Ressourcendateien (`.res`) oder Formulardateien (`.xfm`) zur ausführbaren Datei hinzu. Ressourcen, die in Kylix nicht in idealer Weise editiert werden können, wie z.B. Bitmaps, können Sie so mit einem externen Tool erstellen und in Ihr Projekt einbinden (siehe Kapitel 1.4.9).

Bedingte Compilierung

Durch bedingte Compilierung ist es möglich, mehrere Versionen eines Programms mit nur einer Quelltextdatei zu generieren. Dies lohnt sich dann, wenn die Unterschiede zwischen beiden Versionen nur aus wenigen Zeilen bestehen. Beispielsweise könnte es eine Programmversion für Windows geben und eine andere für Linux, oder Sie könnten zusätzlichen Code für die Fehlersuche einfügen, der jedoch in der endgültigen Programmversion nicht mitübersetzt werden soll:

```
{$ifdef CompilerBedingung}
   ... Programmcode ...
{$else}
   ... alternativer Programmcode ...
{$endif}
```

In diesem Beispiel würde der Compiler den ersten Programmteil übersetzen, wenn das Symbol *CompilerBedingung* definiert ist, ansonsten würde er sich für den zweiten Teil entscheiden. Dieser *else*-Teil ist optional. Statt dem *ifdef* könnten Sie auch ein *ifndef* verwenden, was soviel bedeutet wie *»wenn nicht«*. Das Symbol *CompilerBedingung* hat nichts mit irgendwelchen Bezeichnern des Quelltextes zu tun, sondern muss auf andere Weise definiert werden:

- durch die Compileranweisung *{$define CompilerBedingung}*,
- durch Angabe des Symbols in den Projektoptionen, Seite VERZEICHNISSE/BEDINGUNGEN, Feld DEFINITION,
- durch den Compiler selbst. Die vom Compiler vordefinierten Symbole ermöglichen es, zwischen verschiedenen Versionen zu unterscheiden und je nach Version unterschiedlichen Code auszuführen. Die Zählung der Compiler-Version geht dabei auf Turbo Pascal 1.0 zurück, so dass die aktuellen Entwicklungssysteme hier schon lange in den zweistelligen Versionsnummern sind. Die folgende Tabelle zeigt die wichtigsten Symbole der bisherigen »Delphi-kompatiblen Produkte« von Borland:

Produkt	Vordefinierte Symbole
Delphi 1	VER80, Windows, CPU86
Delphi 2	VER90, Win32, CPU386
Delphi 3	VER100, Win32, CPU386
Delphi 4	VER120, Win32, CPU386
Delphi 5	VER130, Win32, CPU386
Delphi 6	VER140, Win32, MSWINDOWS, CPU386
ylix 1	VER140, Linux, CPU386

Neue Bedingungs-Direktiven

Ab der Version »Ver140« unterstützt der Compiler noch ein alternatives Set von Direktiven zur bedingten Übersetzung: *$if*, *$elseif* und *$ifend*. Über das Abfragen einfacher Symbole hinaus können Sie mit *$if* und *$elseif* auch Ausdrücke auswerten:

```
const MeineKonstante = 'LN';
{$if MeineKonstante = 'XA'}
```

Dabei können Sie auf deklarierte Objekt Pascal-Konstanten wie oben auf *MeineKonstante* zugreifen. Besonders interessant ist die Abfrage der vordefinierten Konstante *RTL-*

Version, die in Delphi 6 und Kylix als »14.0« definiert ist und damit der Compiler-Version entspricht. Der Unterschied zur Abfrage des Compiler-Symbols *VER140* liegt darin, dass Sie auch alle nachfolgenden Versionen in der Bedingung berücksichtigen können:

```
{$ifdef VER140} // in nachfolgender Version nicht mehr definiert
{$if RTLVersion >= 14} // in nachfolgenden Versionen definiert
```

Schließlich bietet Kylix Ihnen für die Ausdrücke in *$if* noch zwei spezielle »Funktionen« an: Mit *Declared(MeinSymbol)* können Sie abfragen, ob ein Object-Pascal-Symbol deklariert wurde, mit *Defined(Compilersymbol)* fragen Sie ab, ob ein Compiler-Symbol, wie es auch mit der alten Direktive *$ifdef* abgefragt werden kann, definiert ist.

2.1.4 Typen und Variablen

In der visuellen Umgebung der Kylix-IDE kommen Sie zwar vorwiegend mit Formularen, Komponenten und Properties in Berührung, auf Methodenebene wird in Kylix jedoch mit den Variablen- und Typenkonzepten programmiert, die überwiegend schon in der 1972 von Niklaus Wirth entwickelten Ur-Pascal-Version vorhanden waren. Klassen und Objekte sind in dieses Konzept eingebaut: So entpuppen sich die Klassen syntaktisch als Erweiterungen der Record*typen*, und dementsprechend sind Objekte *Variablen*, denen als Typ eine Objektklasse zugewiesen wurde. In diesem Abschnitt sollen nur die allgemeinen Regeln für Typen und Variablen zusammengefasst werden, in Kapitel 2.4 finden Sie einen Überblick über alle Typen.

Deklarationsblöcke

Durch die übersichtliche Struktur eines Pascal-Programms lassen sich Variablen, Typen und Konstanten einfacher unterscheiden als beispielsweise in C++ oder Java: Jeder der drei Bezeichnertypen muss in einem Abschnitt mit spezieller Überschrift deklariert werden. Diese Abschnitte befinden sich immer außerhalb der Abschnitte, die den Programmcode enthalten (also außerhalb jeglicher *begin...end*-Blöcke). So gehören die Variablen in den *var*-Abschnitt:

```
var
  EineZahl: Integer;
  Taste1, Taste2: Char;
begin
  ...
```

Eine Variablendeklaration weist einer oder mehrerer durch Kommata getrennten Variablen den hinter dem Doppelpunkt stehenden Typ zu. Die oben verwendeten Typen *Integer* und *Char* sind bereits vordefiniert.

ypendeklarationen

In jeder Formulardatei befindet sich bereits die Deklaration eines Formulartyps bzw. einer Formularklasse, die somit ein besonders umfangreiches Beispiel einer Typendeklaration gibt. Die einfachste Art, einen eigenen Typ zu deklarieren, besteht darin, ihn einfach als Ersatzbezeichner für einen der vordefinierten Standardtypen zu verwenden:

```
type
  TTabellenIndex = Integer; { deklariert den neuen Typ "TTabellenIndex" }
  TSeitenIndex = Integer;
  TBuchIndex = Byte;
```

Sie verwenden selbst definierte Typen genauso wie die vordefinierten:

```
var
  EineSeitenzahl: TSeitenIndex;
```

Die Unterschiede eines Typendeklarationsblocks zum Variablenblock liegen in der berschrift *type*, in der Beschränkung auf einen Bezeichner pro Deklaration und in dem Gleichheitszeichen anstelle des Doppelpunkts. Mit der obigen Typendefinition haben Sie später beispielsweise die Möglichkeit, durch Ändern des *TBuchIndex*-Typs in *Word* alle *Byte*-Variablen, die für den Buchindex stehen, auf den größeren Wertebereich von *Word* zu setzen, während alle anderen *Byte*-Variablen unverändert bleiben – eine Maßnahme, die Ihnen mit Suchen&Ersetzen viel Arbeit gemacht hätte.

Weitere Deklarationsblöcke

Neben den Abschnitten *var* und *type* gibt es noch *const* (Kapitel 2.1.5), *exports* (Kapitel 5.1.7) und *label* (für *goto*-Sprünge, die in diesem Buch nicht näher erläutert werden).

Die Reihenfolge der Abschnitte spielt in Object Pascal keine Rolle: Sie können beliebig viele Variablen- und Typenblöcke miteinander abwechseln. Wenn Sie sie erst nach bestimmten Prozeduren, Funktionen und Methoden aufführen, schränkt das lediglich die Sichtbarkeit der Bezeichner ein, denn diese dürfen erst in den Prozeduren verwendet werden, die unterhalb der Bezeichnerdeklaration stehen.

2.1.5 Konstanten und initialisierte Variablen

Kapitel 2.1.2 hat die direkte Angabe von konstanten Werten bereits als elementaren Bestandteil des Quelltextes genannt. Sie können solchen konstanten Werten auch einen Namen geben und sie dadurch leichter mehrfach verwenden. Auch ohne diesen Namen könnte man einen Wert bereits als Konstante bezeichnen, in Object Pascal meint die Bezeichnung *Konstante* aber einen *Bezeichner*, der für einen konstanten Wert steht.

In Object Pascal gibt es zwei verschiedene Arten von Konstanten: In echten Konstanten geben Sie einem Bezeichner einen festen Wert, den der Compiler überall dort einsetzt, wo Sie den Bezeichner verwenden:

```
const
  SampleRate = 44100;
  BitAnzahl = 16;
  Kanaele = 2;
  BytesProSek = SampleRate*(BitAnzahl div 8)*Kanaele;
begin
  Sekundendaten := GetMem(BytesProSek);
```

Wie Sie dem Beispiel entnehmen können, akzeptiert der Compiler bei Konstanten auch Ausdrücke, solange in diesen wiederum nur Konstanten vorkommen.

Typisierte Konstanten

Bei der zweiten Art von Konstanten handelt es sich in Wirklichkeit um initialisierte Variablen. Hier setzt der Compiler keinen konstanten Wert in Ihr Programm ein, sondern behandelt die typisierten Konstanten wie Variablen:

```
const
   SampleRate: Word = 44100;
begin
   SampleRate := 48000;
```

Die syntaktischen Unterschiede zu einer normalen Variablendeklaration beschränken sich auf die Zuweisung eines Wertes hinter dem Gleichheitszeichen und auf die Überschrift *const*.

Außerdem haben Sie die Möglichkeit, solche typisierten Konstanten zu echten Konstanten zu machen, denen Sie im Programmcode keine neuen Werte mehr zuweisen dürfen. In den Projektoptionen finden Sie diese Option auf der Compiler-Seite unter dem Namen ZUWEISBARE TYPISIERTE KONSTANTEN, die Compileroption für den Quelltext lautet *{$J}*. Der Kompatibilität wegen ist diese Option per Voreinstellung *ein*geschaltet.

Initialisierte Variablen

Es ist jedoch nicht notwendig, den Umweg über typisierte Konstanten zu gehen, um eine initialisierte globale Variable zu erhalten. Sie können hier allen globalen Variablen Initialisierungswerte zuweisen, die diese bei Programmstart anstatt der voreingestellten Null-Werte erhalten. Die Syntax stimmt mit der Konstantendeklaration überein, einziger Unterschied ist die Überschrift *var*:

```
var
  SampleRate: Word = 44100;
```

Lokale typisierte Konstanten

Lokale Variablen bleiben weiterhin immer uninitialisiert. Sie dürfen in Object Pascal jedoch typisierte Konstanten lokal zu einer Funktion deklarieren. Die Konstante ist dann wie eine lokale Variable nur innerhalb dieser Funktion ansprechbar, obwohl es sich eigentlich um eine initialisierte globale Variable handelt.

```
procedure CriticalProcedure;
const
  CallCounter: LongInt = 0;
begin
  inc(CallCounter);
```

In diesem Beispiel zählt die Prozedur *CriticalProcedure* ihre Aufrufe in einer »privaten« ariablen *CallCounter*, die bei Programmstart mit Null initialisiert ist.

2.1.6 Gültigkeitsbereiche und lokale Variablen

Jeden Bezeichner, den Sie in Object Pascal verwenden, indem Sie ihm einen Wert zuweisen oder in einen Ausdruck aufnehmen, müssen Sie vorher deklarieren. Ab dieser Deklaration ist der Bezeichner bis zum Ende seines Gültigkeitsbereichs verwendbar. Als Gültigkeitsbereiche zählen die Bereiche von Modulen, Funktionen und Klassen:

Bezeichner, die innerhalb einer Funktion, Prozedur oder Methode deklariert werden, sind nur innerhalb dieser gültig und werden als *lokal* bezeichnet:

```
procedure TForm1.FormCreate(Sender: TObject);
var { Deklaration lokaler Variablen }
  Start, LokalerZaehler: Integer;
begin
  LokalerZaehler := 0; { Verwendung der Variable }
  ...
```

Wenn Sie Funktionen schachteln, kann die geschachtelte Funktion auf alle vorher deklarierten lokalen Variablen der umgebenden Funktion zugreifen.

Bezeichner, die Sie in einer Klassendeklaration deklarieren, sind in allen Methoden der Klasse gültig und darüber hinaus in abgeleiteten Klassen, wobei hier die Schutzbestimmungen beachtet werden müssen (siehe Kapitel 2.2.6). Auch der Gültigkeitsbereich von Bezeichnern in Records ist auf den Record beschränkt. Innerhalb des Records gibt es zwar niemanden (keine Methode), der dadurch einen Vorteil hätte, jedoch können Sie mit der *with*-Anweisung in den Gültigkeitsbereich eines Records »hinabsteigen« (siehe Kapitel 2.5.1).

Bezeichner, die außerhalb der genannten Bereiche deklariert sind, heißen *globale* Bezeichner. Sie sind im gesamten Modul gültig, beginnend mit der Zeile ihrer Deklaration.

Verdeckungen

Globale Bezeichner können zeitweise durch lokale Bezeichner verdeckt werden. Gibt es im obigen Beispiel neben der lokalen auch eine globale Variable mit dem Namen *Start*, so wird diese von der lokalen Variablen *Start* verdeckt. Sie können die globale Variable aber dennoch ansprechen, indem Sie deren Gültigkeitsbereich direkt angeben. Der globale Gültigkeitsbereich hat den Namen des Moduls – wenn dieser beispielsweise *Unit1* ist, könnte die obige Methode auf die globale Variable wie folgt zugreifen:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Start: Integer;
begin
  Unit1.Start := 0; { spricht die globale Variable an }
  Start := 0; { spricht die lokale Variable an }
  ...
```

2.1.7 Das Modulkonzept

In Kylix gibt es drei verschiedene Modultypen: *library, program* und *unit*. Die Aufgaben der letzten beiden sind in Zusammenhang mit den Dateien eines Projekts schon in Kapitel 1.6.2 erläutert worden: *program*-Module dienen in Kylix als Projektdatei mit der Endung `.dpr`, in die mehrere *unit*-Module (Units) mit der Endung `.pas` eingebunden werden können. Eine *library* ist eine Bibliothek, die erst zur Programmlaufzeit zum Programm hinzugebunden wird (unter Linux an der Endung .so zu erkennen, unter Windows an der Endung .DLL).

Die Projektdatei (bzw. das Hauptmodul)

Am einfachsten ist das Hauptmodul aufgebaut:

```
[program ModulName;]
[uses SysUtils, ...;]
[beliebige Folge von var-, const-, type-Blöcken und Definitionen von Methoden,
Prozeduren und Funktionen]
begin
  [Anweisungsfolge]
end.
```

Alles außer dem *begin...end*-Block (dem so genannten *Hauptprogramm*) ist optional. Unmittelbar nach dem abschließenden Punkt hinter dem *end* wendet sich der Compiler von der Datei ab, unabhängig davon, ob noch etwas folgt oder nicht.

Zu den optionalen Elementen des Programms gehört der Name des Moduls in der ersten Zeile (fehlt die *program*-Zeile, übernimmt der Compiler den Dateinamen). Dieser Name kann z. B. bei Zugriff auf den äußeren Gültigkeitsbereich des Moduls aus dem Innern des Moduls (von innerhalb einer Funktion) oder von einem äußeren Modul aus

verwendet werden, um globale Bezeichner des Moduls anzusprechen, wenn diese durch andere Bezeichner verdeckt sind (siehe Kapitel 2.1.6).

In der *uses*-Klausel können Sie Units in das Programm einbinden (dazu später mehr). In einer nicht festgelegten Reihenfolge folgen darauf Typen, Variablen, Konstanten und Methoden.

Units

Die *unit* weist einen weniger freien Aufbau auf, besteht sie doch aus zwei Teilen namens *interface* und *implementation*.

```
unit ModulName; { Modulname muss angegeben werden }

interface
[uses-Klausel]
[Deklarationen]

implementation
[uses-Klausel]
[Deklarationen und Definitionen]

Unit-Hauptprogramm, abkürzbar als "end."
```

Eine Unit kann alle Arten von Object-Pascal-Objekten enthalten, wobei mit »Objekt« hier allgemein die Objekte des Programmierens, also Prozeduren, Funktionen, Variablen, Klassen etc. gemeint sind. Sinn der Unit ist es, diese Objekte teilweise oder ganz anderen Modulen zur Verfügung zu stellen. Die Unit ist also durchaus berechtigt, einige Bestandteile geheim zu halten.

Alle Objekte, die in anderen Modulen benutzt werden dürfen, müssen im *Interface* deklariert werden. Dadurch werden die entsprechenden Bezeichner öffentlich (*public*), also nach außen hin sichtbar. Bei Variablen genügt diese einmalige Deklaration, alle Funktionen des Programms und der Objekte müssen außerdem noch definiert werden, und zwar im Abschnitt *Implementation*. Alle weiteren Deklarationen, die innerhalb dieses Teils stehen, sind privat.

hnlich verhält es sich mit den *uses*-Klauseln: Nur die öffentlichen Symbole der Units, die Sie im Interface einbinden, werden automatisch auch den äußeren Modulen bekannt. Der eigentliche Sinn zweier getrennter *uses*-Anweisungen ist jedoch, zirkure Nutzungsverhältnisse festzulegen – wenn sich beispielsweise zwei Units gegenseitig benutzen wollen. Das folgende Beispiel würde den Compiler in eine Endlosschleife ürzen, wenn dieser den Fehler nicht erkennen würde:

```
{ Dateibeginn der ersten Unit: }
unit Unit1; interface uses Unit2;
{ Dateibeginn der zweiten Unit: }
unit Unit2; interface uses Unit1;
```

In diesem Fall muss eine der *uses*-Klauseln auf den Implementationsteil der Unit verschoben werden, beispielsweise:

```
{ Beginn der ersten Unit: }
unit Unit1; uses Unit2;
---
{ Beginn der zweiten Unit: }
unit Unit2;
interface
implementation
uses Unit1;
```

Wenn der Compiler nun *Unit1* übersetzt, kann er das Interface der *Unit2* einlesen, ohne dort sofort aufgetragen zu bekommen, sich zuerst dem Interface von *Unit1* zuzuwenden. Die obige Ausweichlösung bringt natürlich als Einschränkung mit sich, dass Sie im Interface von *Unit2* keine Bezeichner aus *Unit1* mehr verwenden können; diese Einschränkung dürfte sich in der Praxis jedoch kaum auswirken.

Units initialisieren und verlassen

Der in der obigen Übersicht als »Unit-Hauptprogramm« ausgewiesene Abschnitt wurde in Object Pascal bisher von Version zu Version leicht erweitert. Da der in diesem Abschnitt ausgewiesene Code beim Programmstart ausgeführt wird und die Unit so Gelegenheit hat, sich selbst zu initialisieren, können Sie diesen Abschnitt statt mit *begin* auch mit *initialization* einleiten.

Passend dazu gibt es noch das Schlüsselwort *finalization*, das als Überschrift für den Code dient, der bei Beendigung des Programms auf jeden Fall aufgerufen werden soll. Das Ende einer Unit kann damit wie folgt aussehen:

```
initialization
  { Code zum Initialisieren der Unit }
finalization
  { Code zum Aufräumen nach der Programmausführung }
end.
```

2.2 Objekte und Klassen in Object Pascal

Das zentrale Konzept der objektorientierten Programmierung ist die Zusammenfassung von Daten und Code – in der prozeduralen Programmierung noch strikt getrennt – zu abgeschlossenen Strukturen namens *Objekten*. Die Objekte, mit denen Sie im ersten Kapitel Bekanntschaft gemacht haben, waren in erster Linie Komponenten und Formulare. Auch die Bestandteile der einzelnen Komponenten, die Sie im Objekt-

inspektor angezeigt finden, sind nicht immer nur einfache Werte wie Zahlen oder Strings, sondern mitunter auch Objekte, beispielsweise die *Font*-Objekte verschiedener Komponenten.

Aus der Vereinigung von Daten und Code ergeben sich viele Vorteile wie z. B.:

- Innerhalb einer Methode haben Sie direkten Zugriff auf alle Elemente des Objekts, ohne dass Sie mit einer *with*-Anweisung oder auf andere Arten auf eine »fremde« Datenstruktur zugreifen müssen (siehe Kapitel 2.5.1).
- Es steht (normalerweise) immer fest, welche Prozeduren zu welchen Daten gehören.
- Alle Probleme der globalen Variablen können vermieden werden.
- Sie können die Daten eines Objekts verbergen, so dass der Nutzer des Objekts nur die Methoden, aber nicht die Implementierung des Objekts zu kennen braucht (insofern ist ein Objekt so etwas wie eine eigene Unit mit Interface- und Implementationsteil).

Hinzu kommen natürlich die Vorteile der objektorientierten Programmierung im Allgemeinen, zu denen die leichtere Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit von Code und die bessere Strukturierung der Programmieraufgabe gehören.

In diesem Kapitel sehen wir uns zuerst das Innenleben der Objekte an, bevor wir in Kapitel 2.3 untersuchen, wo die Objekte entstehen und wo sie enden.

2.2.1 Der Aufbau von Objekten

ür Object Pascal bedeutet die Vereinigung von Code und Daten konkret, dass ein Objekt aus Properties, Variablen und Methoden besteht, wobei von außen betrachtet Properties und Variablen den Datenanteil des Objekts bilden und der Code sich in den Methoden befindet. (Von innen betrachtet handelt es sich bei den Properties ja um eine Kombination von Code und Daten.)

OOP und visuelle Programmierung

In der visuellen Programmierumgebung von Kylix haben Sie Zugriff auf den Teil der Properties und Methoden, den die Programmierer der entsprechenden Komponente ür die Kylix-Oberfläche freigegeben haben (siehe Kapitel 8.4.3).

Wie Sie schon in Kapitel 1 erfahren haben, gestalten Sie beim Entwurf des Formulars nicht die eigentlichen Fensterobjekte, sondern nur die Schablonen dafür. Dass von der Schablone des Hauptformulars beim Programmstart nur ein Fenster geöffnet wird, ist lediglich eine sinnvolle Voreinstellung von Kylix, denn von den Formularschablonen önnen Sie theoretisch beliebig viele Exemplare des gleichen Fensters erzeugen.

2.2.2 Klassen und Instanzen

Das Formular, das am Anfang der Unit deklariert ist, wird in Object Pascal als Objekt*klasse* bezeichnet. Das Fenster, das zur Laufzeit am Bildschirm zu sehen ist, liegt im Programm als Objekt*instanz* vor. Diese Unterscheidung von Objektinstanz und Objektklasse gilt auch für alle anderen Objekte in Object Pascal, die nichts mit Fenstern zu tun haben. Mit der Kurzbezeichnung *Objekt* ist in diesem Buch immer die Objektinstanz, also die Variable gemeint; Objektklassen lassen sich in Object Pascal nun ohne Bedenken kurz als *Klassen* bezeichnen, da sie mit dem Schlüsselwort *class* deklariert werden, und nicht wie unter Borland Pascal mit dem Schlüsselwort *object* (eine alternative Bezeichnung für die Klasse ist *Objekttyp*). Dennoch werden Sie feststellen, dass Borlands Online-Hilfe es hier nicht so genau nimmt und Klassen, die keine Komponentenklassen sind, auch als »Objekte« bezeichnet.

Ein wichtiger Unterschied zum *Record* liegt bei den Klassen darin, dass Sie sie nicht direkt einer Variablen zuweisen können. Record-Variablen können Sie noch wie folgt deklarieren:

```
var
  Struktur: record
    Element1, Element2: Char;
  end;
```

Dies ist bei Objekten nicht mehr möglich. Da Sie von diesen meist mehrere Exemplare benötigen, ist es auch sinnvoll, für jedes Objekt zuerst eine Klasse und davon dann eine Objektinstanz anzulegen. Falls die obige Variable *Struktur* also keine normale Variable, sondern ein Objekt sein soll, müsste die Deklaration wie folgt aussehen:

```
type
  TStruktur: class
    Element1, Element2: Char;
  end;

var
  Struktur: TStruktur;
```

2.2.3 Die Klassendeklaration

Sehen wir uns eine beispielhafte Objektklasse an: Angenommen, Sie haben Ihrem Formular den Namen *MainWindow* gegeben, so erzeugt Kylix im dazugehörigen Programmcode eine Klasse mit dem Namen *TMainWindow*. Diese könnte beispielsweise wie folgt aussehen:

```
type
  TMainWindow = class(TForm)    TabSet1: TTabSet;
    ColorGrid1: TColorGrid;
    ScrollBar1: TScrollBar;
```

```
    CheckBox1: TCheckBox;
    procedure FormPaint(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure TabSet1Change(Sender: TObject; NewTab: Integer;
      var AllowChange: Boolean);
  private
    { Private-Deklarationen }
    x3, y3, x4, y4: Integer;
  public
    { Public-Deklarationen }
  end;
```

Da es sich bei jeder neuen Klasse um einen »selbst definierten Typ« handelt, befindet sich diese Deklaration in einem Abschnitt, der durch das Schlüsselwort *type* eingeleitet wird (allerdings würde die Bezeichnung als »Datentyp« dem Wesen der Objekte nicht mehr gerecht, da sie die Methoden nicht angemessen berücksichtigen würde). Das wichtigste Schlüsselwort bei der Deklaration ist das Wort *class*. Es unterscheidet die ihm folgende Struktur von den Records und den alten Objekttypen, die stattdessen mit *cord* bzw. mit *object* eingeleitet werden. Wir verwenden hier ausschließlich Objektklassen, die mit dem Schlüsselwort *class* deklariert werden; auf Unterschiede zu den mit *object* deklarierten Verwandten geht Kapitel 2.3 (der Lebenslauf von Objekten) ein. Nach *class* wird in Klammern die Basisklasse angegeben, von der die Klasse *erbt*. Mit dieser Vererbung beschäftigt sich Kapitel 2.2.7.

Darauf folgen die Bestandteile der Klasse, und zwar in Abschnitten mit den Überschriften *public*, *published*, *private* und *protected* (siehe Kapitel 2.2.6), die in einer beliebigen Reihenfolge auftreten können, allerdings müssen Sie bei den Formularklassen den ersten, unbetitelten Abschnitt für die automatische Code-Erzeugung von Kylix reservieren.

Innerhalb eines Abschnitts müssen Sie zuerst die Variablen und dann die Methoden aufführen. Sobald die erste Methode erscheint, dürfen Variablen erst wieder nach einer neuen Überschrift wie z.B. *public* folgen. Die einzelnen Deklarationen von Methoden und Variablen halten sich an dieselben Regeln wie normale Variablen, Prozeduren und Funktionen.

Mit einem *end* und einem Semikolon schließen Sie die Klassendeklaration ab. Weitere Informationen speziell zum automatisch erzeugten Anteil der Formularklassen finden Sie in Kapitel 1.5.8.

2.2.4 Zugriff auf die Klassenelemente

Wir haben im Laufe der ersten Kapitel schon des Öfteren auf die Elemente einer Klasse zugegriffen, es wird also Zeit, zu untersuchen, von wo aus die Elemente eines Objekts auf welche Art angesprochen werden können.

Der self-Parameter

Für den Einstieg in die OOP kann das Verständnis des *self*-Parameters wichtig sein, wenn Sie bisher zu stark an die Verwendung von nicht-objektorientierten Strukturen und damit an die strikte Trennung von Code und Daten gewöhnt sind. Da der *self*-Parameter an sich etwas sehr »Natürliches« ist, wird er Ihnen wahrscheinlich bald völlig selbstverständlich vorkommen.

Ein Objekt soll natürlich seine Elemente gut kennen, daher ist es logisch, dass Sie in einer Objektmethode nur den Namen des Elements anzugeben brauchen, um sich auf dieses zu beziehen. So können Sie innerhalb einer Ereignisbearbeitungsmethode eines Formulars direkt auf die Properties des Formulars zugreifen:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Color := clRed+clGreen+clBlue;
```

Im Falle eines einzelnen Fensters scheint es dafür noch keine Probleme zu geben. Wenn nun aber viele Fenster derselben Formularklasse z.B. als MDI-Kinder existieren, dann verfügt jedes dieser Fenster über seine eigene Farbe. Wie kann nun die obige Zeile wissen, wessen *Color*-Property gemeint ist?

Die Antwort liegt im unsichtbaren *self*-Parameter, den jede Methode automatisch zugeordnet bekommt. *Self* gibt das Objekt an, mit dem die Methode arbeitet. Der Compiler stellt jedem Objektelement, das Sie innerhalb der Methode verwenden, automatisch diesen *Self*-Parameter voran. Jede Methode ist also intern so organisiert:

```
procedure TForm1.
  Methode(self: TForm1; ... hier folgen die sichtbaren Parameter);
begin
  with self do begin
    ... hier kommen die sichtbaren Anweisungen
  end;
end;
```

Sie finden *self* weder unter den reservierten Wörtern noch unter den Standarddirektiven, können diesen Namen also uneingeschränkt für eigene Variablen verwenden. Wenn Sie *keine* Variable mit dem Namen *self* deklarieren, können Sie die vordefinierte Version von *self* in Ihrem Code ansprechen, z.B.:

```
with FarbigeKomponente do
  Color := self.Color;
{ ist das gleiche wie: }
FarbigeKomponente.Color := Color;
```

Zugriff von außen auf die Objektelemente

Wenn nun Programmcode von außen auf Elemente des Objekts zugreifen soll, müssen Sie zuerst die Objektinstanz angeben und das gewünschte Element hinter einem Punkt anschließen, um z. B. die *Execute*-Methode aufzurufen:

```
EingabeDialog.Execute;
```

Diese Schreibweise können Sie immer verwenden, wenn Sie in Formularmethoden auf die Komponenten des Formulars zugreifen, eine Alternative dazu ist die *with*-Anweisung (siehe Kapitel 2.5.1).

2.2.5 Properties

Die Properties sind eine der besonders wichtigen sprachlichen Neuerungen von Delphi gegenüber Borland Pascal. Sie scheinen auf den ersten Blick das OOP-Konzept, die Daten eines Objekts nach außen nicht zu zeigen (geschweige denn, sie von dort verändern zu lassen), zu durchkreuzen. Wie Sie in Kapitel 2.2.5 jedoch genauer sehen können, sind die Daten in den Properties mit speziellen Methoden verbunden, die diese Daten verändern. Jedes Mal, wenn Sie einem Property einen Wert zuweisen oder das Property nur lesen (einer anderen Variable zuweisen), wird eine Methode des Objekts aufgerufen, so dass dieses von jeder Änderung unterrichtet wird. Tatsächlich sind die Properties also nur vereinfachte Methodenaufrufe. Ein Beispiel soll den Unterschied erläutern:

Früher hätte man von außen die Farbe eines Objekts so ändern können:

```
Object.Color := clRed;
Object.Paint; (* neue Farbe am Bildschirm ausgeben *)
```

Besser wäre es gewesen, die Objektdaten (*Color*) nicht direkt zu verändern, sondern stattdessen eine Methode aufzurufen:

```
Object.SetColor(clRed);
(* Object.Paint könnte von SetColor
   automatisch aufgerufen werden *)
```

Handelt es sich bei *Color* jedoch um ein Property, weisen Sie ihm einfach einen Wert zu wie im oben gezeigten, damals noch schlechten Beispiel:

```
Object.Color := clRed;
```

Die Methode, die bei dieser Zuweisung automatisch aufgerufen wird, sollte die Farbänderung auch am Bildschirm sichtbar machen. Bei den Komponentenklassen der CLX können Sie sich jedenfalls darauf verlassen, dass diese Aufgaben wie das Aktualisieren der Bildschirmanzeige von alleine erledigen.

Eigene Properties deklarieren R144

Properties eignen sich auch außerhalb von Komponenten in Formularen und eigenen Nicht-Komponenten-Klassen hervorragend zur Vereinfachung der Programmierung. Properties werden an derselben Stelle einer Klassendeklaration deklariert wie Methoden, dürfen also erst nach allen Variablen desselben Abschnitts genannt werden.

Die Deklaration eines Properties sieht wie folgt aus:

```
property Color: TColor read GetColor write SetColor;
```

Dabei ist *Color* der Name, unter dem Sie das Property ansprechen, die dahinter stehenden *read* und *write*-Direktiven geben die Methoden an, die beim Lese- bzw. beim Schreibzugriff aufgerufen werden. Sie können eine der beiden Direktiven auch weglassen, um ein Property zu erhalten, das nur gelesen bzw. nur beschrieben werden kann.

Die Lesemethode eines Properties ist eine Funktion ohne Parameter, deren Ergebnistyp mit dem Typ des Properties übereinstimmen muss, während die Schreibmethode eine Prozedur ist, die eine Variable des Propertytyps als Parameter hat, beispielsweise:

```
function TEineKlasse.GetColor: TColor;
begin
  Result := FColor;
end;

procedure TEineKlasse.SetColor(x: TColor);
begin
  FColor := x;
  Invalidate;
end;
```

Wie in diesem Beispiel liegen einem Property oft Variablen desselben Typs zugrunde und es ist die Aufgabe der Property-Methoden, besondere Aktionen mit dem Lesen und Schreiben dieser Variable zu verknüpfen. Meistens wird diese Variable aus einem vorangestellten »F« und dem Namen des Properties gebildet (in diesem Beispiel *FColor*).

Variablen in Property-Verkleidung

Alternativ zu den Methoden können Sie in den *read*- und *write*-Direktiven auch Variablen angeben, die direkt gelesen oder geschrieben werden können. Im obigen Beispiel, in dem die Methode *GetColor* nur die Variable *FColor* ausliest, wäre es beispielsweise sinnvoller, das Property wie folgt zu deklarieren:

```
{ alle für das Property notwendigen Deklarationen: }
  FColor: TColor;
  procedure SetColor(x: TColor);
  property Color: TColor; read FColor write SetColor;
```

Property-Beispiele

ür vollständige und größere Property-Beispiele sei an dieser Stelle auf andere Teile dieses Buchs verwiesen, hier folgt eine Liste ohne Vollständigkeitsanspruch:

- Formular *TDocumentForm*: *ZoomFactor* (Kapitel 6.6.2) und *FileName* (Kapitel 6.2.3)
- Klasse *TGraphicDoc*: *BrushColor* und *BrushName* (Kapitel 6.3.2)
- Klasse *THistoryList*: *MenuItem* (Kapitel 4.6.2)
- Klasse *THistoryList*: *MaxLen* (Kapitel 4.1.2)

Darüber hinaus definieren natürlich die in Kapitel 8 entwickelten Komponenten Properties, die Sie auch zur Entwurfszeit im Objektinspektor editieren können.

Array-Properties

Ein Array-Property funktioniert nach außen hin so wie ein normales Array, Sie greifen also beispielsweise mit

```
Colors[0] := clWhite;
```

auf das nullte Element des Properties *Colors* zu. Bei der Deklaration geben Sie die Bereiche einer oder mehrerer Dimensionen nicht wie in einem normalen Array an, sondern schreiben jede Dimension des Arrays wie eine Variablendeklaration (z. B. *I: Byte*). Auch wird der Typ eines Array-Properties nicht wie bei Arrays hinter einem *of*, sondern hinter einem Doppelpunkt angegeben:

```
property Colors[i: Byte]: TColor read GetColor write SetColor;
property DreiD[x, y, z: Integer]: Elementtyp; read Read3D write Set3D;
```

Die Schreib- und Lesemethoden sind nun jeweils um einen oder mehrere Index-Parameter erweitert, für die obigen Properties könnten sie folgendermaßen deklariert sein:

```
function GetColor(i: Byte): TColor;
procedure SetColor(i: Byte; val: TColor);
function Get3D(x, y, z: Integer): Elementtyp;
procedure Set3D(x, y, z: Integer; val: Elementtyp);
```

Im schon erwähnten Kapitel 6.3.2 gibt es mit dem von *TList* überschriebenen *Items*-Property auch ein Array-Property-Beispiel.

Das Standard-Property

Wenn Sie die Deklaration eines Array-Properties mit der Direktive *default* abschließen, wie beispielsweise das Property *TList.Items*, wird dieses Property zum *Standard-Property* der Klasse:

```
property Items[Index: Integer]: Pointer read Get write Put; default;
```

Für die Klasse *TList* bedeutet das beispielsweise, dass Sie ein ganzes Objekt dieser Klasse behandeln können wie dieses Array, ohne das Property extra nennen zu müssen. Die Ausdrücke *Liste.Items[i]* und *Liste[i]* haben daher dieselbe Wirkung. Es versteht sich von selbst, dass jede Klasse nur ein derartiges Standard-Property haben kann.

Eine Methode – mehrere Properties

Indizierte Properties sind vergleichbar mit Array-Properties, bei denen Sie nicht über einen Index, sondern über einen Namen auf die einzelnen Elemente zugreifen. Die Properties *Left*, *Right*, *Top* und *Bottom* der Klasse *TGraphicElement* aus Kapitel 6.3.2 beispielsweise sind wie folgt deklariert:

```
property Left: Integer index 1 read GetCoord;
property Right: Integer index 2 read GetCoord;
property Top: Integer index 3 read GetCoord;
property Bottom: Integer index 4 read GetCoord;
```

Alle vier Properties werden mit derselben Methode gelesen, wobei diese die Properties über den Index unterscheidet. Der Compiler fügt automatisch den passenden Index in den Methodenaufruf ein, wenn Sie eines dieser Properties ansprechen:

```
function TGraphicElement.GetCoord(Index: Integer): Integer;
begin
  with Points do
    case Index of
      1: Result := Left;
      2: Result := Right;
      3: Result := Top;
      4: Result := Bottom;
    end;
end;
```

Gleichermaßen funktionieren die Schreibprozeduren für ein solches Property, die natürlich wieder einen zusätzlichen Parameter für den Wert benötigen. Ein weiteres Beispiel für eine Methode, die mehrere Properties behandelt, gibt Kapitel 8.5.3 mit *SetIniData*.

Vererbung von Properties

Sie können Properties in abgeleiteten Klassen überschreiben, wobei Sie den Zugriffsschutz, in dem sie deklariert sind, sowie die Lese- und Schreibmethoden verändern können. Diese und einige weitere Variationen der Properties, die bei der Komponentenprogrammierung verwendet werden, sind in den Kapiteln 8.3.1 und 8.4.3 besprochen.

2.2.6 Zugriffsbeschränkungen

Durch zusätzliche Angaben in der Klassendeklaration können Sie bestimmte Elemente der Klasse vor dem Zugriff von außen schützen. Sie haben die folgenden vier Schutzebenen zur Verfügung (zur Anwendung der Direktiven siehe *Die Klassendeklaration*, Kapitel 2.2.3):

- *public* (öffentlich) lässt die Elemente eines Objekts vollkommen ungeschützt und erlaubt es, dass von außen darauf zugegriffen wird. Optimal ist es, wenn nur Methoden und Properties, aber keine Daten öffentlich sind.
- *protected* schützt die Elemente vor dem Zugriff von außen, was bedeutet, dass beispielsweise eine Variable, die im *protected*-Bereich der *Button*-Komponente deklariert ist, nicht von einer Ereignisbearbeitungsmethode des Formulars aus angesprochen werden kann. Sie dürfen die mit *protected* geschützten Elemente einer Klasse jedoch in von dieser abgeleiteten Klassen ohne Einschränkung verwenden.
- Die Schutzebene *private* versagt sogar abgeleiteten Klassen die Verwendung der so gekennzeichneten Elemente, so dass die besitzende Klasse die alleinige Kontrolle darüber hat, was mit diesen Elementen geschieht.
- Schließlich gehört noch eine vierte Direktive in die Reihe von *private*, *protected* und *public*: Die Schutzebene *published* (veröffentlicht), die noch niedriger als *public* ist, wird nur für Komponenten benötigt, um deren Properties sogar noch außerhalb des Programms sichtbar zu machen, nämlich für die Kylix-IDE bzw. deren Objektinspektor. Standardmäßig sind alle Elemente einer Klasse dieser Schutzebene zugeordnet – solange Sie also keine der drei anderen Schutzebenen angeben, bleiben alle Elemente veröffentlicht. In der Formulardeklaration sind dies beispielsweise alle Elemente, die Kylix automatisch hinzufügt.

ontrolle der Schutzbeschränkungen

Die Schutzebenen *private* und *protected* sind allerdings nicht ganz so streng, wie es zuerst aussieht: Sie gelten nur für den Zugriff aus einem anderen Modul heraus. Wenn Sie in einer Klasse *private*- und *protected*-Elemente deklarieren, können Sie aus anderen Klassen beliebig auf diese Elemente zugreifen, wenn sich diese anderen Klassen in derselben Unit befinden. Mit anderen Worten sind also alle Klassen eines Moduls miteinander befreundet, als wären sie in C++ als *friend* deklariert.

Da sich alle Klassen der CLX in eigenen Modulen befinden, können Sie auf die privaten Elemente der Komponenten gar nicht zugreifen, auf die *protected*-deklarierten Elemente nur innerhalb eigener, abgeleiteter Komponentenklassen.

2.2.7 Vererbung

Vielleicht fragen Sie sich, wo denn in der Formulardeklaration die Properties des Formulars zu finden sind. Die Antwort ist: Sie sind alle in der ersten Deklarationszeile versteckt, in dem Fragment *TForm1 = class (TForm)*, denn auf diese Weise erbt *TForm1* alle Elemente, die schon in der Klasse *TForm* deklariert sind (allerdings dürfen Sie von *TForm1* aus nicht auf die mit *private* geschützten Klassenelemente zugreifen):

In *TForm* implementiert die CLX die grundlegenden Eigenschaften und Fähigkeiten eines Formulars, daher ist ein leeres Formular, zu dem Sie noch keine Ereignisbearbeitungsmethoden hinzugefügt haben, sofort lauffähig. Durch die Vererbung müssen Sie das Formular in eigenen Anwendungen nicht vollkommen neu implementieren, sondern können die bestehende Formularklasse erweitern.

Natürlich gibt es auch Bibliotheken, die nicht objektorientiert sind und mit denen Sie ein Fenster ebenfalls nicht völlig neu implementieren müssen, jedoch vereinfachen die Eigenschaften des OOP die Nutzung bestehenden Codes und dessen Erweiterung enorm und machen sie elegant und leicht verständlich.

Der Prozess, aus einer bestehenden Klasse eine neue zu machen, wird als *Ableiten* bezeichnet. Im obigen Beispiel ist also *TForm1* von *TForm* abgeleitet. Die Klasse, von der abgeleitet wird, heißt *Eltern-* oder *Basisklasse*. Der Prozess des Ableitens ist auch in mehreren Ebenen wiederholbar. Auf diese Weise lassen sich Klassenhierarchien aufbauen, in denen es allgemeine Klassen gibt, die hauptsächlich als Basisklassen für speziellere Klassen dienen. Alle davon abgeleiteten Klassen müssen nur noch den ihnen eigenen Funktionsanteil realisieren und können sich für den Funktionsbereich, in dem Übereinstimmung herrscht, auf die gemeinsame Basisklasse berufen.

Ein sehr gutes Beispiel für eine Klassenhierarchie ist die CLX selbst. So implementiert beispielsweise die CLX-Klasse TWidgetControl alle wichtigen Fähigkeiten, die eine Komponente, die mit einem Qt-Fensterelement verbunden ist, haben muss. Alle visuellen Steuerelement-Komponenten, die von dieser Klasse abgeleitet sind, erben bereits ohne eigenes Zutun die Verbindung zum Qt-Fensterelement. Sie können sich daher auf ihre eigenen speziellen Funktionen konzentrieren.

2.2.8 Vorwärtsdeklaration von Klassen

Damit zwei Klassen sich auch gegenseitig benutzen können, erlaubt Object Pascal die Vorwärtsdeklaration von Klassen. Um dem Compiler beispielsweise die Klasse *TDocument* bekannt zu machen, genügt die folgende Deklaration:

```
type
  TDocument = class;
```

Von da an können Sie Objekte dieser Klasse deklarieren. Die gegenseitige Benutzung zweier Klassen sähe dann so aus:

```
type
  TDocument = class; { Klasse für ein Dokument }
  TItem = class { Klasse für einen Bestandteil des Dokuments }
    { Der Bestandteil weiß, in welchem Dokument er sich befindet: }
    ParentDocument: TDocument;
  end;
  TDocument = class
    { Das Dokument kennt z.B. seinen ersten Bestandteil: }
    FirstItem: TItem;
  end;
```

2.3 Der Lebenslauf von Objekten

Wir untersuchen nun die Vorgänge, die zur Laufzeit des Programms mit einem Objekt passieren (können). Dazu gehören Initialisierung und Löschung mit Konstruktoren und Destruktoren sowie das sehr wichtige OOP-Thema der Polymorphie, bei der sich erst zur Programmlaufzeit entscheidet, welche Methode aufgerufen wird.

2.3.1 Initialisierung von Objekten: Konstruktoren

Wie Sie eine Instanz einer Klasse anlegen, zeigt Kylix Ihnen ebenfalls schon am Beispiel des Hauptformulars: Kylix erzeugt automatisch eine Variable des neuen Formulartyps, die den von Ihnen im Formular-Eigenschaftsfeld *Name* angegebenen Namen erhält, per Voreinstellung also den Namen *Form1*:

```
var
  Form1: TForm1;
```

In Verbindung mit dieser Variablen existiert genau ein Fenster der Klasse *TForm1*.

Wofür der von Kylix automatisch erzeugte Code kein gutes Beispiel gibt, ist die Initialisierung eines Objekts. Nachdem Sie eine Objektvariable wie oben gezeigt deklariert haben, können Sie sie noch nicht benutzen, ohne einen Laufzeitfehler oder eine Schutzverletzungs-Exception (*EAccessViolation*) zu erzeugen, denn jedes Objekt muss zuerst initialisiert werden, indem Sie seinen Konstruktor aufrufen (für Borland Pascal-Gewöhnte: wirklich *jedes* Objekt, das einen *class*-Typ hat, denn diese Objektvariablen bestehen nach ihrer Deklaration intern immer aus einem uninitialisierten Zeiger).

In den meisten Fällen sorgt jedoch die CLX für die Initialisierung der Objekte. Das Formular selbst wird in der Methode *CreateForm* initialisiert, die in der automatisch erzeugten Projektdatei aufgerufen wird. Zur Initialisierung des Formulars gehört auch die Initialisierung all seiner Komponenten, und diese Komponenten sind ja bereits alle Objekte, die in einer einfachen Kylix-Anwendung vorkommen.

Wenn Sie jedoch selbst dynamisch Komponenten oder andere Objekte erzeugen oder Objekte verwenden möchten, die unabhängig von einem Formular sind, müssen Sie Ihre Objekte selbst initialisieren.

Wie schon erwähnt, benötigen Sie dazu einen neuen Konstruktor. In diesem erhält das Objekt Gelegenheit, sich auf die Wirren der Programmlaufzeit vorzubereiten und sich selbst in einen angemessenen Anfangszustand zu versetzen. Das Mindeste, was im Konstruktor geschieht, ist in jedem Fall die Reservierung von Speicher, denn alle *class*-Objekte sind dynamisch und belegen erst nach dem Konstruktoraufruf nennenswerte Speichermengen. Der Konstruktor des Formulars heißt *Create* und ist von *TForm* geerbt.

Bei Formularen ist es normalerweise jedoch nicht notwendig, einen eigenen Konstruktor zu schreiben, denn das Initialisieren des Fensters ist auch ein Ereignis, das Sie im Objektinspektor unter dem Namen *OnCreate* finden. Verknüpfen Sie mit diesem eine Methode, in der Sie Ihre Initialisierungen durchführen.

Konstruktoren aufrufen

Konstruktoren haben nicht nur die Aufgabe, das Objekt zu initialisieren, sondern auch, das Objekt überhaupt zu erzeugen (Speicher für es zu reservieren). Um beispielsweise das Objekt *DynamicForm* zu erzeugen, können Sie daher nicht die folgende Anweisung verwenden:

```
DynamicForm.Create(ParentForm); { keine Objekterzeugung! }
```

Diese Anweisung ruft den Konstruktor *Create* mit einem Objekt (*DynamicForm*) auf, das noch gar nicht existiert. Um ein Objekt neu zu erzeugen, müssen Sie der Objektvariablen (hier *DynamicForm*) ein neues Objekt *zuweisen* (was in Borland Pascal über die *new*-Anweisung gemacht wurde). Das neue Objekt erhalten Sie, indem Sie den Konstruktor nicht auf die Objektvariable, sondern auf die Klasse anwenden. *TForm1.Create(ParentForm)* reserviert den Speicher für ein neues dynamisches Objekt, initialisiert das Objekt und liefert es als Ergebnis zurück, so dass Sie es der Objektvariablen zuweisen können:

```
DynamicForm := TForm1.Create(ParentForm);
```

Konstruktoraufruf ohne Objekterzeugung

Sie können einen Konstruktor auch wie oben als Element eines bestehenden Objekts aufrufen (*DynamicForm.Create*). In diesem Fall wird kein neues Objekt erstellt, sondern der Konstruktor erhält *DynamicForm* als *self*-Parameter. In diesem Fall muss das Objekt also schon vorher mit einem Konstruktor initialisiert worden sein. Normalerweise ist es nur dann notwendig, auf solche Weise einen Konstruktor aufzurufen, wenn Sie selbst einen neuen Konstruktor anlegen und in diesem den geerbten Konstruktor

aufrufen. In diesem Fall müssen Sie den Aufruf jedoch mit Hilfe des Schlüsselworts *inherited* durchführen, wie der nächste Abschnitt zeigen wird.

Eigene Konstruktoren schreiben

Wenn Sie eigene nichtvisuelle Objektklassen schreiben, benötigen Sie statt der Bearbeitung des *OnCreate*-Ereignisses einen eigenen Konstruktor. Sie deklarieren diesen wie eine Methode, verwenden allerdings das Schlüsselwort *constructor*:

```
type
  TGraphicElement = class(TPersistent)
    constructor Create(InitRect: TRect);
     { der Name "Create" ist nicht vorgeschrieben }
```

Während die meisten Konstruktoren der CLX die Besitzerkomponente als Parameter erwarten, können Sie Ihrem Konstruktor die Parameter geben, die Sie zur Objektinitialisierung benötigen. Im obigen Beispiel ist das ein Rechteck, dessen Koordinaten zur Initialisierung des *TGraphicElement*-Objekts verwendet werden sollen.

Innerhalb eines Konstruktors müssen Sie den Konstruktor der Basisklasse aufrufen, damit auch diese sich richtig initialisieren kann:

```
constructor TGraphicElement.Create(InitRect: TRect);
begin
  inherited Create;
  ...
```

Der Aufruf des geerbten Konstruktors erzeugt kein neues Objekt, da er ohne die oranstellung der Klasse stattfand (siehe voriger Abschnitt *Konstruktoraufruf ohne Objekterzeugung*).

Hinweis: Wenn die Klasse, von der Sie eine neue Klasse ableiten, einen virtuellen Konstruktor deklariert, sollten Sie nicht irgendeinen neuen Konstruktor schreiben, sondern den virtuellen Konstruktor der Basisklasse überschreiben. Virtuelle Methoden und Konstruktoren werden später in diesem Kapitel behandelt, das Überschreiben virtueller Konstruktoren ist beispielsweise bei der Entwicklung eigener Komponentenklassen notwendig (siehe z.B. Kapitel 8.5.2, Konstruktor *TScrollBoxEx.Create*).

Hinweis für Borland-Pascal-Umsteiger: In Borland Pascal hätten Sie statt *inherited Create* auch *TPersistent.Create* schreiben können, um den geerbten Konstruktor aufzurufen, ohne dass dadurch ein weiteres neues Objekt konstruiert worden wäre. In Object Pascal würde der Aufruf *TPersistent.Create* ein neues *TPersistent*-Objekt erzeugen und als Funktionsergebnis zurückliefern (wobei dieses Ergebnis verloren gehen würde, weil Sie es keiner Variablen zugewiesen haben). In Object Pascal *müssen* Sie also mit der Direktive *inherited* arbeiten.

Der oben gezeigte Unterschied zwischen dem Aufruf *TClass.Create* und *ObjectVar.Create* wird von dem Code, den der Compiler erzeugt, automatisch berücksichtigt. Wenn die erste Zeile Ihres Konstruktors erreicht ist, ist der Speicher für das Objekt in jedem Fall bereits reserviert. Falls mit dem Konstruktoraufruf ein neues Objekt erstellt wurde, ist dieser Speicher außerdem komplett mit 0 initialisiert. Sie brauchen daher Variablen, deren binäre Form nur aus Nullen bestehen soll (z.B. Zahlen, die 0 sind, leere Mengen und *False*-Werte), im Konstruktor nicht einzeln zu initialisieren.

2.3.2 Aufräumen mit Destruktoren

Das Gegenstück zu den Konstruktoren bilden die Destruktoren. Grundsätzlich sollte jedes Objekt, für das in irgendeinem Teil des Programms Speicher reserviert wurde, an einer anderen Stelle wieder freigegeben werden.

In der Praxis müssen Sie Destruktoren jedoch noch seltener von Hand aufrufen als Konstruktoren. Selbstverständlich gibt die CLX alle Objekte wieder frei, die sie selbst erzeugt hat (also die Komponenten des Formulars und das Formular selber). Darüber hinaus gibt sie auch die von Ihnen erzeugten dynamischen Objekte frei, wenn es sich um Komponenten handelt und Sie diese in die Besitzhierarchie eines Formulars eingefügt haben. Wenn nämlich eine Komponente (oder ein Formular) gelöscht wird, gibt die CLX alle von dieser Komponente besessenen Komponenten frei, indem sie ihren Destruktor aufruft.

Sie müssen also nur für die Objekte einen Destruktor aufrufen, für die Sie einen Konstruktor aufgerufen haben, die Sie aber nicht in andere Komponenten, die von der CLX verwaltet werden, eingefügt haben.

Destruktoren aufrufen

Wenn die genannte Voraussetzung nicht zutrifft und Sie das Objekt selber freigeben müssen, können Sie den Destruktor direkt aufrufen:

```
DynamicObject.Destroy;
```

Empfehlenswerter ist es jedoch, stattdessen die Methode *Free* aufzurufen. Diese testet, ob das Objekt überhaupt initialisiert wurde, und ruft den Destruktor nur dann auf, wenn das der Fall ist. Sie können auch selbst überprüfen, ob ein Objekt bereits initialisiert wurde, indem Sie die Standardprozedur *Assigned* verwenden:

```
{ entspricht dem Aufruf von "Free": }
if Assigned(DynamicObject)
   then DynamicObject.Destroy;
```

Assigned funktioniert nur dann zuverlässig, wenn alle nicht-initialisierten Objekte auch am Wert *nil* erkennbar sind. Objekte, die Teil eines Formulars oder anderen Objekts sind, werden immer mit *nil* vorbelegt, da der Speicher eines Objekts mit Nullen initia-

lisiert wird. Nachdem Sie aber ein Objekt mit *Free* freigegeben haben, wird es nicht automatisch auf *nil* zurückgesetzt. Dies müssen Sie manuell erledigen, falls Sie möglicherweise später mit *Assigned* abfragen wollen, ob das Objekt initialisiert ist:

```
DynamicObject := nil;
```

Destruktoren schreiben

Um einen Destruktor selbst zu schreiben, deklarieren Sie eine Methode mit dem Schlüsselwort *destructor*. Die Sprache Object Pascal erlaubt hier zwar wieder einen eliebigen Namen, da jede Klasse in Object Pascal aber ein Nachfahre der vordefinierten Klasse *TObject* ist, empfiehlt es sich jedoch, den schon von *TObject* vordefinierten virtuellen Destruktor *Destroy* zu überschreiben. In jedem Fall sollte ein Destruktor am Schluss auch einen geerbten Konstruktor aufrufen, im Normalfall also *Destroy*:

```
destructor TGraphicElement.Destroy;
begin
  ...
  inherited Destroy;
end;
```

Destruktoren haben weniger zu tun als Konstruktoren, da sie oft nur die Speicherreservierungen rückgängig machen, die im Konstruktor explizit vorgenommen wurden. Wenn Ihr Konstruktor nur einige Variablen mit Standardeinstellungen belegt, benötigen Sie oft keinen eigenen Destruktor. Ausnahmen von dieser Regel: Wenn andere Methoden des Objekts Speicher zur privaten Verwendung innerhalb des Objekts reservieren, ihn aber selbst nicht freigeben, sollte dies spätestens im Destruktor nachgeholt werden.

Auch für andere Arten von Systemressourcen wie z. B. geöffnete Dateien gilt, dass sie (spätestens) im Destruktor freigegeben werden sollten. Wenn Sie die Klassen der CLX verwenden, sind diese Systemressourcen jedoch meistens in Objekten gekapselt, so dass Sie sich um die Ressourcen selbst nicht zu kümmern brauchen (Dateien können B. mit *TStream*-Objekten verwaltet werden).

Automatischer Destruktoraufruf

Object Pascal unterstützt zwar nicht die automatische Konstruktion und Destruktion von Objekten wie C++, in einem speziellen Fall kommt es jedoch zu einem automatischen Aufruf des Destruktors: Wenn innerhalb des Konstruktors eine Exception auftritt, wird automatisch der Destruktor aufgerufen, um den Speicher wieder eizugeben. (Dieser Mechanismus ist nur aktiv, wenn der Konstruktor auf eine Klasse (genauer: Klassenreferenz) angewendet wurde, um ein neues Objekt zu initialisieren.)

Ihr Destruktor sollte auf den Fall, dass der Konstruktor durch eine Exception unterbrochen wird, vorbereitet sein. Im folgenden Beispiel könnte es Probleme geben:

```
constructor DemoObject.Create;
begin
  inherited.Create;
  OpenFile;
  GetMem(DynArray, 20000);
end;

destructor DemoObject.Destroy;
begin
   { der folgende Code ist nicht sicher,
    wenn der Konstruktor durch eine Exception unterbrochen wird:
  CloseFile;
  FreeMem(DynArray, 20000);
  inherited Destroy;
end;
```

Falls in *Create* während der Operation *OpenFile* eine Exception auftritt, wird der Konstruktor sofort abgebrochen und der Destruktor aufgerufen. Die Variable *DynArray* ist zu diesem Zeitpunkt noch nicht initialisiert. Der Aufruf von *FreeMem* würde also nicht erforderlich sein und würde sogar zu einem Fehler führen. Stattdessen könnte der Destruktor überprüfen, ob die Initialisierungen überhaupt vorgenommen worden sind:

```
destructor DemoObject.Destroy;
begin
  if FileIsOpen then
    CloseFile;
  if Assigned(DynArray) then
    FreeMem(DynArray, 20000);
  inherited Destroy;
end;
```

Freigabe von aktiven Objekten *R34*

Problematisch ist es auch, Objekte zu löschen, die noch aktiv sind. So können Sie in einer *OnClick*-Bearbeitung für einen Schalter diesen Schalter nicht löschen, da die CLX den Schalterdruck eventuell noch weiter bearbeiten will und nicht damit rechnet, dass der Schalter plötzlich verschwindet.

Ebenso wenig dürfen Sie die Methode *Free* eines Formulars innerhalb einer Ereignisbearbeitungsmethode dieses Formulars aufrufen. Es ist jedoch möglich, aktive Formulare zu schließen, indem Sie die Methode *Close* oder (in Spezialfällen) *Release* verwenden.

2.3.3 Polymorphie durch virtuelle Methoden

Im Laufe dieses Kapitels werden wir noch des Öfteren auf Spracheigenschaften stoßen, die von der Komponentenbibliothek aufgewertet – oder die im Gegenteil durch die Nutzung dieser Bibliothek weniger wichtig geworden sind. Die virtuellen Methoden erhalten in Object Pascal gleich mehrfach Konkurrenz: Zum einen können Sie Metho-

den explizit als *dynamic* deklarieren, was jedoch am hier beschriebenen Prinzip nichts ndert, zum anderen macht die CLX bei den Events massiven Gebrauch von den neuen Methodenzeigern (siehe Kapitel 2.5.5).

Methodenzeiger oder virtuelle Methoden?

Das Ergebnis dieses Konkurrenzkampfs ist, dass die virtuellen Methoden für den Entwurf eines Formulars keine Rolle mehr spielen. Stattdessen werden die mit Hilfe der visuellen Tools einfacher handzuhabenden Ereignisbearbeitungsmethoden verwendet. In allen anderen Bereichen bleiben die virtuellen Methoden nahezu unverzichtbar. So brauchen Sie als Komponentenprogrammierer ein fundiertes Wissen über die Polymorphie mit virtuellen oder dynamischen Methoden. Auch wenn Sie fernab jeglicher Komponenten eigene Objektklassen entwickeln wollen, sind virtuelle Methoden meist erheblich besser geeignet als Methodenzeiger. Der große Vorteil der Methodenzeiger bei der visuellen Programmierung resultiert auch erst daraus, dass Kylix Ihnen bei ihrer an sich umständlichen Verwaltung eine Menge Routinearbeit abnimmt.

Zuweisungskompatibilität von Objekten

Die erste Voraussetzung für die Polymorphie (Vielgestaltigkeit) ist, dass ein Objekt nicht nur zu den Objekten kompatibel ist, die von derselben Klasse sind, sondern auch zu den Objekten, die eine von deren Vorfahr-Klassen haben.

Die Kompatibilität gilt nur in einer Richtung: Sie können einer Variablen einer bestimmten Klasse beliebige Objekte von abgeleiteten Klassen zuweisen:

```
var
  Object: TObject;  { TObject ist die oberste aller Klassen }
  Component: TComponent; { TComponent ist abgeleitet von TObject. }
  Form: TForm;  { TForm ist indirekt abgeleitet von TComponent. }
  Button: TButton; { TButton stammt ebenfalls von TComponent ab. }
...
Component := Form;
Component := Button;
Object := Component;
```

Da Formulare und Schalter alle Fähigkeiten von *TComponent* erben, können sie anstelle eines *TComponent*-Objekts (hier die Variable *Component*) verwendet werden. Da außerdem *TComponent* eine Nachfahr-Klasse von *TObject* ist, kann sie alles tun, was ein *TObject* auch tun kann, daher ist auch die letzte Zuweisung des obigen Beispiels erlaubt. Der umgekehrte Weg ist nicht möglich:

```
Form := Object; { Typ-Fehler }
```

on einem Formular werden sehr viele spezielle Dinge erwartet, eines davon ist z.B., dass es auf dem Bildschirm sichtbar ist. Da *Object* aber irgendein Objekt sein kann, ist

es sehr unwahrscheinlich, dass es allen Anforderungen der Klasse *TForm* genügt. Für die Zuweisung an eine *TForm*-Variable kommen also nur Objekte von *TForm* oder von davon abgeleiteten Klassen in Frage.

Das Ableiten einer Klasse von einer anderen führt also nicht nur dazu, dass die abgeleitete Klasse die Elemente der Basisklasse erbt, sondern es führt zu Verwandtschaftsverhältnissen, die bei der Polymorphie eine große Rolle spielen, wie wir gleich sehen werden.

Hinweis: In diesem Zusammenhang ist es wichtig, dass Objektvariablen nur Zeiger sind. Bei den Zuweisungen im obigen Beispiel werden nur diese Zeiger kopiert, die Objekte selbst werden nicht verändert. Andernfalls würden ja einige Daten verloren gehen, wenn Sie z.B. versuchten, den relativ kleinen Speicherbereich eines *TComponent*-Objekts mit einem umfangreichen *TForm*-Objekt zu füllen, das ja außer den von *TComponent* geerbten Daten viele weitere Daten enthält.

Rückblick auf die statischen Methoden

Zunächst einmal ist festzustellen, dass jede Methode, Prozedur, Funktion (und jeder beliebige Programmabschnitt) eine Adresse im Speicher des Computers hat. Die Methoden, die in den Beispielen des Kapitels 2.2 vorgekommen sind, waren alle statisch. Wenn Sie eine solche Methode aufrufen, generiert der Compiler Maschinenbefehle für einen direkten Sprung zu dieser Adresse wie bei einem normalen Prozedur- oder Funktionsaufruf. Sobald Vererbung ins Spiel kommt, kann dieses Verhalten jedoch unerwünscht werden und sogar den Sinn der Vererbung gefährden.

Nehmen wir an, Sie haben verschiedene Klassen, die alle auf eine verschiedene Weise arbeiten. Die Arbeit ist in einer Prozedur *Arbeite* definiert. Jede Klasse besitzt eine eigene Prozedur *Arbeite*, die eine individuelle Arbeit verrichtet:

```
type
  AbstrakteKlasse = class(TObject)
  { Konstruktor Create wird von TObject geerbt }
    procedure Arbeite;
  end;
  Klasse1 = class(AbstrakteKlasse) procedure Arbeite; end;
  Klasse2 = class(AbstrakteKlasse) procedure Arbeite; end;
  ...
  Klasse10 = class(AbstrakteKlasse) procedure Arbeite; end;
```

Sie lassen nun den Benutzer Ihrer Anwendung auswählen, welche Klasse er verwenden möchte. Je nach Auswahl des Benutzers erzeugen Sie dynamisch ein Objekt der gewünschten Klasse und weisen es der Variablen *Objekt* zu:

```
var
  Objekt: AbstrakteKlasse;
begin
  case Benutzerauswahl of
    { "Benutzerauswahl" kann z.B. sein: (Sender as TButton).Tag }
     Button1: Objekt := Klasse1.Create;
     Button2: Objekt := Klasse2.Create;
  ...
     Button10: Objekt := Klasse10.Create;
```

Wir nehmen weiter an, dass Sie an einer anderen Stelle die Arbeit der Klasse ausführen wollen, die der Benutzer ausgewählt hat:

```
Objekt.Arbeite;
```

Wie soll der Compiler nun diesen Methodenaufruf in Maschinenbefehle umwandeln, bzw. welche *Arbeiten*-Methode soll er aufrufen? Schließlich kann er nicht wissen, ob *Objekt* zur Laufzeit den Typ *Klasse1*, *Klasse2* oder sonst einen Typen hat. Da *Objekt* als Instanz der Klasse *AbstrakteKlasse* deklariert wurde, ruft der Compiler in diesem Beispiel die Methode *AbstrakteKlasse.Arbeite* auf, obwohl *Objekt* vorher mit einer anderen Klasse initialisiert wurde. Es gibt jedoch eine Lösung für dieses Problem.

Virtuelle Methoden

Wenn Sie bisher nur die Methodenzeiger und Ereignisproperties kennen, werden Ihnen diese sicher als Lösungsmöglichkeit einfallen: Die Basisklasse (*AbstrakteKlasse*) muss lediglich einen Methodenzeiger, z.B. *OnArbeite*, beinhalten, der von den abgeleiteten Klassen entsprechend gesetzt wird. Der obige Methodenaufruf müsste dann mit Hilfe dieses Zeigers stattfinden:

```
Objekt.OnArbeite;
```

Tatsächlich funktionieren die virtuellen Methoden sehr ähnlich wie in diesem Konzept, sind aber viel einfacher zu handhaben: Das Einzige, was Sie im obigen Beispiel ändern müssen, um von der statischen Bindung zur dynamischen überzugehen, ist, in der Basisklasse ein *virtual* hinter den Methodenkopf von *Arbeite* zu schreiben und bei den abgeleiteten Klassen ein *override*:

```
type
  AbstrakteKlasse = class
  { Konstruktor Create wird von TObject geerbt }
    procedure Arbeite; virtual;
  end;
  Klasse1 = class(AbstrakteKlasse)
    procedure Arbeite; override;
  end;
  ...
```

```
.. weiterer Code wie oben ...
..
  Objekt.Arbeite; { ruft immer die richtige Methode auf }
```

Je nachdem, welches Objekt also in der oben abgedruckten *case*-Abfrage in der Variablen *Objekt* gespeichert wurde, wird für die Anweisung *Objekt.Arbeite* jetzt eine der Methoden *Klasse1.Create*, *Klasse2.Create* usw. aufgerufen. Die Methode *AbstrakteKlasse.Create* wird überhaupt nicht mehr verwendet (sie könnte daher als *abstract* deklariert werden, wie einer der folgenden Abschnitte zeigen wird).

Override

Der Vorgang, eine geerbte virtuelle Methode neu zu definieren, wird als *Überschreiben* der geerbten Methode bezeichnet. Die Angabe der Direktive *override* ist zwingend erforderlich. Wenn Sie sie weglassen, erzeugt der Compiler eine eigenständige Methode, die nichts mit der geerbten Methode zu tun hat und die im obigen Beispiel (in dem die Variable *Objekt* von einer abstrakten Klasse ist) nicht aufgerufen worden wäre. Aus der Sicht der neuen Klasse verdeckt eine solche, nicht mit *override* deklarierte Methode die geerbte Methode gleichen Namens.

Dies ist besonders wichtig für Benutzer von Borland Pascal, da es dort das Wort *override* noch nicht gab und virtuelle Methoden *immer* durch gleichnamige Methoden in abgeleiteten Klassen überschrieben wurden. Allerdings gibt Kylix per Voreinstellung eine Warnung aus, wenn Sie eine geerbte virtuelle Methode verdecken.

Verwechseln Sie *override* außerdem nicht mit der neuen Direktive *overload*. *Overload* wird beim Definieren mehrerer gleichnamiger Methoden, die als Alternativen verwendbar sein sollen, benötigt und hat nichts mit Vererbung zu tun (siehe Kapitel 2.5.4).

Reintroduce

Außerdem gibt es durch die Direktive *Reintroduce* noch eine Möglichkeit, eine geerbte virtuelle Methode im oben beschriebenen Sinne zu verstecken, ohne dass der Compiler eine Warnung ausgibt. Fügen Sie dazu einfach den Zusatz *»Reintroduce;«* an das Ende der Methodendeklaration an.

Wenn Sie also in einer abgeleiteten Klasse eine Methode definieren, deren Name mit dem einer geerbten Methode übereinstimmt, sollten Sie entweder *Reintroduce* oder *Override* angeben, damit der Compiler keine Warnung erzeugt und auch einem Menschen beim Lesen des Quelltextes sofort deutlich wird, wie sich die Deklaration der neuen Methode auswirkt. Die *Override*-Direktive ist natürlich nur erlaubt, wenn Parameterliste und Rückgabetyp mit der überschriebenen Methode übereinstimmen, diese Forderung gilt für *reintroduce* nicht.

berschreiben von Methoden und Inherited

Alles, was Sie tun müssen, um den Mechanismus der virtuellen Methoden zu aktivieen, ist, die Methode dort, wo sie zum ersten Mal in der Hierarchie auftritt, als *virtual* zu deklarieren und in allen Ableitungen als *override*. Die Methode, die durch die neue berschrieben wird, sollten Sie in den meisten Fällen innerhalb der neuen aufrufen, damit deren Funktionalität nicht verloren geht. Dazu verwenden Sie den Namen der Methode, stellen ihm aber das Schlüsselwort *inherited* voran:

```
procedure TForm1.WndProc(var Message: TMessage);
begin
  inherited WndProc(Message);
  {... eigene Anweisungen }
```

Hinweis: Während Sie in der Deklaration der Methode *WndProc* die Direktive *override* nicht vergessen dürfen, fehlt diese Direktive hinter dem obigen Methodenkopf. Auch für alle anderen Direktiven gilt, dass Sie sie nur bei der erstmaligen Deklaration, also innerhalb der Klassendeklaration, anzugeben brauchen.

orteile virtueller Methoden gegenüber einfachen Methodenzeigern

Mit Methodenzeigern hätten Sie für jede virtuelle Methode eine zusätzliche Variable deklarieren und im Konstruktor der abgeleiteten Objekte diese auf die richtige Methode setzen müssen (auf diese Weise könnte man dann gleich auf OOP-Sprachen verzichten und das OOP in rein prozeduralen Sprachen wie Pascal oder C simulieren).

irtuelle Methoden haben noch einen weiteren Vorteil: Wenn Sie mehrere virtuelle Methoden in einer Klasse deklarieren, wird nicht für jede davon ein Methodenzeiger in jeder Objektinstanz abgelegt. Stattdessen erhält jede Objektinstanz einen Zeiger auf eine *VMT* (Virtual Method Table), die nur ein einziges Mal für alle Objekte derselben Klasse gespeichert werden muss. In dieser Tabelle sind sämtliche virtuelle Methoden, die geerbt und/oder überschrieben sind, so abgelegt, dass der Compiler sie schnell finden kann. Der Aufruf einer virtuellen Methode läuft damit intern so ab:

```
{ Der Aufruf ... }
   Objekt.Methode(Parameter);
{ ... wird ähnlich behandelt wie folgt: }
   Objekt.VMT^[MethodenIndex](Parameter);
{ allerdings ist der VMT-Zeiger nicht mit
  normalen Zeigern kompatibel und kann nicht
  unter dem Namen "VMT" angesprochen werden. }
```

Der VMT-Zeiger des Objekts wird wie ein normales Datenelement desselben gelesen. Zu dieser VMT-Adresse rechnet das Programm den Index, an dem sich die Adresse der ür dieses Objekt gültigen virtuellen Methode befindet, hinzu. Dieser Tabelleneintrag

wird ausgelesen und das Programm springt an die darin angegebene Adresse (nachdem die Parameter für die Methode inklusive des *self*-Parameters auf den Stack gelegt wurden).

Dynamische Methoden

Dynamische Methoden sind von der Verwendung her mit den virtuellen Methoden identisch. Statt der Direktive *virtual* verwenden Sie einfach die Direktive *dynamic*, um eine dynamische Methode zu erhalten. Alles andere inklusive des Überschreibens der Methode mit *override* bleibt.

Die dynamische Art, die Methodenadressen zu verwalten, ist noch speicherplatzsparender als die VMTs. Der Compiler legt für die dynamischen Methoden jeder Klasse eine eigene Tabelle (DMT) an. Die Platzersparnis resultiert aus der Tatsache, dass dort Methoden, die zwar geerbt, aber nicht überschrieben werden, nicht aufgeführt werden. Vielleicht fragen Sie sich, ob es überhaupt notwendig ist, den geringen Speicherbedarf von VMTs weiter zu reduzieren. Erst bei 100 Klassen mit jeweils 100 Methoden kämen 10000 VMT-Zeiger, also ca. 40 KByte VMTs zusammen, in 32-Bit-Umgebungen immer noch ein Klacks.

Um den entscheidenden Vorteil der dynamischen Methoden klarzumachen, ist ein kleiner Ausflug in die Windows-Programmierung erforderlich: Hier hat das Pendant zur CLX – die VCL – die Aufgabe, zahlreiche Windows-Nachrichten zu bearbeiten, die mit einem Integer-Nachrichtencode identifiziert werden. Und dies bringt den entscheidenden Vorteil der dynamischen Methoden ins Spiel: Sie können mit einer Integer-Konstante verknüpft sein. Jeder dynamischen Methode, die eine Windows-Botschaft bearbeiten soll, wird ein Index zugeordnet, der dem Nachrichtencode der Windows-Nachricht entspricht. Sollen jetzt etwa die fiktiven Nachrichtencodes 5000 und 10000 bearbeitet werden, braucht keine VMT-Tabelle mit 10000 Einträgen angelegt zu werden, sondern es genügt eine DMT mit nur zwei Einträgen.

Polymorphie

Das Gesamtkonzept, das auf der Vererbung und der dynamischen Bindung basiert, heißt *Polymorphie* (Vielgestaltigkeit). Ein polymorphes Objekt ist eine Objektvariable, von der der Compiler nicht weiß, welche Klasse diese Variable zur Laufzeit haben wird. Tatsächlich kann die Klasse sich zur Laufzeit mehrfach ändern, wenn Sie einem polymorphen Objekt Objekte anderer Klassen zuweisen.

Ein polymorphes Objekt kann also in vielen Gestalten auftreten, wie im obigen Beispiel die Variable *Objekt*. Polymorphe Objekte kommen bei verschiedenen Gelegenheiten vor:

- als Properties: z.B. einige Properties, die Stringlisten enthalten, wie etwa *TListBox.Items*. Dieses Property ist als *TStrings* deklariert, basiert aber auf einem Objekt einer davon abgeleiteten Klasse.
- als Parameter von Ereignisbearbeitungsmethoden: z.B. der Parameter *Sender*, der jedes beliebige Objekt transportieren kann.
- als Parameter von Methoden, die Sie aufrufen können: z.B. die Methode *TComponent.InsertComponent*, mit der Sie eine beliebige Komponente zum Besitz einer anderen Komponente hinzufügen können.

Eine weitere Aufwertung erfährt die Polymorphie durch virtuelle Konstruktoren, die in Kapitel 2.3.4 besprochen werden.

Abstrakte Klassen *R154*

In großen Vererbungshierarchien wie der CLX kommt es häufig vor, dass Klassen nur dazu da sind, die Gemeinsamkeiten verschiedener anderer Klassen zu definieren. Dabei können diese Klassen so weit gehen, dass sie zwar eine virtuelle Methode deklarieren, die in allen abgeleiteten Klassen überschrieben wird, dass sie diese selbst jedoch gar nicht implementieren.

Um eine derartige Methode zu deklarieren, die nur als Platzhalter für die Methoden abgeleiteter Klassen dient, können Sie sie mit der Direktive *abstract* als *abstrakte Methode* deklarieren. Dieser Direktive muss entweder *dynamic* oder *virtual* vorangehen, da eine abstrakte Methode, die nicht überschrieben werden kann, keinen Sinn ergibt. Die Methode *Arbeite* der Klasse *AbstrakteKlasse* aus dem obigen Beispiel könnte also auch so deklariert worden sein:

```
procedure Arbeite; virtual; abstract;
```

Diese Deklaration hat den Vorteil, dass Sie die Methode nicht im Implementationsteil der Unit definieren müssen und dass ein Aufruf dieser Methode zu einem Laufzeitfehler mit der Nummer 210 führt, wodurch sichergestellt ist, dass diese Methode nicht versehentlich aufgerufen wird. In Methoden abgeleiteter Klassen, die abstrakte Methoden überschreiben, dürfen Sie daher die geerbte abstrakte Methode auch nicht mit *inherited* aufrufen.

Beispiele für abstrakte Klassen der CLX sind *TStream*, *TStrings* und *TPersistent*.

Laufzeit-Typinformation

Das Konzept der Polymorphie ist sicher faszinierend, vor allem, wenn man vorher nur prozedural programmiert hat und sich langsam an die Vielgestaltigkeit gewöhnt; aber manchmal ist es vielleicht doch etwas unheimlich, dass man überhaupt nicht weiß, welche Klasse sich hinter dem vielgestaltigen Objekt zur Laufzeit tatsächlich verbirgt.

Anders gesagt: Manchmal ist es notwendig, hilfreich oder auch nur sicherer, wenn zur Laufzeit abgefragt werden kann, welchen Typ ein Objekt hat. Dies wäre zwar schon mit einer virtuellen Methode wie *WelchenTypHabeIch* realisierbar, aber die Typinformation, die Sie zur Laufzeit erhalten können, ist noch weitaus umfangreicher. Die Typinformationen in Object Pascal hängen eng mit Klassenreferenzen und Klassenfunktionen zusammen, die das Thema der nächsten beiden Abschnitte sind.

2.3.4 TClass – die Klasse der Klassen

Für die Implementation von virtuellen Konstruktoren müssen in Object Pascal nicht nur die Objekte einer Klasse, sondern auch die Klasse selbst zur Laufzeit ansprechbar sein. Hierzu gibt es die *Klassenreferenz* (in der deutschen Online-Dokumentation finden Sie den Begriff *Objektbezug*, der hier aus Gründen der Verständlichkeit vermieden wird). In manchen anderen OOP-Sprachen finden Sie für die Klassenreferenz die Bezeichnung *Metaklasse*.

Klassenreferenzen *R153*

Die Unit *System* enthält die Deklaration des Typs für eine grundlegende Klassenreferenz:

```
type
  TClass = class of TObject;
```

Verwechseln Sie *TClass* nicht mit anderen Klassen, die mit *class,* aber ohne *of* deklariert werden. Eine Klassenreferenz erhalten Sie, indem Sie einer Variablen den Klassenreferenz-Typ zuweisen:

```
var
  ClassRef: TClass;
```

Sie können sich eine Klassenreferenz als Objekt vorstellen, das verschiedene Informationen über die Klasse speichert, wie z. B. den Namen der Klasse, ihre Basisklasse usw.

Grundsätzlich können Sie eine Klassenreferenz im Programmcode so verwenden wie einen Klassennamen. Die Vorteile einer Klassenreferenz gegenüber einer direkten Klassenangabe liegen wieder einmal in der Polymorphie. In Kapitel 2.3.3 hatten wir das folgende Beispiel, in dem je nach Wahl des Benutzers eine Instanz einer bestimmten Klasse erzeugt werden sollte:

```
case Benutzerauswahl of
  { Benutzerauswahl kann z.B. sein: (Sender as TButton).Tag }
   Button1: Objekt := Klasse1.Create;
   Button2: Objekt := Klasse2.Create;
...
   Button10: Objekt := Klasse10.Create;
```

Unbefriedigend ist dabei, dass wir zwar jedes beliebige Objekt erzeugen können, dass wir aber die Klasse, die der Benutzer ausgewählt hat, nicht in einer Variablen speichern können. Nachdem die Variable *Objekt* erzeugt ist, stehen deren virtuelle Funktionen zwar wie gewünscht zur Verfügung und begründen die Polymorphie des Objekts *Objekt*; wenn nun aber an einer völlig anderen Stelle des Programms ein zweites Objekt derselben (von der Benutzerauswahl abhängenden) Klasse erzeugt werden soll, wäre erneut eine lange *case*-Anweisung wie oben erforderlich (oder eine eigene Prozedur, die diese *case*-Anweisung zum mehrmaligen Gebrauch zur Verfügung stellt).

erwendungsbeispiel

Mit den Klassenreferenzen können Sie das Problem in Object Pascal jedoch auf viel elegantere Weise lösen:

```
type
  AbstrakteKlasseRef = class of AbstrakteKlasse;
var
  GewaehlteKlasse: AbstrakteKlasseRef;
  Objekt: AbstrakteKlasse;
begin
  Objekt := GewaehlteKlasse.Create;
```

Mit der obigen Anweisung erreichen Sie dasselbe wie mit dem langen *case*-Block aus vorhergehenden Beispielen. Natürlich muss *ClassRef* schon vorher den gewünschten Wert erhalten haben. Wenn dieser wieder von der Benutzerauswahl abhinge, wäre dazu immer noch eine *case*-Anweisung wie die obige erforderlich:

```
case Benutzerauswahl of
  Button1: GewaehlteKlasse := Klasse1;
  Button2: GewaehlteKlasse := Klasse2;
usw.
```

Es ist jedoch ein klarer Vorteil, dass diese *case*-Anweisung nur ein einziges Mal erforderlich ist und die Wahl des Benutzers als *ClassRef* gespeichert werden kann. Eine Klassenreferenz ist also auch vorstellbar als eine Variable, die kein Objekt, sondern eine Klasse enthält.

Eigene Klassenreferenzen deklarieren

Sie können – wie im letzten Beispiel geschehen – auch selbst Klassenreferenztypen der Art *class of* deklarieren:

```
type
  TMyClassReference = class of TMyClass;
  AbstrakteKlasseRef = class of AbstrakteKlasse;
```

Eine solche Deklaration dient lediglich dazu, einen festen Namen für einen Klassenreferenztypen zu vergeben. Der Compiler erzeugt sowieso zu jeder Klasse automatisch

eine Klassenreferenz, womit auch der Typ dieser Klassenreferenz existiert. Mit der *class-of*-Deklaration weisen Sie diesem Typen *zusätzlich* einen eigenen Typenbezeichner zu. Einen solchen Typenbezeichner benötigen Sie immer, wenn Sie eine Klassenreferenz-Variable deklarieren wollen, wie in einem früheren Beispiel die Variable *GewaehlteKlasse*. Mit der Methode *TObject.ClassType* können Sie übrigens auch ohne eine solche Deklaration jederzeit in Besitz einer Klassenreferenz für die Klasse eines beliebigen Objekts gelangen, siehe dazu Kapitel 2.3.6.

Virtuelle Konstruktoren

Der Code im Abschnitt *Verwendungsbeispiel* bringt noch eine Gefahr mit sich: Zwar wird mit der Zeile

```
Objekt := GewaehlteKlasse.Create;
```

immer ein Objekt der richtigen Klasse erzeugt, trotzdem wird jedoch nur der von der Klasse *AbstrakteKlasse* bereitgestellte Konstruktor aufgerufen, der an die abgeleiteten Klassen vererbt wird. Falls diese abgeleiteten Klassen wichtige eigene Initialisierungen in eigenen Konstruktoren vornehmen müssen, so sollten diese mit der obigen Zeile ebenfalls aufgerufen werden. Damit das geschieht, muss *AbstrakteKlasse* den *Create*-Konstruktor als *virtual* deklarieren, und die abgeleiteten Klassen (*Klasse1*, *Klasse2* usw., siehe oben) müssten diesen mit *override* überschreiben.

Virtuelle Konstruktoren sind auch nur dann sinnvoll einsetzbar, wenn sie wie in diesem Beispiel über eine Klassenreferenz aufgerufen werden. In allen anderen Fällen steht schon bei der Übersetzung fest, welcher Konstruktor aufgerufen werden soll, weshalb dann die statische Bindung genügt.

Konstante Klassenreferenzen

Der normale Konstruktoraufruf mit Hilfe einer Klassenangabe, der in diesem Kapitel schon mehrfach angesprochen wurde, macht ebenfalls Gebrauch von einer Klassenreferenz, allerdings ist die Klassenreferenz hier nicht in einer Variablen gespeichert, sondern wird direkt über den Klassennamen angegeben:

```
Stringliste := TStringList.Create;
```

Kompatibilität

Für Klassenreferenzen gelten dieselben Kompatibilitätsregeln wie für Objekte. Das bedeutet, dass Sie einer Klassenreferenz vom Typ *TClass* jede beliebige Klassenreferenz zuweisen können, genauso wie Sie einem Objekt vom Typ *TObject* jedes beliebige Objekt zuweisen können, da *TObject* eine Vorfahrenklasse aller anderen Klassen ist.

Die Unit *Forms* deklariert beispielsweise den folgenden Klassenreferenztypen:

```
type
  FormClass = class of TForm;
```

Einer Variablen dieses Typs dürfen Sie nur eine Klasse zuweisen, die von *TForm* abgeleitet ist, beispielsweise die Klasse eines Ihrer Formulare oder die Klasse *TForm* selber.

2.3.5 Klassenmethoden

Eine interessante Erweiterung der OOP-Fähigkeiten früherer Pascal-Versionen stellen die Klassenmethoden dar. Die Bezeichnung *Klassenmethode* meint hier weniger Methoden der Klasse« als eher »Methoden für die Klasse«, also Methoden, die nicht ür spezifische Objekte, sondern für die Klasse selbst aufgerufen werden (C++-Programmierer werden dies von den statischen Elementen der C++-Klassen kennen). Ein ein fiktives Beispiel ist:

```
type
  TDemoClass = class
    class function Arbeitsgeschwindigkeit: Byte; virtual;
  { mit dieser Funktion kann die Klasse versprechen, besonders
    schnell zu arbeiten (falls Rückgabewert > 100) }
```

Klassenmethoden erhalten als *self*-Parameter kein Objekt ihres Klassentyps, sondern ein Objekt ihres Klassenreferenztyps (also eine Klassenreferenz). Als Konsequenz daraus können Klassenmethoden nicht auf die Variablen eines Objekts zugreifen. Dies sollte normalerweise auch nicht nötig sein, da es ja die Aufgabe der Klassenmethoden ist, Informationen über die gesamte Klasse zu liefern und nicht über einzelne Objekte. Sie können sich Klassenmethoden also auch als Methoden der Klassenreferenz vorstellen.

Oft sind Klassenmethoden nur sinnvoll, wenn sie auch virtuell sind. Die Methode im obigen Beispiel könnte beispielsweise die Unterschiede zwischen verschiedenen abgeleiteten Klassen deutlich machen, muss sich also in abgeleiteten Klassen überschreiben lassen. Wenn sie nicht virtuell wäre, würde sie kaum noch einen Sinn haben, da ihr Ergebnis vorhersehbar wäre.

Simulation von Klassenvariablen *R152*

Sinnvolle Beispiele für Klassenmethoden lernen Sie im nächsten Kapitel kennen. All diese Beispiele geben Informationen über die Klasse zurück; es wäre jedoch auch vorstellbar, dass Sie dadurch Einstellungen an der Klasse vornehmen. So könnte beispielsweise eine Rechteckklasse nicht nur über eine voreingestellte Breite und Höhe verfügen, sondern eine Klassenmethode anbieten, mit der Sie diese Voreinstellung ndern können:

```
type
  TRechteck = class
    class procedure SetStandardWH(W, H: Integer);
    ...
```

Leider hat diese Methode jedoch keine Möglichkeit, diese Daten direkt in der Klasse zu speichern, da es in Object Pascal keine Klassen*variablen* gibt. Allerdings können Sie sich in einem solchen Fall leicht mit globalen Variablen helfen, die Sie im Implementationsteil der Unit verbergen. Da die Klasse nur einmal vorkommt, sind globale Variablen hier zwar nicht elegant, aber aus technischer Sicht mit Klassenvariablen nahezu gleichwertig.

2.3.6 Typinformationen zur Laufzeit

Einen großen Fortschritt gegenüber der letzten *Pascal-with-Objects*-Version stellen die in diesem Abschnitt vorgestellten Neuerungen zur Laufzeit-Typinformation dar. Der englische Fachbegriff für diese Typinformationen ist RTTI (Run Time Type Information) und ist aus der C++-Welt importiert. Als Delphi die RTTI in Object Pascal einführte, war RTTI in der C++-Welt noch so neu, dass es noch nicht überall unterstützt wurde. Dabei enthalten die RTTI von Object Pascal noch deutlich mehr Informationen als die von C++.

TObject-Methoden

In Object Pascal erhalten Sie die Typinformationen, indem Sie Methoden der Klasse *TObject* aufrufen. Die folgende Tabelle fasst diese Methoden zusammen.

Funktionsart	Funktion	Ergebnistyp	Ergebnis
class function	ClassName	String	Name des Objekts oder der Klasse
class function	ClassNameIs(Str)	Boolean	Besagt, ob der Klassenname mit dem Parameterstring übereinstimmt
class function	ClassParent	TClass	Elternklasse des Objekts oder der Klasse
function	ClassType	TClass	Klasse des Objekts
class function	ClassInfo	Pointer	Zeiger auf ausführliche Informationen
class function	InheritsFrom(AClass)	Boolean	*True*, wenn die Klasse von *AClass* abgeleitet ist
class function	GetInterfaceTable	Zeiger auf TInterfaceTable	Informationen über die von einer Klasse implementierten Interfaces, siehe in der Online-Referenz auch die Funktion *GetInterfaceEntry*

Funktionsart	Funktion	Ergebnistyp	Ergebnis
function	GetInterface(IID, out)	Boolean	Ein Interface-Zeiger für ein bestimmtes, z. B. mit *IUnknown::QueryInterface* erfragtes Interface (siehe Kapitel 2.7.3)
class function	InstanceSize	Word	Größe einer Objektinstanz der Klasse

ClassName

Bei fünf der sechs Methoden handelt es sich um Klassenfunktionen, das heißt, dass sie (eigentlich) unabhängig von Objektinstanzen aufgerufen werden:

```
Name := TButton.ClassName;
```

In diesem Beispiel erhält *Name* den String mit dem Inhalt *'TButton'* zugewiesen. Interessant wird es erst, wenn das Ergebnis nicht schon von vornherein feststeht, wenn Sie also Klassennamen einer *polymorphen* Objektinstanz abfragen, z. B. in einer Ereignisbearbeitungsmethode, deren Parameter als *TObject* deklariert ist:

```
procedure TForm1.ButtonClick(Sender: TObject);
begin
  if Sender.ClassName = 'TSpeedButton' then
  ... else if Sender.ClassName = 'TButton' then ...
  { effizientere Abfragen mit dem is-Operator! }
```

Das Ergebnis von *ClassName* hängt nun von dem Objekt ab, das tatsächlich als Parameter *Sender* übergeben wurde. Die Methode *ButtonClick* kann auf diese Weise gleichzeitig Ergebnisse von zwei verschiedenen Schaltertypen verarbeiten und doch zwischen beiden unterscheiden (abgesehen davon, dass dieser spezielle Fall in der Praxis eher unwahrscheinlich ist).

Wie erklärt sich nun aber der Widerspruch zwischen der Aussage, dass Klassenmethoden unabhängig von Objektinstanzen sind, und der Tatsache, dass im letzten Beispiel ein Objekt und keine Klasse abgefragt wurde? Die Antwort ist einfach: Das Programm stellt zur Laufzeit fest, welche Klasse zur Instanz *Sender* gehört, und *für diese Klasse* wird dann die Methode *ClassName* aufgerufen. Diese ist dann wirklich nur noch von der Klasse abhängig. Tatsächlich sieht die Abfrage also wie folgt aus:

```
if Sender.ClassType.ClassName = 'TSpeedButton'
```

ClassType

Mit *ClassType* erhalten Sie die Referenz auf die Klasse selbst, also z. B.:

```
Ref := TObject.ClassType;
  { ergibt eine Referenz des Typs TClass }
```

```
Ref := TForm1.ClassType;  { oder }
Ref := Form1.ClassType;
  { ergibt eine Referenz des Typs "class of TForm1" }
```

ClassType ist keine Klassenmethode, d.h. dass sie nur mit Objekten verwendet werden kann. Eine Verwendung beispielsweise als *TClass.ClassType* wäre auch nicht sinnvoll, denn dann müssten Sie eine Referenz auf die Referenz erhalten.

ClassParent

Die Methode *ClassParent* liefert Ihnen eine Referenz auf die Elternklasse einer Klasse. Da sie nur den direkten Vorfahren einer Klasse angibt, können Sie damit nicht in einem Schritt ermitteln, ob eine Klasse von einer anderen abgeleitet ist (direkt oder indirekt). Das folgende Beispiel demonstriert die Verwendung von *ClassParent*:

```
function BaumEbene(Klasse: TClass): Integer;
begin
  Result := 0;
  { Wiederholen, bis die Klasse TObject erreicht ist: }
  while Klasse.ClassName <> 'TObject' do begin
    { Klassenreferenz der Basisklasse ermitteln: }
    Klasse := Klasse.ClassParent;
    inc(Result);
  end;
end;
```

Diese Funktion ermittelt, wie viele Ebenen sich die als Parameter übergebene Klasse unter der Klasse *TObject* befindet, sie gibt damit quasi die Entfernung der *Klasse* von *TObject* im Hierarchiebaum an. Diese Funktion ist im Beispielprogramm *DragDrop-Demo* (Kapitel 6) eingebaut.

InheritsFrom

InheritsFrom beseitigt die Nachteile der Methode *ClassParent*, indem sie angibt, ob eine Klasse auf irgendeine Weise von einer anderen Klasse abgeleitet ist, ob sie also von dieser erbt. Die folgenden Ausdrücke sind alle wahr:

```
TButton.InheritsFrom(TControl) = True
TForm.InheritsFrom(TGraphicControl) = False
EineKlasseX.InheritsFrom(EineKlasseX) = True
```

Die Operatoren is und as

Die beiden Operatoren *is* und *as* sind dazu da, Typenumwandlungen und Abfragen von Vererbungsverhältnissen auf besonders elegante Weise durchführen zu können. Auf der rechten Seite beider Operatoren steht immer eine Objektinstanz oder eine Klassenreferenz, während auf der linken Seite nur eine Klassenreferenz stehen darf.

testet, ob die Klasse des rechten Operanden ein Nachfahre der auf der linken Seite angegebenen Klasse ist. Die beiden folgenden Ausdrücke sind also gleichbedeutend:

```
(Object Is Class) = Object.InheritsFrom(Class)
```

As hat gegenüber einer herkömmlichen Typenumwandlung einen technischen Vorteil: Bevor dieser Operator einen Typ in einen anderen umwandelt, testet er, ob die Umwandlung überhaupt zulässig ist. Dabei testet *as*, ob der als Ergebnis gewünschte yp kompatibel zum Typ vor der Umwandlung ist, es läuft also dieselbe Prüfung ab, die vom *is*-Operator durchgeführt wird (es muss gelten: »Ergebnistyp *is* Ausgangstyp«). Sind die Typen nicht kompatibel, erzeugt *as* eine *EInvalidCast*-Exception. Damit ist die Umwandlung *A := B as ClassType* gleichbedeutend mit:

```
if B is ClassType then
  A := ClassType(B) { normale Typenumwandlung durchführen }
else
  raise EInvalidCast; { Ausnahmebedingung ausrufen }
```

RTTI

Wie Sie oben gesehen haben, enthalten die Klassenreferenzen selbst keine Informationen außer der Klasse, zu der sie gehören. Über die Methoden von *TObject* können Sie Informationen über den Klassennamen und Vererbungsverhältnisse erhalten, Vergleiche zwischen Klassen sind über Vergleich der Referenzen oder mit dem *is*-Operator möglich. Normalerweise werden Sie nicht mehr Informationen zur Laufzeit benötigen, denn welche Elemente Ihre Klassen enthalten, können Sie ja aus den Klassendeklarationen ablesen.

rotzdem sind Informationen über veröffentlichte Properties und Methoden auch zur Laufzeit abfragbar. Die CLX selbst verwendet diese Informationen beispielsweise zur Speicherung der Properties, wenn Komponenten in eine Datei geschrieben werden.

Schlüssel zu diesen Informationen ist die Methode *ClassInfo*. Diese liefert zwar einen untypisierten Zeiger zurück, die Datenstruktur, auf die dieser Zeiger weist, ist aber nicht undokumentiert, sondern nur versteckt, da sie wie gesagt kaum gebraucht wird. Aus demselben Grund soll sie hier nur kurz erwähnt werden:

Die Unit TypInfo

Um einen bequemen Zugriff auf die Informationen von *ClassInfo* zu erhalten, müssen Sie die Unit *TypInfo* einbinden und den *Pointer*, den *ClassInfo* zurückliefert, in *PTypeInfo* umwandeln, z.B.:

```
uses TypInfo;

var
  TypeInfo: PTypeInfo;
```

```
  TypeData: PTypeData;
begin
  TypeInfo := PTypeInfo(TForm.ClassInfo)
  TypeData := GetTypeData(TypeInfo);
```

Schon haben Sie mit *TypeData* einen Zeiger auf einen höchst informativen Datenblock, dessen Struktur vom Wert von *TypeInfo.Kind* abhängt. Ist dieser beispielsweise *tkClass*, dann befindet sich in *TypeData*^ neben dem Verweis auf die Basisklasse und anderen Daten eine Liste von Properties, die in dieser Klasse deklariert sind. Auch der Aufbau dieser Liste ist in der Unit *TypInfo* in Form von Typendeklarationen dokumentiert. Wenn Sie also tief greifende RTTI-Informationen benötigen, sollten Sie sich an das Interface dieser Unit wenden.

2.3.7 Kompatibilität mit Borland Pascal

Das neue Objektmodel von Object Pascal führt dazu, dass die Deklaration der Variablen *Form: TForm* nicht, wie bei Borland Pascal, Platz für die gesamten Objektdaten von *TForm* reserviert. Diese Deklaration entspricht vielmehr der BP-Deklaration *Form: ^TForm* – es werden also genau vier Bytes für einen Zeiger belegt.

Dadurch ändert sich auch der Zugriff auf Methoden solcher dynamischer Objekte: Statt der Dereferenzierung wie *Objekt^.ZaehleFenster* findet jetzt ein einfacher Zugriff mit *Objekt.ZaehleFenster* statt, als ob das Objekt statisch wäre. Der Konstruktoraufruf ist bei Object-Pascal-Objekten auch dann erforderlich, wenn Sie eine statische Methode aufrufen wollen.

Der Weg zu den von Borland Pascal gewohnten Objekten ist jedoch nicht weit: Benutzen Sie statt des Schlüsselworts *class* wie gewohnt *object*, so erzeugt der Compiler Objekte nach dem alten Muster.

2.4 Typen

Auf dem Gebiet der Variablenverwaltung führt Object Pascal gegenüber dem Ur-Pascal keine neuen Vereinfachungen oder gar visuelle Hilfsmittel ein, d.h. dass Sie lokale Variablen nach wie vor selbst deklarieren müssen, bevor Sie sie benutzen (siehe Kapitel 2.1.4). Die einfachen Variablentypen von Object Pascal entsprechen grundsätzlich den verschiedenen Wertetypen für Properties des Objektinspektors, weisen aber weitere Variationsmöglichkeiten auf.

2.4.1 Einfache Typen

Zu den einfachen Typen gehören die Fließkommatypen und die Typen, deren Werte als ganze Zahlen darstellbar sind, die so genannten *ordinalen* Typen. Die wichtigste Gruppe der ordinalen Typen sind die Integertypen.

Integertypen

Die verschiedenen Integertypen unterscheiden sich nach Speicherbedarf und durch die Tatsache, ob ein Bit für das Vorzeichen verwendet wird oder ob nur positive Zahlen gespeichert werden können:

yp	Wertebereich	Speicherbedarf
ShortInt	-128..127	1 Byte
Byte	0..255	1 Byte
SmallInt	-32768-32767	2 Bytes
Word	0..65535	2 Bytes
LongWord	0..42949672950	4 Bytes
LongInt	-2147483648..2147483647	4 Bytes
Int64	$-2^{63}..2^{63}-1$	8 Bytes
enerische Typen:		
Integer	-2147483648..2147483647	4 Bytes
Cardinal	0..42949672950	4 Bytes

Da die Typen *Cardinal* und *Integer* keine fest definierte Größe haben, sondern sich gegebenenfalls an die Prozessor-Architektur anpassen (beim Schritt von 16- zu 32-Bit-Code wurde die Größe von *Integer* beispielsweise verdoppelt), werden sie auch als *generische* ypen bezeichnet, im Gegensatz zu den Typen, die sich auf neuen Prozessor-Architekturen garantiert nicht ändern, den *fundamentalen* Datentypen. (Auf zukünftigen 64-Bit-Plattformen werden die *Integers* sich übrigens nicht weiter verbreitern; wer 64-bittige Integers haben möchte, wird dann wohl auf den heute schon existierenden Typ *Int64* zurückgreifen müssen.)

Int64 und LongWord in der Praxis

Die Typen *LongWord* und *Int64* (eingeführt in Delphi 4) lassen sich im Prinzip genau wie die alten Typen verwenden. Da sich *LongWord* zu *LongInt* genauso verhält wie *Word* zu *SmallInt*, werfen wir hier nur noch einen kurzen Blick auf die Besonderheiten des Typs *Int64*. Die gute Nachricht ist zunächst einmal, dass der Compiler seit der Einührung von *Int64* auch direkt angegebene Konstanten dieses Typs, also etwa ganze Zahlen mit 18 Nullen akzeptiert, beispielsweise so:

```
ShowMessage('Hello Int64: ' + IntToStr(1000000000000000000));
```

Ein weiteres Beispiel zeigt allerdings, dass Sie beim Typ *Int64* eventuell beachten müssen, dass der Standard-Integer von Kylix nach wie vor ein 32-Bit-Integer ist. Der folgende Aufruf, den Sie in eine beliebige *OnClick*-Methode einfügen können, erzeugt ämlich einen Überlauf beim Übersetzen des Programms:

```
ShowMessage('Int64-Test ergab: ' + IntToStr(1000000000 * 1000000000));
```

Dies liegt daran, dass Kylix jede der beiden Zahlen als 32-Bit-Integer interpretiert, da sie sich noch als solche darstellen lassen. Damit erhält aber auch das Ergebnis der Multiplikation vom Compiler nur noch einen 32-Bit-Integer zur Verfügung gestellt, wo ein 64-Bit-Integer notwendig wäre. Um das Problem zu beheben, müssen Sie die beiden Zahlen vor der Multiplikation manuell in *Int64*-Werte umwandeln:

```
ShowMessage('Int64-Test ergab: ' +
            IntToStr(Int64(1000000000) * Int64(1000000000)));
```

Hinweis: Für die Umrechnung von Strings in *Int64*-Werte gibt es die Funktion *StrToInt64*. Bei den Funktionen der Laufzeitbibliothek (und allgemein bei allen Funktionen und Methoden), die nur 32-Bit-Werte akzeptieren, werden 64-Bit-Integer-Werte auf 32 Bit gekürzt, wenn sie an diese Funktionen übergeben werden. Zu den Funktionen, die auch mit *Int64*-Werten umgehen können –, wie z. B. die oben verwendete *IntToStr* – siehe die Online-Hilfe unter dem Stichwort »Int64«.

Currency

Der Typ *Currency* (engl.: Währung) ist mit einem maximal darstellbaren Wert von 922.337.203.685.477,5807 (im negativen Zahlenbereich dehnt er sich sogar noch um 0,0001 weiter aus) auch für ausgefallene geldbezogene Berechnungen geeignet. Trotz der vorhandenen Nachkommastellen handelt es sich nicht um einen Fließkommatypen, denn der Kommabereich ist auf vier Stellen fixiert.

Vielmehr ist *Currency* bei Vernachlässigung des Kommas ein 64-Bit-Integer-Wert, der die Zehntausendstel einer (Geld-)Einheit angibt. Berechnungen, die nur *Currency*-Werte enthalten, lassen sich so vom Prozessor als schnelle Integer-Operation ausführen (wenn auch etwas langsamer als die »natürlichen« 32-Bit-Operationen).

Der Typ *Currency* reiht sich artig in die Reihe der anderen Typen ein, lässt sich also mit denselben Operatoren verarbeiten und wird bei Bedarf in Fließkommawerte umgewandelt (wobei ein Genauigkeitsverlust möglich ist), so dass Sie ihn wie eine Fließkommazahl behandeln und wie eine solche auch mit Integer-Typen kombinieren können (indem Sie den Integertyp in eine Fließkommazahl umwandeln oder den *Currency*-Typ in einen Integer, z. B. mit den Funktionen *Trunc* oder *Round*).

Fließkommatypen

Die Fließkommatypen von Kylix entsprechen den Fließkommaformaten der Intel-Prozessoren. Der Standard-Fließkommatyp ist *Double*, die Alternativen zu ihm verbrauchen mit 4 Bytes entweder weniger Speicher (*Single*) oder weisen mit ihren 10 Bytes

eine höhere Genauigkeit und einen größeren Zahlenbereich auf (*Extended*). Der Typ *Comp* macht eine Ausnahme und bietet bei relativ kleinem Zahlenbereich die Genauigkeit des Typs *Extended*, verbraucht aber nicht so viel Speicher wie dieser.

yp	Bereich	Genauigkeit	Speicherbedarf
Single	1.5*10^-45..3.4*10^38	7-8 Stellen	4 Bytes
Double	5*10^-324..1.7*10^308	15-16 Stellen	8 Bytes
Extended	3.4*10^-4932..1.1*10^4932	19-20 Stellen	10 Bytes
Comp	-9.2*10^18..9.2*10^18	19-20 Stellen	8 Bytes
Real	wie *double*		

Bis Delphi 3 hatte der Typ *real* noch sein aus alten Zeiten geerbtes eigenes 6-Byte-Format und daher natürlich einen kleineren Wertebereich. Da dieses Format aber von den Fließkommaeinheiten der aktuellen Prozessoren nicht unterstützt wird, war es trotz der handlicheren Ausmaße der Daten langsamer, weshalb Delphi seit der Version 4 und jetzt auch Kylix per Voreinstellung *Real*-Variablen wie ein *Double* behandelt. Über die Compilerdirektive {$RealCompatibility ON} bringen Sie auch die neueste Kylixersion noch dazu, die 6-Byte-Version von Real zu verwenden.

Zeichen: AnsiChar, WideChar und Char

Der Typ eines Zeichens heißt in Pascal *Char*. In einer *Char*-Variablen können alle Zeichen des ANSI-Zeichensatzes (bzw. ASCII) gespeichert werden, *Char*-Konstanten werden in Hochkommata eingeschlossen. Zeichen, die sich nicht auf der Tastatur befinden, önnen Sie mit ihrem Zahlenwert (z.B. #*10*) oder als Steuerzeichen (z.B. ^*A*) angeben (siehe Übersicht in Kapitel 2.1.2).

Kylix unterstützt auch Zeichen des Unicode-Zeichensatzes, die zwei Bytes benötigen. Der Typenbezeichner hierfür lautet *WideChar*. Da die ersten 256 Zeichen des Unicode-Zeichensatzes mit denen des ANSI-Zeichensatzes übereinstimmen, können Sie einer *WideChar*-Variablen ohne Probleme einen *Char*-Wert zuweisen.

echnisch gesehen sind *Char* und *WideChar* die einzigen Zeichentypen von Kylix 1. Allerdings sollten Sie sich auch mit dem Typ *AnsiChar* befassen. Dieser Typ bezeichnet ebenfalls nichts weiter als ein Zeichen nach dem ANSI-Zeichensatz, wie es auch in *Char* enthalten ist. Allerdings verhält es sich mit dem Typ *Char* so wie mit dem Typ *Integer* (es handelt sich also um einen generischen Typen): Er kann sich bei der Weiterentwicklung von Kylix ändern, was spätestens dann denkbar wäre, wenn Kylix auf einem Betriebssystem läuft, das nur noch mit dem Unicode-Zeichensatz arbeitet. Um auf jeden Fall mit einem nur ein Byte beanspruchenden ANSI-Zeichen zu arbeiten, sollten Sie daher den (fundamentalen) Typ *AnsiChar* verwenden. Die Größe des Typs *Char* ist für die Zukunft nicht festgelegt.

Aufzählungstypen

Pascal kennt zwei Arten von einfachen Typen, die nicht durch ein Schlüsselwort, sondern durch eine von Ihnen gegebene Definition festgelegt werden: *Teilbereichstypen* und *Aufzählungstypen*.

Aufzählungstypen kommen bei den Properties der Standardkomponenten sehr häufig vor. Die *WindowState*-Eigenschaft der Klasse *TForm* hat beispielsweise den folgenden Aufzählungstyp:

```
type
  TWindowState: (wsNormal, wsMinimized, wsMaximized);
```

Ein Aufzählungstyp besteht also aus einer Auswahl von benannten Zuständen, die Sie der Variable oder dem Property zuweisen können. Der Compiler verwaltet die in Frage kommenden Werte intern als Konstanten mit aufsteigenden Werten: Im obigen Beispiel erhält *wsNormal* den Wert 0, *wsMaximized* den Wert 2. Typisch für die CLX-Aufzählungstypen ist übrigens die Voranstellung eines Kürzels, wobei z.B. *ws* für *WindowState* steht. Auf diese Weise werden Konflikte mit anderen Aufzählungstypen, bei denen auch ein Zustand wie *Normal* vorkommen könnte, vermieden.

Sie können mit Aufzählungstypen auch rechnen bzw. alle Werte eines Aufzählungstyps in einer Schleife durchlaufen, wenn Sie sich das ordinale Wesen des Aufzählungstyps zunutze machen (siehe ordinale Typen, Seite 218).

Teilbereichstypen

Durch die Definition eines Teilbereichstyps teilen Sie dem Compiler mit, dass nicht der gesamte Wertebereich eines Bytes, Words oder sonstigen Integers für diese Variable erlaubt ist:

```
type
  ArrayIndex = 0..55;
  Temperatur = -200..10000;
```

Variablen des obigen Typs *ArrayIndex* würden vom Compiler wie ein Byte, die des Typs *Temperatur* wie eine Integer-Variable behandelt werden, da diese den von Ihnen gewünschten Wertebereich am platzsparendsten umfassen. Zusätzlich besteht der Compiler darauf, dass Sie diesen Variablen nur Werte zuweisen, die in den angegegeben Bereich fallen. Weisen Sie einer Variablen statt einer Konstanten einen Ausdruck mit Variablen zu, kann der Compiler natürlich nicht vorhersehen, ob das Ergebnis zur Laufzeit in den geforderten Bereich fällt. Daher bietet er Ihnen an, mit der Option *Bereichsüberprüfung* bzw. dem Compilerschalter *$R+* Code zu generieren, der die Bereichsgrenzen auch zur Programmlaufzeit überprüft und bei Überschreitung eine *ERangeError*-Exception erzeugt. Die Bereichsüberprüfung kann auch bei der Indizie-

ung von Arrays und Strings nützlich sein. Sie ist allerdings nur für Programme zu empfehlen, die noch in der Entwicklung sind, denn sie kostet viel (Lauf-)Zeit und vergrößert den Umfang des Programms.

Boolesche Typen

Als einen vordefinierten Aufzählungstyp, für den nur zwei Werte in Frage kommen, önnen Sie den Typ *Boolean* ansehen. Seine beiden Werte stehen für wahr (*True*) und falsch (*False*) und entstehen bei der Auswertung von booleschen Ausdrücken wie ($x <$ *MaxX*) and ($y <$ *MaxY*). Für eine *Boolean*-Variable reserviert der Compiler ein Byte Speicherplatz.

Intern wird *True* als 1, *False* als 0 gespeichert. Die interne Repräsentation von *ByteBool*, *WordBool* und *LongBool* wurde unter Delphi 3 allerdings an die entsprechenden Visual-Basic-Typen angepasst, *True* wird darin als -1 gespeichert. Dadurch sind *Byte-, Word-* und *LongBool* auf der einen und *Boolean* auf der anderen Seite nicht mehr völlig miteinander kompatibel.

Ordinale Typen

ür alle einfachen Typen außer den Fließkommatypen und den »Riesen-Integer« *Int64* gibt es eine eigene Bezeichnung: die der *ordinalen Typen*. Ihre wichtigste Gemeinsamkeit ist, dass sie alle auf leicht einsehbare Weise in ganze Zahlen umgewandelt werden önnen, falls sie das nicht schon sind. Mit der Funktion *ord* erhalten Sie zu einer ordinalen Variablen diesen zugehörigen Zahlenwert. Da jede ordinale Variable intern als eine solche Zahl gespeichert wird, muss die Funktion *ord* überhaupt nichts tun und kommt einer Typenumwandlung gleich. Hinter den verschiedenen Ordinaltypen verbergen sich die folgenden Zahlen:

- Die Ordinalzahl eines Zeichens ist sein ANSI-Code bzw. sein Unicode (für *WideChars*),
- die einer booleschen Variablen ist 1 für den Wert *True*, ansonsten 0,
- und die Konstanten eines Aufzählungstyps werden aufsteigend und von 0 beginnend durchnummeriert.

Natürlich können Sie auch den umgekehrten Weg einschlagen und einen Zahlenwert in einen anderen ordinalen Typ umwandeln, mit dem Ausdruck *Boolean(1)* erhalten Sie also wieder den Wert *True*.

High and Low R159

Zwei weitere wichtige Funktionen für ordinale Typen sind *High* und *Low*. Mit *High(Typenbezeichner)* oder *High(Variablenbezeichner)* erhalten Sie den höchsten Wert,

der im Wertebereich des Typs bzw. der Variablen liegt. *High(Boolean)* ergibt also den Wert *True*, *High* angewendet auf *TWindowState* fördert *wsMaximized* zu Tage. Den kleinsten Wert erhalten Sie entsprechend mit der Funktion *Low*.

Mit diesen beiden Funktionen können Sie leicht alle Werte eines Aufzählungstyps durchlaufen:

```
type
  TWerte = (Null, Eins, Zwei);
   { wenn dies z.B. geändert wird auf:
     "TWerte = (Null, Eins, Zwei, Drei, Vier);"
      ... muss der folgende Code nicht geändert werden: }
var
  Zaehler: TWerte;
begin
  for Zaehler := low(TWerte) to high(TWerte) do
    ... { Zaehler nimmt hier die Werte 'Null', 'Eins', und 'Zwei' an. }
```

2.4.2 Operatoren und Ausdrücke

Die arithmetischen Operatoren für die vier Grundrechenarten und die Operatoren für Vergleiche von Zahlen sehen in Pascal so aus, wie man es erwartet, lediglich die Operatoren für ungleich (<>), kleiner gleich (<=) und größer gleich (>=) müssen aufgrund fehlender anderer Symbole auf der Tastatur aus zwei Zeichen zusammengesetzt werden.

Die meisten anderen Operatoren sind Schlüsselwörter, die hinsichtlich ihrer Bedeutung leichter zu merken sind als Zeichenkombinationen.

Die Rangfolge der Operatoren

Nach der Rangfolge sind die Operatoren in vier Prioritätsstufen unterteilt; folgen mehrere Verknüpfungen gleicher Priorität, werden sie von links nach rechts ausgewertet. Für eine davon abweichende Verknüpfungsreihenfolge müssen Sie Klammerungen vornehmen:

Priorität	Operatoren	Beschreibung
1	()	Klammern (werden in Pascal nicht als Operatoren bezeichnet)
2	@, not	unäre Operatoren
3	*, /, div, mod, and, shl, shr	multiplizierende Operatoren
4	+, -. or, xor	addierende Operatoren
5	=, <>, <, >, <=, >=, in	relationale Operatoren

Spezielle Integer-Operatoren

Eine Besonderheit hat sich der Pascal-Erfinder Wirth bei den Divisionen von Integern einfallen lassen: Hierfür dürfen Sie nicht das übliche Symbol »/« verwenden, sondern müssen zu *div* greifen. Den Rest einer Division ermitteln Sie mit dem Operator *mod* (z.B. *7 mod 3 = 1*).

Der Adressoperator

Ebenfalls etwas aus der Reihe fällt der Adressoperator »@«. Er ist der einzige Operator, der nur mit Variablen anzuwenden ist und bei dem der Typ egal ist. Sein Ergebnis ist die Adresse der angegebenen Variablen bzw. ein Zeiger darauf (zu Zeigern siehe Kapitel 2.4.6).

Bitweise Operatoren

ür die Verknüpfung von Zahlen auf Bitebene verwenden Sie Operatoren, deren Name bereits die Art der Operation angibt:

Operator	**Verknüpfung**	**Beispiel**
not	bitweises Negieren	not $F0 = $0F
and	bitweises Und	$0F or $11 = $01
or	bitweises Oder	$40 or $04 = $44
xor	bitweises Exklusiv-Oder	$A8 xor $A0 = $08
shl	alle Bits nach links schieben	$01 shl 4 = $10
shr	alle Bits nach rechts schieben	$20 shr 2 = $08

Boolesche Operatoren

Die Operatoren *not*, *and*, *or* und *xor* können auch zur Verknüpfung von booleschen Werten verwendet werden und bedeuten auch dort so viel wie *nicht*, *und* bzw. *oder*; *xor* bedeutet so viel wie »genau eines von beiden«. So entsteht aus zwei booleschen Werten ein neuer boolescher Wert: Der Ausdruck

```
Button1.Checked and Button2.Checked
```

prüft z.B., ob zwei Markierungsschalter angewählt sind, und kann in einer der Anweisungen zur Programmablaufsteuerung verwendet oder einer booleschen Variablen zugewiesen werden.

Auswertung von booleschen Ausdrücken

Die Auswertung von booleschen Ausdrücken kann auf verschiedene Arten erfolgen. Wenn Funktionen am Ausdruck beteiligt sind – und in den folgenden Beispielen soll jeder Bezeichner für eine Funktion stehen –, kann das große Auswirkungen auf die Programmgeschwindigkeit haben ...

```
if True or RechneEineHalbeStunde then
  ...
```

... oder auf die Programmlogik:

```
if DisketteLeer and BenutzerErlaubt and DisketteFormatieren
   then Meldung('Diskette erfolgreich formatiert.');
```

Wenn der erste Operand in einer Oder-Verknüpfung wahr ist, wie im ersten Beispiel, steht bereits fest, dass der gesamte Ausdruck wahr ist, auch wenn der zweite Teil *False* ergibt. Sind im zweiten Teil intensive Berechnungen erforderlich, ist es sehr vorteilhaft, wenn das Programm die Auswertung des Ausdrucks so früh wie möglich abbricht und bei vorzeitigem *True*-Ergebnis den *then*-Teil sofort ausführt.

Auch im zweiten Beispiel hätte eine vollständige Auswertung des Ausdrucks gravierende Folgen: Selbst wenn der Benutzer in letzter Sekunde das Formatieren der Diskette verhindern wollte und die Funktion *BenutzerErlaubt* den Wert *False* zurückliefert, würde das Programm versuchen, den letzten Teil (*DisketteFormatieren*) auszuwerten, indem es die Funktion aufruft, die die Diskette formatiert.

Welche Auswertungsmethode der Compiler verwendet, hängt von der Einstellung der Option BOOLESCHE AUSDRÜCKE VOLLSTÄNDIG im Optionsdialog bzw. der Compileranweisung *$B* ab; per Voreinstellung bricht der Compiler die Auswertung so früh wie möglich ab, tut also in beiden obigen Beispielen das Sinnvolle.

Manchmal sind auch vollständige Auswertungen sinnvoll – der folgende Ausschnitt wird wegen der *$B*+-Option in jedem Fall mit vollständiger Auswertung übersetzt:

```
{$B+}
   GesamtTestOK := CDROMansprechbar and FestplatteOK;
```

In diesem Beispiel sollen CD-ROM und Festplatte getestet werden. Wenn sich beim Testen des CD-ROM-Laufwerks und seines Treibers ein Fehler ergibt, kann der Gesamttest zwar nicht mehr erfolgreich ausgehen, der Benutzer möchte aber trotzdem erfahren, ob denn wenigstens die Festplatte in Ordnung ist.

Relationale Operatoren

Die Vergleichsoperatoren haben ebenfalls boolesche Werte zum Ergebnis. Bei ihnen ist zu beachten, dass sie niedriger priorisiert sind als die oben beschriebenen booleschen Operatoren. Im folgenden Fall benötigen Sie also eine Klammerung, sonst verknüpft

der Compiler zuerst *MaxX* und *Y*, um dann einen Fehler wegen des »>«-Operators zu melden:

```
if (X > MaxX) or (Y > MaxY) then
   raise EIndexOutOfRange.Create('X oder Y'+
              ' außerhalb des zulässigen Bereichs');
```

Operatoren und andere Typen

Einige der bisher besprochenen Operatoren sind gewissermaßen polymorph wie eine virtuelle Funktion. Sie verursachen bei verschiedenen Typen verschiedene Operationen. Andere Typen außer den ordinalen und den Fließkommatypen, deren Variablen ebenfalls mit Operatoren verknüpft werden können, sind Zeiger, Mengen und Strings. Mehr dazu lesen Sie im entsprechenden Abschnitt.

2.4.3 Arrays

Arrays sind Tabellen mit einer oder mehreren Dimensionen, in denen Variablen, die alle denselben Typ haben, gespeichert werden. Object Pascal kennt auch indizierte Properties, die für den Nutzer wie ein Array aussehen. Einzelne Elemente von Arrays und indizierten Properties werden auf dieselbe Weise angesprochen: Hinter dem Namen des Arrays/Properties geben Sie die Indizes in eckigen Klammern an. Die Indizes müssen nicht unbedingt Zahlen sein, Arrays können alle ordinalen Typen als Indizes haben, bei Properties sind sogar alle Typen erlaubt. Ein Beispiel für Arrays mit ordinalen Dimensionen:

```
GruenAnteil := Farbe[Gruen];
SpreadSheet[5, 6] := '4*b';
ObjektImRaum := Raum[x,y,z];
```

Wie indizierte Properties deklariert werden, lesen Sie in Kapitel 2.2.5; an dieser Stelle sei angenommen, dass es sich im obigen Beispiel um normale Arrays handelt. Diese önnten dann mit Hilfe der Schlüsselwörter *array* und *of* wie folgt deklariert worden sein:

```
type
  (*  zwei ordinale Typen  *)
  Farbanteile = (rot, gruen, blau);
  TabellenIndizes = 1..100;
var
  Farbe: array[Farbanteile] of Integer;
  SpreadSheet: array[TabellenIndizes, TabellenIndizes] of Integer;
  Raum: array[0..10, 0..10, 0..10] of TRaumObjekt;
```

Es ist also möglich, das erste Element eines Arrays mit dem Index 0, 1 oder jeder anden Zahl im zulässigen Wertebereich zu belegen.

Dynamische Arrays

Object Pascal bietet eine besondere Unterstützung für *dynamische Arrays*, d.h. Arrays, deren Speicher erst zur Laufzeit reserviert wird. Im Gegensatz zu »handgemachten« dynamischen Arrays (mit Hilfe von Zeigern und Routinen zur dynamischen Speicherverwaltung) können Sie auf ein dynamisches Array im Sinne von Object Pascal wie auf ein statisches Array zugreifen, also ohne einen Zeiger dereferenzieren zu müssen. Darüber hinaus können Sie durch einfache Anweisungen eine Größenveränderung des Arrays veranlassen. Schließlich krönt Borland seine dynamischen Arrays noch mit einer Referenzzählung, doch dazu später mehr.

Ein dynamisches Array wird wie folgt deklariert:

```
var
  intArray1: array of Integer;
```

Dies mag so aussehen wie ein offenes Array, der Unterschied ist aber, dass es sich bei obiger Deklaration nicht um einen Funktionsparameter handelt. Wie bei dynamischen Objekten in Object Pascal üblich (siehe Konstruktor-Aufruf von Objekten), müssen Sie auch bei dynamischen Arrays explizit veranlassen, dass Speicher reserviert wird:

```
begin
  intArray1[1] := 100; // << Fehler, da noch nicht reserviert!
  SetLength(intArray1, 100);
  intArray1[100] := 100; // << Fehler, da Indizes von 0 bis 99
  intArray1[0] := 100; // Beschreiben des ersten Elements
```

Das Erfreuliche ist nun, dass Sie die Funktion *SetLength* auch weitere Male aufrufen können, wenn Sie die Größe des Arrays ändern wollen. Der bisherige Array-Inhalt wird dadurch nur dann »beschädigt«, wenn Sie das Array verkleinern.

Speicherverwaltung der dynamischen Arrays

Wie schon die langen Strings und Objekte mit Interfaces verfügen auch die dynamischen Arrays über eine automatische Referenzzählung, die es der Kylix-Laufzeitbibliothek erlaubt, automatisch für die Freigabe des Speichers zu sorgen, sobald das Array (bzw. der String bzw. das COM-Objekt) nicht mehr benötigt wird. Im einfachsten Fall ist das, wenn das Array in einer lokalen Variable erzeugt wird und die lokale Variable ungültig wird, weil die Ausführung der Funktion, in der sie deklariert wurde, beendet ist. Wenn Sie ein Array schon früher freigeben wollen, als dies automatisch geschehen würde, können Sie dies durch folgenden Aufruf tun:

```
Finalize(intArray1);
```

erwaltung mehrerer Referenzen

Der Referenzzähler eines dynamischen Arrays gibt an, von wie vielen Variablen ein Array referenziert wird. Im Schritt (*2*) des folgenden Beispiels wird beispielsweise erreicht, dass das im ersten Schritt erzeugte Array gleichzeitig von zwei verschiedenen ariablen benutzt wird:

```
type
  TIntArray = array of LongInt; // Diese Typdeklaration ist für die
                                // Zuweisung in Zeile (*4*) erforderlich.
var
  GlobaleInts : TIntArray;

procedure LokalerGueltigkeitsbereich;
var
  intarray1, intarray2: TIntArray;
begin
(*1*)  SetLength(intarray1, 1000000); (* Speicherverbrauch: 4 MByte *)
(*2*)  intarray2 := intarray1;
(*3*)  intarray2[50] := 100;
(*4*)  GlobaleInts := intarray2;
(*5*) end;
```

Wichtig ist, was in Zeile 3 passiert: Die Zuweisung an die fünfzigste Zahl des zweiten Arrays verändert gleichzeitig auch *intarray1[50]*, da beide Array-Variablen auf dasselbe Array verweisen, dessen Referenzzähler sich nun auf 2 erhöht hat. Probierten wir hnliches mit langen Strings, würden wir zwei verschiedene Strings mit einem Referenzzähler von 1 erhalten (zu *Copy On Write* siehe Kapitel 2.4.4, *Interner Aufbau der langen Strings*).

Zeile 4 bewirkt schließlich, dass der Referenzzähler auf 3 steigt, da nun auch die globale Variable *GlobaleInts* auf das Array verweist. Am Ende der Prozedur in Zeile 5 werden nur die beiden lokalen Variablen ungültig, der Referenzzähler des dynamischen Arrays wird um zwei kleiner, sinkt also nicht auf Null, so dass das Array auch noch nicht automatisch freigegeben wird.

Mehrdimensionale Arrays

Alles bisher Gesagte lässt sich direkt auf mehrdimensionale Arrays übertragen, wenn Sie nur wissen, wie ein mehrdimensionales Array deklariert wird ...

```
var
  IntMatrix: array of array of Int64;
```

... und dass die Prozedur *SetLength* mehrere Parameter haben kann:

```
SetLength(IntMatrix, 100, 100); // reserviert 100*100 Int64-Zahlen
```

Stellt man sich *array of Int64* als geklammert vor, so zeigt die Deklaration von *IntMatrix* bereits, dass sich *IntMatrix* auch als eindimensionales Array ansehen lässt, und es ist tatsächlich erlaubt, es auch so im Speicher anzulegen:

```
SetLength(IntMatrix, 100); // reserviert 100 array of Int64-Variablen
```

Für die 100 Elemente des Arrays gilt dasselbe wie für normale Variablen des Typs *array of Int64*: Jedes Element ist ein dynamisches Array, das einzeln mit *SetLength* oder durch Zuweisung initialisiert werden muss. Und da dynamische Arrays verschieden groß sein können, können Sie jedem Element ein Array einer anderen Größe zuweisen, so dass das gesamte Array *IntMatrix* nicht mehr rechteckig ist, sondern beispielsweise dreieckig oder völlig unregelmäßig gestaltet. Die Speicherverwaltung der Kylix-Laufzeitbibliothek jedenfalls lässt sich durch so etwas nicht verwirren.

2.4.4 Die verschiedenen Stringtypen

In Object Pascal gibt es drei unterschiedliche Stringtypen: *ShortString* (Strings mit internem Längenbyte, wie von Turbo Pascal bekannt), *String* (seit Delphi 2 wird für diesen Standard-Stringtyp ein neues Format verwendet) und *WideString* (ein String, der aus *WideChars* besteht, der also mit Zeichensätzen umgehen kann, die mehr als 255 verschiedene Zeichen enthalten). Für die Interoperabilität mit C/C++-Code wichtig ist der Typ *PChar*, der ein Zeiger auf ein C-kompatibles Zeichen-Array darstellt. Er stellt in Object Pascal keinen eigenen String-Typ dar (das heißt, dass Sie *PChar*-Variablen nicht wie *String*-Variablen manipulieren können). Wenn Sie einen *PChar*-Zeiger brauchen, verwenden Sie in Object Pascal normalerweise einen *String* und gewinnen daraus durch Typumwandlung einen *PChar*, wenn erforderlich.

Der Typ *String* ist vergleichbar mit den generischen Typen *Integer* und *Char*, da sich sein interner Aufbau zwischen Delphi 1 und Delphi 2 verändert hat, seine Verwendung aber nahezu gleich geblieben ist. Für die Zukunft ist eine erneute Änderung absehbar, falls der UniCode-Zeichensatz einmal den ANSI-Zeichensatz als Standard ablösen sollte (*String* wird dann zu einem *WideString*). Wenn Sie einen Stringtyp suchen, dessen interner Aufbau gleich bleibt (vergleichbar mit den fundamentalen Integer-Datentypen), so bietet Ihnen Object Pascal diesen im Typ *AnsiString* (String aus *AnsiChar*-Zeichen, zurzeit noch mit den normalen Strings identisch).

- Grundsätzlich sind alle Strings, die Sie mit dem Bezeichner *String* deklarieren, unabhängig von der verwendeten Compiler-Version sehr einfach zu handhaben (z.B. können Sie zwei Strings mit dem Operator '+' zu einem dritten String zusammenfügen). Die Einführung des neuen String-Formats in Delphi 2 wurde aber nicht nur erforderlich, um die steinzeitliche Längenbegrenzung auf 255 Bytes aufzuheben, sondern auch, um eine höhere Interoperabilität mit C/C++-Funktionen zu erreichen: Um einen Object-Pascal-String an eine C/C++-Funktion wie etwa einer

Funktion der C-Bibliothek von Linux oder dem Qt-Interface von Kylix weiterzugeben, müssen Sie ihn lediglich in einen *PChar* umwandeln, z.B. *ApiFunktionsAufruf(PChar(LangerPascalString))*.

Die folgenden Abschnitte behandeln die drei verschiedenen Stringtypen im Einzelnen.

Alte Pascal-Strings mit Längenbyte

Intern bestehen die Pascal-Strings des alten Formats (die Standard-Strings von allen urbo/Object-Pascal-Compilern bis Delphi 1) aus einem Zeichenarray, dessen Indizieung bei 0 beginnt. An Position 0 befindet sich ein Längenbyte, das angibt, wie viele Zeichen der String umfasst und wie viele Bytes des reservierten Speichers er somit verwendet. Die Menge des reservierten Speichers ist bei diesem Typ konstant. Wenn Sie also beispielsweise mit *string[50]* 51 Bytes (inklusive des Längenbytes) reservieren und der Variablen einen String zuweisen, der kürzer als 50 Zeichen ist, bleibt ein Teil des Speichers ungenutzt.

Die Deklaration von alten Pascal-Strings unterscheidet sich zwischen den Delphi-Versionen etwas. Die empfehlenswerte, weil eindeutige Deklarationsweise ist jene, die änge des Strings in eckigen Klammern hinter dem Wort *String* anzugeben:

```
var
  StringFuer100Zeichen: string[100];
```

Diese Deklaration reserviert in allen Delphi-Versionen einen statischen String des alten yps. Wenn Sie die Größenangabe in Klammern weglassen, erhalten Sie

- unter 16-Bit-Delphi einen alten String mit einer Kapazität von 255 Zeichen;
- unter 32-Bit-Delphi und in Kylix mit der standardmäßig eingestellten Option für lange Strings (*Huge Strings* in den Compileroptionen bzw. *{$H+}* im Quelltext) einen neuen dynamischen String mit bis zu 2 GB Kapazität;
- unter 32-Bit-Delphi und in Kylix mit abgeschalteten langen Strings (*{$H-}*) einen alten String mit 255 Zeichen Kapazität.

Normalerweise können Sie es jedoch immer bei der Einstellung *{$H+}* belassen, denn einen alten String mit 255 Zeichen erhalten Sie in jeder Delphi-Version mit *String[255]*. In der aktuellen Object-Pascal-Version gibt es außerdem die Typenbezeichner *ShortString* (für einen String im alten Format) und *AnsiString* (für einen langen String).

Die Verwendung dieses Stringtyps funktioniert so wie die Verwendung der langen Strings, sofern Sie nur die Routinen der Kylix-Laufzeitbibliothek auf diese Strings anwenden.

Die langen 32-Bit-Strings

Einen langen String erhalten Sie auf zwei verschiedene Arten:

- in der Voreinstellung *{$H+}* (*Huge Strings* in den Compileroptionen) einfach über den Typenbezeichner *String* oder über *AnsiString*;
- in der Einstellung *{$H-}* nur über den Typenbezeichner *AnsiString*.

Zeichenketten, die durch Hochkommas eingeschlossenen sind, sind mit dem neuen Stringtyp (und auch mit dem alten kurzen String) kompatibel. Sie können eine solche Zeichenkette also direkt einer Stringvariablen zuweisen und als Parameter für Funktionen verwenden, die einen String erwarten (sofern sie nicht zwingend eine String-*variable* verlangen).

```
var
  s: string;
begin
  s := 'Zuweisung eines (kurzen bzw. langen) Strings';
  MessageDlg('Ausgabe eines Strings im Meldungsfenster',
             mtInformation, [mbOK], 0);
```

Um die Länge eines Strings in Erfahrung zu bringen, sollten Sie immer die Funktion *Length* verwenden:

```
Len := Length(s);  { funktioniert immer }
Len := Byte(s[0]); { Direkte Abfrage des Längenbytes: funktioniert
                     nur bei den alten, kurzen Strings }
```

Um Strings miteinander zu verbinden oder ihre alphabetische Reihenfolge zu vergleichen, verwenden Sie den '+'-Operator bzw. die relationalen Operatoren '<' und '>'. Mit '=' ist ein schneller Stringvergleich möglich. All diese Operatoren unterscheiden zwischen Groß- und Kleinschreibung. (Für einen Stringvergleich ohne Beachtung der Groß- und Kleinschreibung ist die Funktion *CompareStr* vorgesehen.)

Zur Manipulation der Strings stehen Ihnen verschiedene Funktionen zur Verfügung, zu denen Sie in Kapitel 2.8.4 eine Übersicht finden. Um beispielsweise von einem String, der einen Punkt enthält, den Teil rechts des Punktes in einen neuen String zu kopieren, können Sie den Punkt mit der Funktion *Pos* suchen und dann den rechten Stringteil mit *Copy* kopieren:

```
PunktPosition := Pos(Eingabe, '.');
if PunktPosition > 0 then
  Ausgabe := Copy(Eingabe, { String, von dem kopiert werden soll }
                  PunktPosition+1, { Erstes Zeichen zum Kopieren }
                  Length(Eingabe)-PunktPosition) { Zahl der kopierten Z. }
else Ausgabe := 'Die Eingabe enthält keinen Punkt!';
```

Stringmanipulation auf Zeichenbasis

Auf die einzelnen Zeichen eines Strings können Sie wie bei einem ganz normalen Array zugreifen:

```
ZehntesZeichen := s[10];
s[9] := ZehntesZeichen;
```

Und Sie können sogar Zeichen beschreiben, die hinter dem bisherigen String-Ende liegen, wenn Sie den String vorher explizit verlängern. Zu Beginn hat beispielsweise jeder mit *String* deklarierte lange String die Länge 0 (ein mit *String[255]* außerhalb einer Klasse deklarierter String des alten Formats hat im uninitialisierten Zustand eine nicht vorhersagbare Länge von maximal 255 Zeichen, da das Längenbyte einen zufälligen Wert enthält). Wenn Sie nun im langen String das fünfte Zeichen setzen wollen, önnen Sie das wie folgt tun:

```
var
  s: AnsiString;
begin
  // s ist jetzt 0 Zeichen lang
  SetLength(s, 5);
  // s ist jetzt 5 Zeichen lang
  s[5] := NeuesZeichen;
```

Die Funktion *SetLength* funktioniert sowohl mit kurzen als auch mit langen Strings. In urbo Pascal und Delphi 1 hätten Sie die Länge eines Strings jedoch direkt mit *s[0] := Char(5)* setzen müssen. Wichtig ist bei den langen Strings, dass Sie die Länge *vor* dem Zugriff auf das Zeichen schon groß genug einstellen, da Sie sonst eine *EAccessViolation*-Exception erhalten.

Die Zeichen an den Positionen eins bis vier im obigen Beispiel haben übrigens nach Ausführung der beiden obigen Anweisungen noch einen zufälligen Inhalt.

Interner Aufbau der langen Strings

ariablen für lange Strings sind selbst immer nur vier Bytes groß und stellen einen Zeiger auf einen dynamisch verwalteten Speicherbereich dar, der den eigentlichen String-Inhalt enthält. Die Verwaltung dieses Speicherbereichs (dessen Größe bei einer ängenänderung des Strings angepasst wird) geschieht völlig automatisch, ebenso wie ie Dereferenzierung des Zeigers.

Der dynamisch verwaltete Speicherblock des Strings ist wie folgt aufgebaut: Er beginnt mit zwei 32-Bit-Werten, von denen einer die Länge des Strings angibt und der zweite einen *Referenzzähler* enthält. Daraufhin folgt der eigentliche String-Inhalt, abgeschlossen durch ein Nullzeichen.

Der Referenzzähler ermöglicht eine große Einsparung von Speicher, denn er erlaubt, dass sich mehrere Stringvariablen einen String teilen. Er zählt, wie viele Stringvariab-

len ein und denselben dynamischen String-Datenblock verwenden. Im folgenden Programmausschnitt wird nur einmal Speicher für einen String reserviert. *A* und *B* verwenden danach denselben Speicherbereich, der Referenzzähler dieses Bereichs enthält danach den Wert 2:

```
var
  a, b: AnsiString;
begin
  a := 'Zuweisung eines langen Strings';
  b := a;
```

Diese doppelte Verwendung desselben Speicherbereichs hindert Sie jedoch nicht daran, einen der beiden Strings zu verändern, denn die Laufzeitbibliothek verfährt in diesem Fall nach der so genannten *Copy-On-Write*-Methode. Wird die folgende Zeile ausgeführt ...

```
b := b+' mit anschließender Verlängerung';
```

... kopiert die Laufzeitbibliothek den noch unveränderten (von *a* und *b* genutzten) String in einen neuen Speicherbereich, setzt danach die Referenzzähler der beiden jeweils nur noch einmal verwendeten Speicherbereiche auf eins und hängt an den Speicherblock, der jetzt nur noch für die Variable *b* vorgesehen ist, die oben angegebene Zeichenkette an. (Dieses Verhalten steht im Unterschied zu den dynamischen Arrays, bei denen es keinen *Copy-On-Write*-Mechanismus gibt, siehe Kapitel 2.4.3)

Das Nullzeichen nach dem eigentlichen String-Inhalt ist zwar nicht mehr nötig, um das String-Ende festzustellen, da es hierfür ja schon die Größenangabe am Anfang des Speicherblocks gibt; aber durch das Nullzeichen lässt sich der String einfach in einen nullterminierten String umwandeln, beispielsweise, um ihn an eine C-Funktion zu übergeben:

```
syslog(LOG_INFO or LOG_USER, PChar(DebugString)); // (aus R162)
```

Dieser Aufruf erhöht allerdings *nicht* den Referenzzähler der beiden umgewandelten Strings, daher bergen solche Umwandlungen die Gefahr in sich, dass ein String zu früh automatisch freigegeben wird. Und auch wenn Sie einen so umgewandelten String verändern wollen, ist Vorsicht geboten: Indem Sie einen *String* als *PChar* ansprechen, umgehen Sie ebenfalls die automatische Verwaltung des Referenzzählers. Wenn Sie Strings dennoch auf diese Weise verändern wollen, finden Sie in der Online-Hilfe zu den langen Strings weitere Informationen dazu (so gibt es z. B. eine spezielle Prozedur *UniqueString*).

Umgekehrt können Sie einen nullterminierten String in einen *String* umwandeln, indem Sie beispielsweise *String(NullTermString)* schreiben. Oft wird eine solche Umformung jedoch schon automatisch durchgeführt, z. B. wenn Sie einem *String* einen nullterminierten String einfach zuweisen.

Hinweis: Für einen langen String mit einer Länge von null Zeichen wird überhaupt kein dynamischer Speicher reserviert. Lediglich der Zeiger, aus dem die eigentliche Stringvariable besteht, nimmt weiterhin vier Bytes in Anspruch; er enthält dann den Wert *nil*. Da Sie die Länge eines langen Strings von der Funktion *Length* erfahren können, brauchen Sie sich auch um dieses interne Detail normalerweise nicht zu kümmern.

Nullterminierte C-Strings

Nullterminierte Strings waren zu 16-Bit-Zeiten die einzige Möglichkeit, Strings, die nger als 255 Zeichen sind, zu verwalten. Inzwischen sind sie als eigenständige Strings kaum mehr nötig, denn jeder String des aktuellen Formats *enthält* quasi einen nullterminierten String, ergänzt durch eine zusätzliche Längenangabe, die die Verwaltung des Strings vereinfacht, und weiterhin ergänzt durch ein vollkommen automatisches Speichermanagement.

Ein nullterminierter String besitzt kein Längenbyte, sondern definiert sein Ende durch ein Terminator-Nullzeichen (#0). Durch das Fehlen des Längenbytes gestalten sich einige Operationen einfacher. So können Sie beispielsweise einen Teilstring aus einem nullterminierten String erhalten, indem Sie einfach einen Zeiger auf das Zeichen setzen, mit dem der Teilstring beginnen soll. Der Zeiger weist zwar nur auf einen Teilstring, dieser ist aber jederzeit als eigenständiger String verwendbar. Dies ist mit den Object-Pascal-Strings nicht möglich: Weder aus kurzen, noch aus langen Pascal-Strings lassen sich so einfach Teilstrings herausziehen, die wieder ein Pascal-String sind.

Natürlich gibt es auch Nachteile; vor allem ist die Frage, wie lang ein nullterminierter String ist, nur sehr zeitaufwändig zu beantworten. So muss die Funktion *StrLen* (das Gegenstück zur Pascal-String-Funktion *Length*) die Länge eines Strings ermitteln, indem sie alle Zeichen des Strings bis zum abschließenden Nullzeichen zählt. Außerdem wird die Verwendung von nullterminierten Strings weitaus weniger durch den Compiler unterstützt als die Verwendung von Pascal-Strings. So können Sie die nullterminierten Strings nicht mit Operatoren manipulieren, sondern müssen die Funktionen der Unit *SysUtils* (siehe auch Kapitel 2.8.4) verwenden. Sogar für die Zuweisung eines in Hochkommas angegebenen Strings an ein nullterminiertes Zeichenarray benötigen Sie einen Funktionsaufruf (*StrCopy*).

Doch zuerst zur Deklaration eines nullterminierten Strings. Sie deklarieren ihn wie ein normales Zeichenarray, dessen erster Index 0 ist:

```
var
  NullStr: array[0..100] of Char;
```

Nun können Sie die zahlreichen Funktionen der Unit *SysUtils* auf diesen String anwenden. Um beispielsweise einen Pascal-String in diesen zu kopieren, verwenden Sie die Funktion *StrPCopy*:

```
var
  Str: string;
begin
  StrPCopy(NullStr, Str);
```

PChar

Schneller geht es, wenn Sie Konstanten vom Typ *PChar* verwenden, der einen Zeiger auf einen nullterminierten String darstellt. Ihm können Sie Zeichenketten direkt zuweisen:

```
const
  StringZeiger: PChar = 'Dieser String wird nicht kopiert.';
```

Dies veranlasst den Compiler dazu, den angegebenen String als nullterminierten String zu speichern und den Zeiger *StringZeiger* auf dessen Anfangsadresse zu setzen. Eine *PChar*-Variable können Sie überall dort einsetzen, wo Sie auch einen nullterminierten String verwenden können. Zu bedenken ist aber, dass Sie die *PChar*-Variable vorher auf einen reservierten Speicherbereich setzen müssen.

Das folgende Beispiel demonstriert den oben schon erwähnten Vorteil der nullterminierten Strings und durchläuft einen String vom ersten Zeichen bis zum abschließenden Nullzeichen. Wichtig ist dabei, dass *Zeiger* zu jeder Zeit auf einen gültigen nullterminierten String weist, den Sie wie jeden anderen nullterminierten String verwenden können:

```
procedure DurchlaufeStr(Str: PChar);
var
  Zeiger: PChar;
begin
  Zeiger := Str;
  while Zeiger^ <> #0 do begin
    inc(Zeiger); { um eins erhöhen }
    ...
```

Sie können also mit einem *PChar*-Zeiger rechnen, indem Sie Zahlen hinzuaddieren (oder Funktionen wie *inc* und *dec* verwenden), wodurch sich seine Position um die entsprechende Zeichenzahl nach rechts oder links bewegt. Wenn zwei *PChar*-Variablen in den gleichen String zeigen, können Sie durch die Vergleichsoperatoren feststellen, welche der beiden Positionen vor der anderen liegt.

WideStrings

Ein Unicode-String ist ein String, dessen einzelne Zeichen jeweils zwei Bytes groß sind. Für diese Strings gibt es den Typ *WideChar*, der ein zwei Byte großes Zeichen speichert, und den Typ *PWideChar*, der dem Typ *PChar* für 1-Byte große Zeichen entspricht, mit dem Sie also ebenfalls »rechnen« können und den Sie ebenfalls als *array[0..x] of WideChar* anlegen können.

Passend zum *WideChar* gibt es den Typ *WideString*, der wie ein *AnsiString* automatisch verwaltet wird, allerdings ohne die Vorteile einer Referenzzählung. Das folgende Beispiel zeigt, dass Sie mit *WideStrings* wie mit einem *AnsiString* umgehen können, dass Sie jedoch für die Konvertierung von *WideString* in *AnsiString* eine Hilfsfunktion aufrufen müssen (Gleiches gilt für den umgekehrten Weg):

```
var
  W1, W2: WideString;
  NormalString: String;
begin
  W1:='Test der langen und weiten Strings. ';
  W2:=W1+W1;
  NormalString:=WideCharToString(@w1[1]);
```

Unicode-Strings werden bisher noch eher selten eingesetzt. Ein Beispiel für die zunehmende Verwendung gibt die Qt-Bibliothek: Sie kapselt Unicode-Strings in der Klasse *QString*, die aber auch noch normale C-Strings verwalten kann. Wenn Sie nun beispielsweise die Textausgabemethoden der *QPainter*-Klasse verwenden wollen, benutzen Sie jedoch nicht die *QString*-Klasse, sondern übergeben einen *PWideString* an die von Borland bereitgestellt Qt-Schnittstellen-Funktion *QPainter_drawText* (siehe z. B. Kapitel 5.4.3).

2.4.5 Strukturierte Typen

Die Records der herkömmlichen Pascal-Programmierung sind die Grundlage für die Objekttypen und können auch jetzt noch bei einfachen Strukturen, die keine Methoden benötigen, als reine Datentypen verwendet werden. Eine Record-Deklaration besteht aus dem Schlüsselwort *record*, einer Liste von Datenelementen und einem abschließenden *end*, dem noch ein Semikolon folgt:

```
type
  TDatei = record
    Name: string[12];
    Pfad: string[120];
    Groesse: LongInt;
  end;
```

Ein Record ist ein eigener Gültigkeitsbereich. Seine Elemente dürfen grundsätzlich alle als Bezeichner erlaubten Namen haben, auch wenn diese in einem äußeren Gültigkeitsbereich schon definiert sind, denn um die Elemente anzusprechen, müssen Sie sowieso zuerst den Namen des Records angeben, wodurch Verwechslungen ausgeschlossen sind:

```
var
  Name: string;
  Datei: TDatei;
begin
  Datei.Name := Name;
```

Varianten

Eine kompliziertere Form der Records sind die Records mit Abschnitten, die auf verschiedene Weise gedeutet werden können, den *Varianten*. In einer Syntax, die der *case*-Anweisung ähnelt, geben Sie hier verschiedene Möglichkeiten an, wie der mehrdeutige Bereich interpretiert werden kann. Der mehrdeutige Bereich muss sich immer am Schluss des Records befinden (weshalb er nicht durch ein eigenes *end* beendet wird) und kann z. B. so aussehen:

```
type
  VariantRecord = record
     VariableImmerDa: Integer;
     case GrosseTeile: Boolean of
       True: (a, b, c: LongInt);
       False: (d, e, f, g: Word;
               i: LongInt)
  end;
```

Semikolons sind hier erforderlich, um die einzelnen Fälle und die Deklarationen innerhalb der Fälle zu trennen.

Mengen

Wir kommen nun zu einem Typ, der in der CLX zahlreiche Verwendung findet: dem Mengentyp. Grundsätzlich gilt die folgende Syntax für die Deklaration eines Mengentyps und die Angabe von Mengen, die darin gespeichert sind:

```
var
  CharSet: set of Char;
begin
  CharSet := ['a'..'z'];
```

Der Basistyp einer Menge (der Typ, der hinter dem *of* angegeben wird) muss ein ordinaler Typ sein, der nicht mehr als 256 verschiedene Werte annehmen kann, denn für jeden möglichen Wert braucht eine Mengenvariable ein Bit, um zu speichern, ob der Wert in der Menge enthalten ist oder nicht. Das heißt, dass eine Zeichenmenge wie die obige mit 32 Byte so viel Platz wie 8 *LongInts* in Anspruch nimmt.

Eine Menge ist natürlich nur dann interessant, wenn schnell Elemente hinzugefügt oder herausgenommen werden können und wenn man überprüfen kann, ob ein Element enthalten ist oder nicht. Zwei Standardfunktionen vereinfachen das Hinzufügen und Wegnehmen einzelner Elemente:

```
{ Hinzufügen eines Elements: }
  Include(CharSet, 'd');
{ Löschen eines Elements: }
  Exclude(CharSet, 'x');
```

Diese Funktionen sind allerdings nicht auf Properties anwendbar, da ihre Parameter ariablenparameter sind. Ein Mengen-Property wird beispielsweise wie folgt durch ein neues Element erweitert:

```
BorderIcons := BorderIcons+[biMaximize];
```

Neben dem '+'-Operator können Sie noch viele weitere Operatoren auf Mengen anwenden:

Operator	Funktion	Ergebnis	Beispiel
	Vereinigung	Menge	[1..3,5,9]+[4..6]=[1..6, 9]
	Differenz	Menge	['c'..'h']-['a'..'z']=[]
	Schnittmenge	Menge	[1,3,5,6]*[1,2,3]=[1,3]
	gleich	Boolean	([blau]=[blau])=True
<>	ungleich	Boolean	
<=	Teilmenge	Boolean	([5]<=[1,5,10])=True
>=	Obermenge	Boolean	
	echte Teilmenge	Boolean	([rot]<[rot])=False
	echte Obermenge	Boolean	
	Element enthalten	Boolean	(Gruen in [Rot, Gruen])=True

2.4.6 Zeigertypen

Die Bedeutung der Zeigertypen (Zeiger = *Pointer*) ist in Object Pascal zwar dadurch etwas zurückgegangen, dass der Compiler Objekte und Strings von sich aus dynamisch verwaltet und diesen Vorgang nahezu völlig transparent für den Entwickler gestaltet. In anderen Bereichen sind Zeiger jedoch weiterhin sehr hilfreich: Die Verwaltung von nullterminierten Strings und dynamischen Speicherblöcken, deren Größe sich womöglich zur Laufzeit noch ändern kann, ist mit Zeigern besonders vorteilhaft, ebenso wie der Aufbau von dynamischen Datenstrukturen wie Listen und Bäumen.

Während bei allen anderen Variablen Speicher für irgendwelche Daten reserviert wird, enthalten Zeigervariablen lediglich die Adresse eines solchen Speicherbereichs. Für diese werden bisher seit den Anfangstagen von Turbo Pascal 4 Bytes Speicher benötigt, bei zukünftigen 64-Bit-Prozessorarchitekturen kann dieser Bedarf aber auch auf 64 Bit anwachsen.

Bei der Deklaration einer Zeigervariablen wird zwischen typisierten und untypisierten Zeigern unterschieden. Der Inhalt beider ist völlig identisch, jedoch kann der Compiler den Umgang mit typisierten Zeigern besser überwachen. Einen typisierten Zeiger, der auf die Variable eines Basistyps zeigt, erhalten Sie mit der Schreibweise »^*BasisTyp*«.

Meistens erhalten Zeigertypen eigene Bezeichner und werden erst dann einer Variablen zugewiesen (im folgenden Beispiel werden die Typen *PByteFeld* und *PInteger* auf diese Weise deklariert):

```
type
  TByteFeld = array[0..100] of Byte;
  TEinfachRecord = record
    a, b: Byte;
  end;
  PByteFeld = ^TByteFeld;
  PInteger = ^Integer;

var
  ZeigerAufInteger: PInteger;
  ByteFeldPtr: PByteFeld;
  RecordPtr1, RecordPtr2: ^TEinfachRecord;
```

Zuweisungen an Zeigervariablen

Da Zeigervariablen Adressen enthalten, können Sie ihnen auch nur Adressen zuweisen. Diese können Sie grob gesagt auf drei Arten erhalten:

- Sie nehmen die Adresse eines anderen Zeigers,
- reservieren neuen Speicher und verwenden die Adresse, die von einer Speicherreservierungs-Funktion wie *New* oder *GetMem* zurückgeliefert wird,
- oder Sie holen sich mit dem Adressoperator »@« oder der Funktion *Addr* die Adresse einer Variablen:

```
var
  ReserviertesByteFeld: TByteFeld;
  ReservierteZahl: Integer;
{ außerdem gelten die Bezeichner des letzten Beispiels }
begin
  RecordPtr1 := RecordPtr2;
  ByteFeldPtr := addr(ReserviertesByteFeld); { oder: }
  { Reservierung von 101 Bytes für ein TByteFeld }
  GetMem(ByteFeldPtr, sizeof(TByteFeld));

  ZeigerAufInteger := new(PInteger); { oder: }
  ZeigerAufInteger := addr(ReservierteZahl);
```

Das obige Beispiel setzt voraus, dass *RecordPtr2* vorher auf eine der letzten beiden Arten auf einen gültigen Speicherbereich gesetzt wurde, denn beim Programmstart zeigen Zeigervariablen (die nicht Teil eines automatisch initialisierten Objekts sind) in zufällige Bereiche, was zu schweren Fehlern führen kann, sobald Sie in diese Bereiche hineinschreiben.

nil-Pointer

Um Zeiger zu kennzeichnen, die auf kein gültiges Objekt zeigen, können Sie ihnen den Wert *nil* zuweisen. Ein Schreiben an die Adresse *nil* führt zwar immer noch zu einem Fehler, jedoch können Sie die Adresse *nil* vor dem Schreiben eindeutig als fehlerhaft erkennen. Andererseits können Sie davon ausgehen, dass Zeiger, die nicht *nil* sind, gültig sind, wenn Sie am Anfang des Programms alle ungültigen Zeiger mit *nil* initialisiert haben und jeden Zeiger wieder auf *nil* zurücksetzen, wenn Sie den ihm zugeordneten Speicher freigeben. Zur Überprüfung auf *nil* können Sie einen direkten Vergleich anstellen oder die Standardprozedur *Assigned* verwenden:

```
RecordPtr2 := nil;
if Assigned(RecordPtr2) then
  { ... Zugriff }
else { kein Zugriff }
```

Dereferenzierung

Sehen wir uns nun an, wie diese kritischen Zugriffe überhaupt erfolgen können. Um von der Adresse, auf die ein Zeiger zeigt, lesen zu können oder auf sie zu schreiben, müssen Sie den Zeiger *dereferenzieren*. Dafür ist wieder das Zeichen '^' zuständig, allerdings wird es diesmal hinter die Variable geschrieben. *RecordPtr2*^ gibt also nicht mehr den Zeiger an, sondern den Speicherbereich, auf den er zeigt. Mit diesem können Sie so arbeiten wie mit einer normalen Variable des Zeiger-Basistyps. Wenn es sich um einen Record handelt, können Sie dessen Elemente einzeln wie im folgenden Beispiel ansprechen:

```
RecordPtr2^.a := 1;
ByteFeldPtr^[0] := 0;
ZeigerAufInteger^ := 1000;
GetMem(ByteFeldPtr, sizeof(ByteFeldPtr^));
```

Hinweis: Die Dereferenzierung ist auch beim Debuggen wichtig. Wenn Sie etwa im Datenbereich des CPU-Fensters den Speicherbereich sehen wollen, auf den ein Zeiger zeigt, müssen Sie unter ZU ADRESSE GEHEN den Ausdruck *ZeigerVar*^ eingeben. Wenn Sie nur *ZeigerVar* eingeben, zeigt Kylix den Speicherbereich der eigentlichen Zeigervariablen, also den Bereich, in dem die Adresse liegt.

Untypisierte Zeiger

Wenn Sie Zeigeroperationen frei von den Zwängen des Compilers durchführen wollen, können Sie zu den untypisierten Zeigern greifen (meist ergibt sich deren Benutzung jedoch nicht aus Freiheitsstreben, sondern aus pragmatischen Gründen). Untypisierte Zeiger werden mit dem Wort *Pointer* dek

lariert:

```
var
  NatuerlicherZeiger: Pointer;
  KultivierterZeiger: PByteArray;
begin
  NatuerlicherZeiger := KultivierterZeiger; { ohne Typenumwandlung }
  { mit Typenumwandlung: }
  KultivierterZeiger := PByteArray(NatuerlicherZeiger);
```

Das Beispiel zeigt, dass untypisierte Zeiger überall dort verwendet werden können, wo auch andere Zeiger stehen. Um einen untypisierten Zeiger jedoch einem typisierten zuweisen zu können, müssen Sie dem Compiler über eine Typenumwandlung wie *PByteArray(NatuerlicherZeiger)* garantieren, dass der untypisierte Zeiger tatsächlich auf einen Speicherbereich zeigt, der mit dem Typ des typisierten Zeigers verträglich ist (siehe Abschnitt 2.4.7).

2.4.7 Typenkompatibilität und Typenumwandlungen

Mit den in Abschnitt 2.4.6 beschriebenen Record-Varianten können Sie bereits auf verschiedene Weise auf ein und denselben Speicherbereich zugreifen. Zu diesem und noch zu mehr in der Lage ist die Typenumwandlung (auf die Speicherbereiche kommen wir Ende des Abschnitts noch einmal zurück).

Kompatibilitätsregeln

Werfen wir zunächst einen Blick auf die Typenkompatibilitätsregeln von Object Pascal. Dass zwei Variablen, die denselben Typenbezeichner als Typ besitzen, sich gut vertragen, versteht sich von selbst. In vielen Fällen wäre es jedoch sehr unbequem, wenn nur identische Typen miteinander kooperieren könnten. Daher gibt es in Pascal zahlreiche Regeln zur *Typenkompatibilität*, die z.B. erlauben, dass in einem Ausdruck verschieden große Integervariablen (z.B. *Byte* und *Integer*) und Variablen von verschiedenen Fließkommatypen, z.B. *Real* und *Double* miteinander »verrechnet« werden können. Andere Regeln betreffen z.B. die Kompatibilität verschiedener Prozedur- und Recordtypen, werden aber so selten benötigt, dass an dieser Stelle auf eine Object-Pascal-Referenz verwiesen sei.

Ein etwas strengeres Kriterium ist die *Zuweisungskompatibilität*, die z.B. nicht gegeben ist, wenn Sie eine *LongInt*-Variable in ein *Byte* zwängen wollen, denn das würde den Zahlenwert verstümmeln, sofern er größer als 255 oder kleiner als 0 ist. So achtet der Compiler bei Zuweisungen immer darauf, dass die rechte Seite der Zuweisung in die linke hineinpasst (dabei wandelt er bei Bedarf Integerwerte automatisch in Fließkommawerte um).

Umwandlung von Werten *R158*

Die Zuweisung einer größeren Variablen an eine kleine ist ein gutes Beispiel für Typenumwandlung. Wenn Sie beispielsweise wissen, dass eine Ihrer *Word*-Variablen nicht größer als 1000 werden kann, und Sie sie durch 10 teilen, können Sie danach beruhigt eine Typenumwandlung vornehmen. Dieses und weitere Beispiele im folgenden Listing:

```
ByteVar := Byte(WordVar div 10);
Zeichen := Char(10); { Umwandlung von Zahl in Char (Char(10) = #10) }
Falsch := Boolean(0); { Umwandlung von Zahl in Boolean }
```

Umwandlung von Variablen

Die Typenumwandlungen aus diesen Beispielen sind nur auf der rechten Seite der Zuweisung möglich, da sie einen Wert zum Ergebnis haben. Die andere Art der Typenumwandlung betrifft die Interpretation von Variablen, ähnlich den Varianten in den Records (besprochen in Kapitel 2.4.5).

```
type
  TCasedLongInt = record
     case Boolean of
       True: (LoWord, HiWord: Word;);
       False: (AsLongInt: LongInt;)
  end;
  TLongIntSplit = record
    Low, High: Word;
  end;

var
  L1: LongInt;
  L2: TCasedLongInt;
```

Um eine der *LongInt*-Variablen nun als *LongInt* und zugleich als zwei *Words* ansprechen zu können, haben Sie bei obigen Deklarationen zwei Möglichkeiten:

```
begin
{ Möglichkeit 1: }
  L2.AsLongInt := ... { als LongInt ansprechen }
  L2.LoWord := ... { eine Hälfte als Word ansprechen }
{ Möglichkeit 2: }
  L1 := ...  { als LongInt }
  TLongIntSplit(L1).Low := ...  { als Word }
```

Eine Typenumwandlung wie die in Möglichkeit 2 wandelt das auf der linken Seite der Zuweisung stehende Objekt in einen anderen Typ um. Dies ist nur erlaubt, wenn der neue Typ exakt die gleiche Größe hat wie der alte, so dass seine Struktur die Struktur der umzuinterpretierenden Variablen genau überdeckt.

Hinweis: Diese Einschränkung haben Sie bei Varianten in Records nicht, denn diese dürfen auch unterschiedlich lang sein.

2.4.8 Initialisierte Konstanten strukturierter Typen

Während Sie einfache Konstanten typisieren oder als echte Konstanten deklarieren können, müssen Sie bei strukturierten und Array-Konstanten immer den Typ angeben. Das folgende Beispiel zeigt die Syntax dieser Deklarationen:

```
const
  ObjectConst: TDatei = (
    Name: 'dcc';
    Pfad: '/root/kylix/bin'
  );
  ArrayConst: array[0..10] of Byte =
  ( 0, 0, 0, 100, 1, 100, 0, 0, 0, 1, 0);
```

Bei initialisierten Records müssen Sie nicht alle Elemente initialisieren, die initialisierten müssen jedoch in der richtigen Reihenfolge angegeben werden. Wichtig ist auch, dass hinter der letzten Angabe kein Semikolon stehen darf. Bei initialisierten Arrays müssen Sie alle Elemente angeben und als Trennzeichen ein Komma verwenden.

Besonders bei den Arrays ist diese Art der Initialisierung bequemer, als wenn Sie das Array im Programm mit einzelnen Zuweisungen füllen würden. Darüber hinaus ist sie viel eleganter, denn zur Programmlaufzeit wird keine Zeit mit dem Kopieren einzelner Elemente verschwendet, da diese sich schon wohlgeordnet in Record- bzw. Array-Struktur in der ausführbaren Datei befinden. Im Falle einer Initialisierung per einzelner Zuweisung würden die einzelnen Werte zur Programmlaufzeit unter Umständen sogar doppelt vorliegen: Beim Programmstart wären sie nur einmal als einzelne Konstanten zu finden, bei der Zuweisung würden sie dann kopiert werden, was eine Verdopplung bedeutete.

2.5 Anweisungen und Funktionen

Das Thema dieses Kapitels sind reine Prozeduren und Funktionen, die auch ohne OOP verwendet werden können. Da die Funktions- und Prozedursyntax auch zur Deklaration von Methoden in Objektklassen Verwendung findet, sind diese Grundlagen jedoch keineswegs veraltet. Object Pascal weist außerdem gegenüber älteren Turbo-Pascal-Versionen Neuerungen wie offene Strings, offene Arrays und Prozedurtypen auf.

Zuerst fasst dieses Kapitel jedoch einen wichtigen Bestandteil für den Code der Prozeduren und Funktionen – die zur Sprache gehörenden Anweisungen – zusammen, die Object Pascal von Borland Pascal komplett erbt. Neue Kontrollstrukturen für Ausnahmebehandlung (Exceptions) werden in Kapitel 2.6 erläutert.

2.5.1 Pascal-Anweisungen

Die meisten Anweisungen der Sprache Pascal dienen zur Programmablaufsteuerung, sind also Schleifen, Bedingungsabfragen und Sprünge – alles Aufgaben, die mit Prozeduren und Funktionen alleine nicht zu bewältigen sind. Pascal verfügt mit der *with*-Anweisung über eine weitere, hauptsächlich der Bequemlichkeit dienende Anweisung, die nichts mit dem Programmablauf zu tun hat.

Die for-Schleife

Die einfachste Kontrollstruktur ist die *for*-Schleife, die eine oder mehrere Anweisungen mehrmals ausführt. Die Zahl der Wiederholungen steht vor der ersten Wiederholung fest:

```
x := a + b;
for i := a to b do
   WiederholteAnweisung;
```

Zu Beginn der Schleife setzt das Programm die Variable *i* auf den Wert *a*, erhöht sie am Ende jeder Schleife und führt diese das letzte Mal aus, wenn *i* den Wert hat, den *b* vor Beginn der Schleife besaß. Änderungen von *a* und *b* in der Schleife haben also keine Auswirkungen.

öcke

Alle Kontrollstrukturen können nicht nur einzelne Anweisungen, sondern auch Anweisungsblöcke ausführen. Außer bei der *repeat...until*-Schleife müssen diese in *begin* und *end* eingefasst werden, was im Beispiel der *for*-Schleife wie folgt aussieht:

```
for i := 1 to 100 do begin
   Anweisung1;
   Anweisung2;
   ...
end;
```

Bedingungen

Alle anderen Anweisungen zur Programmablaufsteuerung arbeiten mit expliziten Bedingungen. Bei der Auswertung einer Bedingung entsteht in Pascal immer entweder einer der Werte *True* und *False*, der besagt, ob die Bedingung erfüllt (*True*) ist oder nicht (*False*). Die Formulierung der Bedingungen ist einfach, da meist bekannte Vergleichsoperatoren der Mathematik Verwendung finden.

Die if-Anweisung

Die *if*-Anweisung besteht aus einem zu testenden Ausdruck und maximal zwei alternativen Anweisung(sblöck)en. Sie wertet den Ausdruck aus und führt, falls sie auf das Ergebnis *True* gekommen ist, die erste Alternative aus, andernfalls die zweite, hinter dem Schlüsselwort *else* angegebene, die jedoch auch weggelassen werden kann:

```
if Ausdruck
   then EsIstWahr
else EsIstFalsch;
```

Um nicht von den Unterschieden der *if*-Anweisungen in den verschiedenen Sprachen (z.B. im Vergleich zu C++) verwirrt zu werden, sollte man sich merken, dass in Pascal die gesamte *if*-Anweisung mitsamt dem *else*-Teil als eine einzelne Anweisung zählt und dass ein Semikolon nie innerhalb einer Anweisung stehen darf, also auch nicht vor dem *else* wie in C++. Auch die Interpretation der Schachtelung basiert auf diesem Grundsatz:

```
if Alternative1 then
   if Alternative2 then
     Anweisung1
   else Anweisung2;
```

Hier stellt sich bei jeder Sprache die Frage, zu welcher Abfrage *Anweisung2* zählt. In Pascal erwartet der Compiler nach dem ersten *then* eine komplette Anweisung, und die ist nach *Anweisung1* noch nicht zu Ende, da kein Semikolon dahinter steht (und wenn eins da stünde, wäre es ein Fehler, da es die äußere *if*-Anweisung unterbrechen würde). Das heißt also, dass *Anweisung2* in Object Pascal zur zweiten *if*-Anweisung gezählt wird. Das obige Beispiel könnte durch ein weiteres *else* ergänzt werden, das dann zur ersten *if*-Anweisung gehören würde (dann müsste das Semikolon hinter *Anweisung2* allerdings wieder gelöscht werden).

All diese Fragen stellen sich natürlich nicht, wenn Sie die zusammengehörenden Anweisungen durch *begin* und *end* kennzeichnen, wodurch Sie auch den obigen *else*-Teil einer anderen *if*-Ebene zuordnen können:

```
if Alternative1 then begin
  if Alternative2 then
     Anweisung1;
end
else Anweisung2;
```

Die repeat-Schleife

Kommen wir zur ersten Endlosschleifen-Kandidatin, der *repeat...until*-Schleife. Sie führt die zwischen den beiden Schlüsselwörtern stehenden Anweisungen so lange durch, bis der hinter *until* angegebene Ausdruck *True* ergibt. Das Folgende ist eine Endlosschleife, die jedoch in ihrem endlosen Tun keinen Sinn erkennen lässt:

```
repeat
   a := a+1;
   if a > 1 then a := 1;
until a > 9;
```

Die while-Schleife

Es ist nicht auf eine empfehlenswerte Weise möglich, den Rumpf der *repeat...until*-Schleife zu überspringen, da die Bedingung erst am Ende überprüft wird. Diese Tatsache macht diese Schleife zu einem Spezialfall der wirklich allgemeinsten Pascal-Schleife, der *while*-Schleife. Diese prüft schon zu Beginn, ob der Rumpf überhaupt ausgeführt werden soll:

```
{ Leeren eines Stacks: }
while not Stack.Empty do
   Stack.Pop;
```

Auch hier gelten für mehrere Anweisungen und für die Bedingung hinter dem *while* die bekannten Regeln.

Die case-Anweisung

Die *case*-Anweisung hilft, komplizierte *if*-Abfragen zu vermeiden, indem sie praktisch eine Tabelle mit möglichen Werten und den dazugehörigen Funktionen enthält:

```
case Key of
   VK_F1: Anweisung1;
   VK_F2: begin
     Anweisungen;
   end;
   VK_, VK_...:
   else ...
end;
```

Die Variable hinter dem *case* muss einen ordinalen Typ haben, der maximal die Größe eines *Word* haben darf. Sie können mehrere Bedingungen wie im Beispiel durch Kommas trennen und gemeinsam behandeln. Alle übrig bleibenden Bedingungen können optional im *else*-Teil mit einer speziellen Aktion behandelt werden.

Ist eine Bedingung gefunden, so wird der *case*-Block nach den dahinter stehenden Anweisungen vollständig beendet, es ist dazu also nicht wie etwa in C extra ein *break* erforderlich.

Sprünge

In Object Pascal gibt es vier Arten von Sprüngen:

- Mit *Exit*, das ausnahmsweise kein Schlüsselwort ist, verlassen Sie eine Prozedur oder Funktion sofort, was gleichbedeutend ist mit einem Sprung zum *end* der

Funktion. Mit dieser Anweisung lassen sich unter Umständen eine oder mehrere Ebenen von *if*-Anweisungen umgehen. Eine neue Sprungart macht die *exit*-Anweisung jedoch unnötig:

- Exceptions sind die modernste Art von Sprüngen, unter Hinzunahme der anderen Kontrollstrukturen lassen sich sowohl *exit*- als auch *goto*-Sprünge leicht vermeiden. Ihnen gehört ein eigenes Kapitel dieses Buchs (Kapitel 2.6).
- Sprünge in Schleifen: Mit *continue* springen Sie innerhalb einer Schleife zurück an den Schleifenanfang, wo die Schleife bei der nächsten Wiederholung fortgesetzt wird. Mit *break* verlassen Sie eine Schleife sofort.
- Für besonders waghalsige Sprünge stehen weiterhin die *goto*-Anweisungen zur Verfügung, die zu einer Zahlenmarke springen, die vorher in einem *label*-Block wenigstens scheinbar zivilisiert vorangekündigt werden muss. Aus technischen Gründen (Stack) ist es außerdem nicht möglich, zwischen verschiedenen Funktionen herumzuspringen. Wir verzichten hier auf ein Beispiel.

Aufgrund besonderer Grobheit verwandt mit *break*, *continue* und *exit* ist außerdem die Anweisung *Halt*. Sie beendet das Programm, ist aber intern erheblich komplexer aufgebaut als ein normaler Sprung, da beispielsweise die *finalization*-Abschnitte aller Units ausgeführt werden müssen.

Die with-Anweisung

Die einzige Kontrollstruktur von Object Pascal, zu der es kein C++-Äquivalent gibt, ist die *with*-Anweisung. In ihr geben Sie einen Gültigkeitsbereich an, in dem der Compiler zuerst nach den Bezeichnern suchen soll – die er dann natürlich auch verwendet. Mit der *with*-Anweisung können Sie statt

```
DefaultFontStruct.Name := 'Phantom';
DefaultFontStruct.Height := 20;
DefaultFontStruct.UseRaytracing := True;
```

einfach schreiben:

```
with DefaultFontStruct do begin
   Name := 'Phantom';
   Height := 20;
   UseRaytracing := True;
end;
```

Durch die *with*-Anweisung werden also eventuell Bezeichner im aktuellen Gültigkeitsbereich verdeckt. Sie können hinter dem *with* sogar mehrere durch Kommas getrennte Gültigkeitsbereiche angeben und *with*-Blöcke schachteln. Diese Möglichkeiten sind in der Praxis jedoch selten sinnvoll anwendbar.

2.5.2 Prozeduren und Funktionen

Die Routinen und Methoden werden in Object Pascal nach Prozeduren und Funktionen unterschieden – wohl eher aus traditionellen als aus logischen Gründen, denn auch Funktionen, die mehrere Ergebnisse nicht in einer Struktur zurückgeben wollen, werden häufig in eine Prozedur umgewandelt.

Theoretisch unterscheidet sich die Funktion durch die Rückgabe eines Ergebnisses von der Prozedur. Dieses geben Sie hinter einem Doppelpunkt nach der Parameterliste an:

```
function BerechneTangens(f: real): real;
begin
  Result := sin(f) / cos(f);
end;

procedure FuehreGeheimauftragDurch(ae34, bx45, cd77: LongInt;
                                   returntime: Word);
begin
  ...
end;
```

Die Regeln für die Parameterliste sind bei Prozeduren und Funktionen identisch: Die Syntax ist dieselbe wie innerhalb eines *var*-Blocks, allerdings hat das Schlüsselwort *var* hier eine andere Bedeutung, wie der nächste Abschnitt erläutern wird.

Zum Aufrufen einer Prozedur schreiben Sie deren Namen, gefolgt von der Liste der Parameter, wie z. B.:

```
FuehreGeheimauftragDurch(12, 9978, 9990000, 1010);
```

Bei Aufrufen von Prozeduren ohne Parameter dürfen dem Prozedurnamen keine leeren Klammern (wie in C) folgen. Für Funktionen gilt dasselbe mit dem Zusatz, dass das Funktionsergebnis für irgendeinen Zweck verwendet werden sollte, wie oben in der Beispielfunktion mit dem Ergebnis von *cos* geschehen. Funktionsergebnisse können auch ignoriert werden, so dass Sie die Funktionen wie Prozeduren aufrufen können (dazu muss allerdings die Compileroption ERWEITERTE SYNTAX / *$X* eingeschaltet sein, was standardmäßig der Fall ist).

Funktionsergebnisse

Eine der vielen kleinen, aber sehr nützlichen Erweiterungen von Object Pascal ist die variable *Result*, die automatisch in jeder Funktion definiert ist. Sie ist eine neue Bezeichnung für etwas, das es in Pascal schon immer gab: eine Speicherstelle für das Funktionsergebnis. Bisher hatte diese Speicherstelle immer den Namen der Funktion, nun können Sie sie alternativ unter dem Namen *Result* ansprechen. Dies macht es erheblich einfacher, die Funktion umzubenennen (oder zwecks geringfügiger Veränderung zu kopieren, was trotz OOP noch ab und zu vorkommen soll ...).

Vorwärtsdeklarationen

Wenn sich zwei Funktionen gegenseitig benutzen, müssen Sie eine davon in einer Vorwärtsdeklaration erwähnen:

```
procedure ProcBbenutztA; forward; { Vorwärtsdeklaration }
procedure ProcAbenutztB;
begin
  ProcBbenutztA;
end;
procedure ProcBbenutztA;
  { Definition der vorwärts deklarierten Prozedur }
begin
  ProcAbenutztB;
...
```

Die erste Funktion könnte die zweite nicht aufrufen, wenn der Compiler diese nicht schon durch die Vorwärtsdeklaration kennen würde.

Solche Vorwärtsdeklarationen sind bei Methoden natürlich nicht erforderlich, da die Klassendeklaration bereits quasi eine einzige große Vorwärtsdeklaration für alle Methoden ist.

Aufrufkonventionen

Hinter einer Funktionsdeklaration können Sie eine Direktive angeben, die festlegt, auf welche Art die Funktion aufgerufen wird:

- *Register* ist die voreingestellte Methode, bei der der Compiler versucht, möglichst viele Parameter zu übergeben, ohne dafür den Stack in Anspruch zu nehmen. Einige Parameter werden so direkt in den Registern des Prozessors an die Funktion übergeben, was natürlich erheblich schneller geht als die Inanspruchnahme des im Hauptspeicher befindlichen Stacks.
- *Pascal* ist eine etwas antiquiert anmutende Methode, die seit Turbo Pascal und bis zur ersten Delphi-Version als Standardmethode verwendet wurde. Die Parameter werden hier in der Reihenfolge auf den Stack gelegt, in der sie deklariert werden.
- *Cdecl* ist nach der Sprache C benannt, in der die Funktionen ihre Parameter traditionell in umgekehrter Reihenfolge erwarten und zusätzlich voraussetzen, dass der Aufrufer der Funktion einige Stack-Aufräumarbeiten durchführt.
- *StdCall* ist eine Art Mischung zwischen *pascal* (für die Reihenfolge) und *cdecl* (für die Stack-Verwaltung).
- *SafeCall* entspricht *StdCall* mit einer zusätzlichen Fehler- und Ausnahmebehandlung.

2.5.3 Parametertypen

Zuerst zwei Begriffserklärungen: Die Parameter, die Sie in der Deklaration einer Funktion angeben, heißen *formale Parameter*, während die Parameter, mit denen die Funktion zur Laufzeit arbeitet, als *aktuelle Parameter* und in der deutschen Übersetzung der Compiler-Fehlermeldungen neuerdings auch als »*wirkliche Parameter*« bezeichnet werden.

Wertparameter

ür Parameter gibt es in Pascal, wie in den meisten anderen Sprachen auch, zwei Möglichkeiten der Übergabe: Entweder handelt es sich bei den aktuellen Parametern um die Originalvariablen, die Sie an die Funktion übergeben haben, oder um Kopien davon. Im letzteren Fall kann die Funktion den Wert der Parameter ändern, ohne dass sich das auf die aufrufende Funktion auswirkt, solche Parameter heißen *Wertparameter*:

```
procedure PaintCells(von, bis: Integer);
begin
  for von := von to bis do begin
     ...
  end;
end;

var
  Start, Ende: Integer;
begin
  Start := 0;
  Ende := 10;
  PaintCells(Start, Ende);
```

In diesem Beispiel wird die Variable *Start* nicht verändert, da *PaintCells* eine Kopie des Wertes in einer neuen Variablen *von* erhält. Diese Variable liegt auf dem Stack und wird ungültig, sobald *PaintCells* beendet wird.

ariablenparameter

Damit die Prozedur aus dem letzten Beispiel den Originalparameter *Start* erhält, müsste sie wie folgt deklariert werden (was natürlich in diesem Fall nicht sinnvoll wäre):

```
procedure PaintCells(var von: Integer; bis: Integer);
```

Das Schlüsselwort *var* teilt dem Compiler mit, dass er der Funktion einen Zeiger auf den Originalparameter übergeben soll. Der Name *von* würde sich dann auf denselben Speicherplatz beziehen wie die Variable *Start*. Es versteht sich von selbst, dass Sie bei ariablenparametern keine Werte übergeben können, die quasi obdachlos sind, weil sie keine Adresse im Speicher haben (z. B. direkt angegebene Zahlen).

Nützlich sind Variablenparameter nicht nur zum Verändern der Werte, sondern auch zum Sparen von Stack-Speicher und zu einem schnelleren Aufruf der Prozedur/Funktion:

```
type
  TBigArray: array[0..100] of LongInt;

procedure TakeArray(var A: TBigArray);
```

Ohne das Schlüsselwort *var* würde das Programm bei jedem Aufruf von *TakeArray* die 400 Bytes des Array-Parameters auf den Stack kopieren. Mit *var* wird lediglich ein Zeiger auf das Array übergeben, der 4 Bytes benötigt. Allerdings kann die Prozedur das Original-Array nun verändern. Um diesen Nachteil von *var* auszugleichen, ist das *const*-Schlüsselwort auch für Parameter anwendbar.

Const-Parameter

Object Pascal bietet die Möglichkeit, einen Parameter als *const* zu deklarieren. Der Compiler erlaubt dann nicht, dass dieser Parameter innerhalb der Prozedur verändert wird. Wenn es sich bei dem Parameter sowieso nur um eine Kopie handelte, wäre das sicher nicht sinnvoll, daher wird ein *const*-Parameter so übergeben wie ein *var*-Parameter. Dies ist bei Variablen, die viel Speicher in Anspruch nehmen, die effektivere Lösung, da anstatt einer kompletten Kopie nur ein Zeiger übergeben werden muss.

Untypisierte Parameter

Des Weiteren gibt es die Möglichkeit, Parameter ohne einen festgelegten Typ zu deklarieren. In diesem Fall muss es sich entweder um einen *var* oder um einen *const*-Parameter handeln:

```
procedure InterpretiereDatenBlock(const Datenblock);
```

Der Parameter *Datenblock* hat keinen Typ, was den Vorteil hat, dass er in jeden beliebigen Typ umgewandelt werden kann, und den Nachteil, dass der Compiler nicht überprüfen kann, ob diese Umwandlung zulässig ist. Die Prozedur *InterpretiereDatenBlock* könnte den Datenblock beispielsweise in einen privaten Recordtypen umwandeln, der mit der Struktur der Daten übereinstimmt (siehe Typenumwandlung in Kapitel 2.4.7).

Offene Arrays

Im Zusammenhang mit Arrays gibt es als letzte Parametervariante die des *offenen Arrays* und speziell für Strings die *offenen Strings* – sofern Sie die voreingestellte Compileroption *$P+* (*Offene Arraygrenzen*) nicht ausgeschaltet haben. Diese Parameter heißen offen, da im formalen Parameter (also in der Deklaration) nicht festgelegt wird, wie viele Elemente das Array oder wie viele Zeichen der String hat. Das heißt, dass Sie

Strings und Arrays beliebiger Größe als (aktuelle) Parameter übergeben können. Sie müssen beim offenen Array lediglich den Typ der Elemente angeben und haben darüber hinaus die Wahl zwischen normaler, *const*- und *var*-Parameterübergabe:

```
function SummiereArrayKopie(a: array of Integer): Integer;
function SummiereArrayDirekt(const a: array of Integer): Integer;
procedure SortiereArray(var a: array of Integer);
```

Die Unterschiede zwischen *var*-, *const*- und normalem Aufruf stimmen mit dem von normalen Variablen überein, die Prozedur *SummiereArrayKopie* ist also zur Laufzeit deutlich weniger effektiv als *SummiereArrayDirekt*, da sie eventuell große Kopieraktionen notwendig macht, die aufgrund der Offenheit des Arrays nicht vorhersehbar sind.

Einen offenen formalen Array-Parameter können Sie auf zwei verschiedene Weisen mit einem aktuellen Array-Parameter füllen:

```
var
  ParamArray: array[0..10000] of Integer;
begin
  SortiereArray(ParamArray); { durch Angabe einer Array-Variablen }
  SummiereArrayKopie([322, 223, 443, 988]); { mit einem temporären Array }
```

SortiereArray nimmt beliebige Integer-Arrays an, daher können Sie die Variable *ParamArray* ohne Bedenken übergeben. In der zweiten Anweisung wurde keine Array-Variable verwendet, sondern das im nächsten Abschnitt beschriebene Konstrukt.

Der Open-Array-Konstruktor

Die zweite Anweisung im letzten Beispiel verwendet den neuen *Open-Array-Konstruktor* von Object Pascal, der unscheinbarerweise einfach aus eckigen Klammern besteht. In diesen eckigen Klammern geben Sie eine Liste von Werten an, die zu einem temporären Array zusammengesetzt werden, das dann anschließend an die Funktion weitergegeben wird. Bei der Funktion *SummiereArray* im Beispiel kommt das konstruierte Array so an, als hätten Sie es wie folgt deklariert:

```
const
  TempArray: array[0..3] of Integer =
   (322, 223, 443, 988);
```

Es ist logisch, dass ein so konstruiertes Array als Variablenparameter nicht sinnvoll ist, weil die veränderten Werte sofort nach dem Aufruf wieder verloren gehen – schließlich handelt es sich um ein temporäres Array.

Arbeiten mit offenen Arrays

Die sinnvolle Verwendung von offenen Arrays ist nur möglich, wenn Sie innerhalb der Prozedur feststellen können, wie groß das Array tatsächlich ist. Innerhalb der Prozedur können Sie daher die folgenden Funktionen auf das Array anwenden:

```
StartIndex := Low(a); { Index des ersten Elements (ist immer 0) }
EndIndex := High(a); { Index des letzten Array-Elements }
GesamtArrayGroesse := SizeOf(a);
```

Da das erste Element den Index 0 hat, ist die Gesamtzahl der Elemente *EndIndex+1*. Wenn Sie die Bereichsüberprüfung *$R* eingeschaltet haben, generiert der Compiler zusätzlichen Code, der jeden Zugriff auf das offene Array auf die Einhaltung der tatsächlichen Array-Grenzen überprüft (siehe Kapitel 2.6.5).

Hinweis: Den offenen Arrays sehr ähnlich sind die offenen Strings (Typ *OpenString* bzw. Compileroption *$P*), die jedoch aufgrund der langen Strings seit Delphi 2 nicht mehr benötigt werden und hier nur der Vollständigkeit halber erwähnt werden sollen.

Variable Parameterlisten

Der Open-Array-Konstruktor nimmt bereits eine beliebige Anzahl von Parametern auf, aus denen er ein temporäres Array macht, das Sie einer Prozedur mit offenem Array-Parameter übergeben können. Auf diese Weise könnte man schon fast variable Parameterlisten in Object Pascal nachbilden. An der Simulation einer wirklich variablen Parameterliste hindert nur noch die Beschränkung, dass Array-Parameter nur Werte eines einzigen Typs übernehmen können.

Als ersten Schritt zur Lösung bietet die Unit *System* den Typ *TVarRec* an, der 16 verschiedene Parametertypen – inklusive Objektinstanzen und Klassenreferenzen – aufweist:

```
type
  TVarRec = record
    case Byte of
      vtInteger: (VInteger: LongInt; VType: Byte);
      vtBoolean: (VBoolean: Boolean);
      vtChar:    (VChar: Char);
      vtExtended:(VExtended: PExtended);
      vtString:  (VString: PShortString);
      vtPointer: (VPointer: Pointer);
      vtPChar:   (VPChar: PChar);
      vtObject:  (VObject: TObject);
      vtClass:   (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
      vtInt64:      (VInt64: PInt64);
    end;
```

ür alle Typen, die größer als 4 Bytes sind, speichert *TVarRec* einen Zeiger, der auf das Element weist, so dass in jedem Fall 5 Bytes zur Speicherung eines *TVarRec* genügen (inklusive dem Byte *VType*, das in jedem Fall den Typ des Elements angibt, auch, wenn es nur in der Alternative *vtInteger* deklariert ist). Wenn Sie ein Parameter als ein offenes Array von *TVarRec*-Elementen deklarieren, können Sie bereits Parameter beliebiger Anzahl und beliebigen Typs entgegennehmen. Es kommen jedoch noch zwei kleine Erweiterungen von Object Pascal hinzu, die den Vorgang weiter vereinfachen:

- Der offene Array-Konstruktor ist nun auch in der Lage, *TVarRec*-Arrays automatisch zu konstruieren. Umschließen Sie einfach die zu übergebenden Parameter in eckigen Klammern, und der Array-Konstruktor macht daraus ein *TVarRec*-Array, wobei er auch das Feld *TVarRec.Type* selbstständig ausfüllt.
- Schließlich gibt es für derartige offene Arrays eine eigene Schreibweise: *array of const*.

Eigene Funktionen mit variablen Parameterlisten *R160*

Das folgende Beispiel fasst die Verwendung des offenen Arrays mit variablen Elementtypen zusammen und zeigt, wie die Funktion *WriteLn* mit einem offenen Array-Parameter nachgebildet werden kann: (*Writeln* schreibt eine Zeile in eine Textdatei, wobei die Zeile aus einer beliebigen Folge von Strings und verschiedenen anderen Variablen bestehen kann.)

```
function NewWriteln(var f: Text; const Args: array of const): string;
var
  i: Integer;
begin
  for i := 0 to High(Args) do
    case Args[i].Type of
      tkInteger: ...
end;

var
  Str: string;
  Int: Integer;
  Ext: Extended;
begin
  { Vergleich des Aufrufs einer System-Funktion mit variabler
    Parameterliste ... }
  System.Writeln(Zieldatei, 'Ausgabe dreier Variablen: ',
                 Str, ' ', Int, ' ', Ext);
  { ... mit dem Aufruf einer selbst geschriebenen Funktion mit
    einem offenen TVarRec-Array (array of const): }
  NewWriteln(Zieldatei, ['Ausgabe dreier Variablen: ',
                         Str, ' ', Int, ' ', Ext]);
```

Formal hat *NewWriteln* zwar immer nur zwei Parameter, durch die einfache Konstruktion des offenen Arrays bleibt aber kaum noch ein Unterschied zur völlig freien Parameterangabe beim Original-*Writeln*.

Laufzeitbibliothek und CLX enthalten bereits mehrere Funktionen, die offene Parameter dieser Art erhalten, z.B. die Funktion *Format*, die Strings ähnlich wie die C-Funktion *sprintf* formatiert, die Menü-Konstruktionsfunktionen (Kapitel 4.6) und einige Methoden von *T(Client)DataSet* (Kapitel 7.4). Zwar ist die Konstruktion dieser temporären Parameter nicht gerade zeitsparend, aber solange Sie eine solche Aktion nicht gleich tausend Mal hintereinander ausführen, dürfte kaum Gefahr für die Programmgeschwindigkeit bestehen.

2.5.4 Überladen von Funktionen, Standardparameter

Haben Sie sich schon gefragt, warum es in der Laufzeitbibliothek von Kylix Routinen gibt, die auf eine viel flexiblere Art aufgerufen werden können als Ihre selbst geschriebenen Routinen? Zwei simple Beispiele dafür sind die Prozeduren *inc* und *write*. Bei *inc* können Sie einen optionalen Parameter angeben:

```
inc(Zaehler);    // erhöht den Zaehler um den Standardwert 1
inc(Zaehler, 5); // erhöht den Zaehler um 5
```

Mit *write* können Sie verschiedene Datentypen beispielsweise in eine Textdatei schreiben:

```
var
  Zahl: Integer;
begin
  write(Textdatei, Zahl); // schreibt eine Variable des Typs Integer
  write(Textdatei, '. Element'); // schreibt einen String
```

Seit Delphi 4 erlaubt es auch Object Pascal jedem Entwickler, Prozeduren, Funktionen und Methoden zu schreiben, die in dieser variablen Art aufgerufen werden können. Dies funktioniert im Prinzip übrigens genauso wie die entsprechenden Techniken von C++.

Standardparameter

Wenn Sie eine Funktion haben, bei der einige Parameter meistens mit einem Standardwert belegt werden, so können Sie es dem Aufrufer der Funktion ersparen, diesen Standardwert jedes Mal aufschreiben zu müssen, indem Sie den Standardwert schon in der Funktionsdeklaration angeben:

```
// Eine Liste von Strings wird in einem einzigen String aneinander
// gereiht, optional kann eine maximale Länge der Einzelstrings und ein
// Trennzeichen angegeben werden:
```

```
function MakeString(Einzelwerte: TStringList; Trennzeichen: char = ';';
          MaxLen: Integer = 0) // 0 steht für "keine Längenbegrenzung"
        : String;
```

Wichtig ist für die Deklaration, dass alle Parameter, die einen Standardwert haben, am Schluss der Parameterliste stehen müssen. Dies erlaubt dem Compiler beim Aufruf der Methode, alleine an der Zahl der angegebenen Parameter festzustellen, welche Parameter mit Standardwerten zu belegen sind. Die Beispiel-Funktion kann auf drei verschiedene Arten aufgerufen werden:

```
Str := MakeString(Liste);   // Erzeugung eines Strings nach Voreinstellung
Str := MakeString(Liste, ','); // statt ';' ein ',' zur Trennung verwenden
Str := MakeString(Liste, ',', 30); // zusätzlich die einzelnen Strings
                                   // nach 30 Zeichen abschneiden
```

ür den Aufruf der Funktion gilt eine ähnliche Regel wie für die Deklaration: Parameter dürfen nicht an beliebiger Stelle weggelassen werden, sondern nur am Schluss. Wenn Sie also den dritten Parameter angeben wollen, dürfen Sie den zweiten nicht weglassen, selbst wenn dieser über eine Standardeinstellung verfügt. Im Beispiel müsste also der voreingestellte Wert für den zweiten Parameter wiederholt werden, damit die Voreinstellung für den dritten Wert überschrieben werden kann:

```
Str := MakeString(Liste, ';', 30);
```

ür den Rumpf der Funktion ändert sich nichts, dieser greift genauso auf die einzelnen Parameter zu, als wenn keine Standardwerte existieren würden.

Hinweis: Standardparameter eignen sich auch gut für den Fall, dass Sie eine bereits häufig verwendete Methode um neue Parameter erweitern wollen, die Sie aber nicht in allen schon bestehenden Methodenaufrufen dazuschreiben wollen. Wenn Sie die neuen Parameter an den Schluss der Parameterliste anhängen und mit Standardwerten versehen, mit denen sich die schon bestehenden Aufrufe vertragen, müssen Sie die bestehenden Aufrufe nicht ändern.

berladen von Funktionen

Auch Routinen, die sich wie die vordefinierte *write*-Prozedur mit verschiedenen Parametertypen aufrufen lassen, können Sie nun selbst definieren. Für die *write*-Aufrufe aus dem einleitenden Beispiel hätten Sie z. B. die beiden folgenden Prozeduren deklarieren können:

```
procedure write(var f: file; i: Integer); overload;
procedure write(var f: file; s: String); overload;
```

Da es ja schon Prozeduren gibt, die Integers und Strings in eine Datei schreiben, wäre es sicher sinnvoller, eine Prozedur zu schreiben, die einen eigenen Datentypen in eine Datei schreibt:

```
procedure write(var f: file; Rec: TMeineDatenstruktur); overload;
```

Wichtig ist nur, dass Sie die folgenden Dinge beachten:

- Die *overload*-Direktive muss angegeben werden, damit der Compiler die neue Funktion wirklich nur als Alternative zur bisher bestehenden Funktion ansieht und nicht als *Ersatz* (wobei ein solcher Ersatz nur dann erlaubt wäre, wenn die beiden Funktionen in verschiedenen Gültigkeitsbereichen deklariert worden wären).
- Alle überladenen Funktionen mit dem gleichen Namen müssen anhand der Typen ihrer Parameter oder an ihrem Rückgabetyp eindeutig unterscheidbar sein.
- Standardparameter können die geforderte Unterscheidbarkeit erschweren, da eine Funktion mit einem Standardparameter bereits mit zwei verschiedenen Arten von Parameterlisten aufgerufen werden kann.
- Und schließlich: Es müssen *alle* Funktionen, die als Alternativen gelten sollen, mit *overload* deklariert sein. Da die vordefinierte *write*-Prozedur *nicht* mit *overload* deklariert ist, ist es nicht möglich, sie für den Datentyp *TMeineDatenstruktur* zu überschreiben. Die obige Deklaration von *write(var f: file; Rec: TMeineDatenstruktur)* würde also die vordefinierte *write*-Prozedur *ersetzen* und nicht überladen. Um die normalen Datentypen schreiben zu können, müssten Sie die vordefinierte Prozedur dann mit *System.write* aufrufen. Entsprechendes gilt auch für Methoden und für geerbte Methoden: Auch diese werden nach wie vor von gleichnamigen Methoden verdeckt, sofern nicht beide Methoden mit *overload* deklariert sind.

Wenn schon nicht die *write*-Prozedur, so gibt es doch zumindest ein paar vordefinierte Routinen, die schon als *overload* deklariert sind, so z. B. in der *SysUtils*-Unit:

```
function Min(A,B: Integer): Integer; overload;
function Min(A,B: Int64): Int64; overload;
function Min(A,B: Double): Double; overload;
```

2.5.5 Prozedurtypen

Die auf Methodenzeiger erweiterten Prozedurtypen spielen in der CLX eine Schlüsselrolle bei der Verteilung der Nachrichten: Wenn Sie im Objektinspektor dem Ereignis *OnClick* eine Methode zuweisen, setzen Sie in Wirklichkeit das Property namens *OnClick* auf einen Zeiger zu dieser Methode. Alle Properties auf der Ereignisseite sind derartige Zeiger auf Methoden.

Beginnen wir bei den normalen Prozedurtypen: Einen Zeigertyp auf eine Funktion, die eine Fließkommazahl als Parameter erwartet und eine ebensolche als Ergebnis zurückliefert, deklarieren Sie z. B. wie folgt:

```
type
  TRealFunction = function(number: real): real;
```

Der Unterschied zu einer normalen Funktionsdeklaration ist also nur, dass statt eines Funktionsbezeichners *hinter »function«* ein Typenname und ein Gleichheitszeichen *vor function«* steht. Zur Anwendung dieses Typs weisen Sie ihn einfach einer Variablen zu und setzen diese Variable auf eine Funktion, deren Parameter und Rückgabewerte mit denen der obigen Deklaration übereinstimmen. Im folgenden Beispiel wird der Prozedurparameter *Funktion* mit dem Typ *TRealFunction* deklariert:

```
procedure ZeichneKurve(Funktion: TRealFunction);
function Funktion1(x: real): real;
function Funktion2(x: real): real;
...
ZeichneKurve(Funktion1);
ZeichneKurve(Funktion2);
```

Die beiden Funktionen müssen so deklariert sein wie oben gezeigt, nur der Name des Parameters darf abweichen.

Methodenzeiger

Die kleine Erweiterung, die die schon etwas älteren Prozedurtypen für das OOP erfahren haben, besteht aus den beiden Schlüsselwörtern *of* und *object*. Der Typ für Methoden, die das *OnClick*-Ereignis bearbeiten können, ist beispielsweise wie folgt deklariert:

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

Methodenzeiger sind nicht mit Zeigern auf normale Funktionen kompatibel, da Methoden zusätzlich einen *self*-Zeiger auf das Objekt als unsichtbaren Parameter benötigen (zu *self* siehe Kapitel 2.2.4) und da Methodenzeiger neben der Methode noch das Objekt speichern, von dem sie stammen, und das später zum *self*-Parameter wird (dies ist ein wesentlicher Vorteil gegenüber den Methodenzeigern von C++).

2.6 Fehlerbehandlung mit Exceptions

Seit der Einführung von Delphi können auch Pascal-Programmierer das tun, was vorher verschiedenen Präsidenten, Regierungen und C++-Programmierern vorbehalten war: Ausnahmezustände verhängen. Die allgemeine Bezeichnung als *Ausnahmen* kann Sie daran erinnern, dass Fehlersituationen nicht das einzige Einsatzgebiet der *Exceptions* sind, sondern dass damit beispielsweise auch besonders freudige Ereignisse behandelt werden können (beispielsweise, wenn sich die Größe der Festplatte völlig unerwartet verzehnfacht).

Die besondere Natur der Exceptions

Exceptions sind völlig anders als alle bisher behandelten Kontrollstrukturen: Sie führen, wenn tatsächlich eine Ausnahmebedingung auftritt, zu einem komplexen Programmablauf, der ohne Exceptions nur durch zusätzliche Funktionsparameter und die Kombination vieler anderer Kontrollstrukturen oder *goto*-Anweisungen durchführbar wäre. Innerhalb dieses komplexen Systems werden nun Objekte befördert, deren Aufbau viel einfacher ist als beispielsweise der Aufbau der Komponenten. Trotz des einfachen Aufbaus handelt es sich um echte Objekte, deren Klassen in einer Klassenhierarchie angeordnet sind, die fast komplizierter ist als die Objekte selber. Tatsächlich dienen die Objekte oft nur dazu, eine bestimmte Klasse zu haben, aber keinen wichtigen Inhalt. Mit diesen Merkwürdigkeiten müssen Sie rechnen, wenn Sie das Folgende lesen.

2.6.1 Verhängung des Ausnahmezustandes

Wir beginnen damit, den Ursprung einer Exception zu betrachten, und versetzen uns in die Lage einer Funktion, die versucht, eine Datei zum Lesen zu öffnen, die jedoch nicht existiert. Angenommen, sie ruft eine Funktion *CallDosInterrupt* auf, die frei erfunden ist und nur als Beispiel dient:

```
(* Datei vom Betriebssystem öffnen lassen *)
ErrorCode := CallDosInterrupt(xyz, Dateiname);
(* Fehlerstatus überprüfen: *)
if ErrorCode <> 0 then
  raise EFileNotFound.Create('Datei konnte nicht gefunden werden.',
                             ErrorCode);
```

Die letzte Zeile ist der Ursprung einer Exception der Klasse *EFileNotFound*. Die eigentliche Erzeugung des Exception-Objekts läuft so ab wie bei jedem anderen Objekt auch: über den *Create*-Konstruktor der zugehörigen Klassen. Um den besonderen Programmablauf der Ausnahmebehandlung zu starten, ist zusätzlich das Schlüsselwort *raise* notwendig. Mit der Bedeutung von »Erheben« zeigt dieses bereits gut an, was geschieht: Die aktuelle Funktion wird verlassen und die Exception eine Ebene höher an die aufrufende Funktion übergeben. Wie diese darauf reagieren kann, zeigt das Kapitel 2.6.4.

2.6.2 Exception-Klassen

Statt für jeden Fehler einen Fehlercode zur Verfügung zu stellen, zu dem es dann womöglich noch eine Fehlermeldung gibt, enthalten die Units der CLX für jeden Fehlertyp eine eigene Exception-Klasse. Jede davon ist direkt oder indirekt von der Klasse *Exception* abgeleitet, die in der Unit *SysUtils* deklariert wird.

Daten der Exceptions

Die Klasse *Exception* enthält das, was alle Exceptions übereinstimmend benötigen: einen String für die Fehlermeldung, den Sie über das Property *Exception.Message* ansprechen können, und eine Hilfekontextnummer, die den Benutzer vom Hilfeschalter des Fehlermeldungsfensters in die Hilfedatei führen kann.

In der Vererbungshierarchie kann nun jede Exception-Klasse diese Meldung mit weiteren Informationen ergänzen. Von den in Kylix vordefinierten Exception-Klassen machen jedoch nur wenige von dieser Möglichkeit Gebrauch, so etwa die Klassen *EInOutError* und *EOsError*, die das zusätzliche Datenelement *ErrorCode* für den Fehlercode des Betriebssystems enthalten. Die einzige Information, die Sie ansonsten neben der Fehlermeldung und dem Hilfekontext haben, ist die Klasse, zu der die Exception gehört. So kann beispielsweise bei Divisionen die Exception *EZeroDivide* auftreten, die zur Klasse der *EMathError*-Exceptions gehört.

Eigene Exceptions definieren

Alle vordefinierten Exceptions können Sie natürlich auch in Ihrem Code erzeugen. Wenn in Ihrem Programm ein neuer Fehlertyp auftreten kann, den Sie ebenfalls in die Ausnahmebehandlung mit aufnehmen möchten, können Sie auch eigene Exception-Klassen deklarieren, die Sie wahrscheinlich in die schon existierende Hierarchie einfügen möchten, z. B.:

```
type
  ETableOverflow = class (EOutOfMemory)
    public
      TableSize: LongInt;
  end;
```

Diese Exception könnten Sie beispielsweise dann »erheben«, wenn in einer internen Tabelle des Programms kein Speicher mehr vorhanden ist, sei es, weil die Tabelle statisch ist oder weil kein zusätzlicher dynamischer Speicher mehr vorhanden ist, durch den die Tabelle erweitert werden könnte. Sie könnten auch direkt die Klasse *EOutOfMemory* verwenden, wenn Sie das oben deklarierte Datenelement *TableSize* nicht benötigten, aber dann könnten Sie nicht zwischen Ihrer und den vordefinierten Exceptions unterscheiden (zur Abfrage der Exception-Klasse siehe Kapitel 2.6.4).

Dieses Beispiel zeigt auch, warum Exception-Klassen so wenige Daten enthalten: *TableSize* wird höchstwahrscheinlich ein unnötiges Element sein, weil das Programm auf andere Weise feststellen kann, wie groß seine eigene Tabelle ist.

2.6.3 Schadensbegrenzung mit finally

Wir betrachten hier allgemein alle Funktionen, die die Exceptions nicht abschließend behandeln, sondern an die übergeordnete Funktion weitergeben (dazu gehört meistens auch die Funktion, die die Exception erzeugt hat). In vielen Fällen genügt es nämlich nicht, dass die Funktion einfach abgebrochen wird: Wenn diese in ihrem bisherigen Verlauf schon einige Ressourcen reserviert hat, sollte sie diese auch dann freigeben, wenn eine Exception auftritt. Ohne Exceptions hätte der Code dann beispielsweise so aussehen können:

```
Speicher := GetMem(10000); { Belegen der Ressource }
...
if not OpenFile(...) then begin { Fehlerquelle }
   FreeMem(Speicher);  { Notfreigabe der Ressource }
   exit;
end;
{ weitere Anweisungen }
FreeMem(Speicher, 10000);  { normale Freigabe }
```

Das Beispiel setzt voraus, dass *OpenFile* einen Fehlschlag beim Öffnen durch die Rückgabe des Wertes *False* anzeigt. Nachdem der obige Code festgestellt hat, dass ein Fehler aufgetreten ist, kann er selbst entscheiden, ob er trotzdem weiterarbeitet oder ob die Funktion mit *exit* verlassen werden soll. Im letzteren Fall muss natürlich vorher der Speicher wieder freigegeben werden.

Würde *OpenFile* das Scheitern ihrer Bemühungen jedoch nicht über das Funktionsergebnis, sondern mit einer Exception bekannt geben, müsste das Beispiel wie folgt umgeschrieben werden, um die Freigabe des Speichers weiterhin sicherzustellen:

```
Speicher := GetMem(10000);
...
try
  OpenFile()...
  { weitere Anweisungen }
finally
  FreeMem(Speicher, 10000)
end;
```

Die Anweisungen, die nach *OpenFile* und vor *finally* noch folgen, werden bei einer Exception immer übersprungen, hier hat das Programm keine Wahlmöglichkeit mehr wie im ersten Beispiel ohne Exceptions.

Der Ablauf einer *try...finally...end*-Konstruktion ist wie folgt: Tritt eine Exception innerhalb des von *try* und *finally* umgebenen Blocks auf, wird der Block sofort verlassen und die Anweisungen zwischen *finally* und *end* werden ausgeführt. Wenn keine Exception auftritt, läuft das Programm so ab, als wären die Wörter *try* und *finally* gar nicht anwesend: Zuerst wird der gesamte *try*-Block ausgeführt, dann der *finally*-Block.

Geschützte Ressourcen

Dadurch, dass die Ressourcen, die vor dem *try* reserviert wurden, auf jeden Fall freigegeben werden (falls Sie die entsprechenden Anweisungen in den *finally*-Teil geschrieben haben), erhält der *try*-Block die Bezeichnung *geschützter Block*. Wenn Sie innerhalb dieses Blocks weitere Ressourcen reservieren, die ebenso geschützt werden müssen, önnen Sie diese Kontrollstruktur auch schachteln:

```
AllocateRes1 { 1. Ressource }
try { Schutzblock für die 1. Ressource }
  ...  { Position (*) }
  AllocateRes2; { 2. Ressource }
  try { Schutzblock für 2. Ressource }
    ...
  finally
    FreeRes2; { Freigabe für Ressource 2 }
  end;
finally
  FreeRes1; { Freigabe der ersten Ressource }
end;
```

ritt in diesem Fall im ersten, aber noch vor dem zweiten geschützten Block eine Exception auf (an der mit (*) markierten Position), so wird nur der *finally*-Block der ußeren Ebene ausgeführt, der innere *finally*-Block ist auch nicht nötig, weil die zweite Ressource noch gar nicht reserviert ist. *Finally*-Blöcke werden vom Programm also erst beim Eintritt in den zugehörigen *try*-Block für die unbedingte Ausführung vorgemerkt und dann auf jeden Fall ausgeführt – übrigens auch dann, wenn Sie den *try*-Block mit *exit* verlassen.

2.6.4 Behandeln der Exceptions

Mit den im letzten Abschnitt beschriebenen Aktionen ist der Ausnahmezustand noch nicht aufgehoben. Nach der Ausführung des *finally*-Blocks wird das Exception-Objekt so lange eine Aufrufebene höher gereicht, bis eine Funktion erreicht wird, die einen *except*-Block zur Verfügung stellt. Alle Funktionen, die das nicht tun, werden entweder ganz oder bis zu ihrem *finally*-Teil übersprungen.

Der *except*-Block (Exception-Handler) dient dazu, den Fehler so zu behandeln, dass das Programm danach wieder normal weiterlaufen kann. Wie schon bei *finally* müssen Sie zuerst mit dem *try* einen Programmabschnitt einleiten, der auf die Exceptions reagieren soll.

Im folgenden Beispiel wird die Exception praktisch an den Benutzer weitergegeben:

```
begin
  try
    OpenFile();
    ...
```

```
  except
    on EFileNotFound do
      MessageBox("Die Datei konnte nicht geöffnet werden.");
  end;
end;
```

Abfragen der Exception-Klasse

Wie das Beispiel zeigt, sind *except*-Blöcke differenzierter als *finally*-Blöcke, denn sie richten sich nach der Klasse der Exception und funktionieren ähnlich wie eine *case*-Anweisung. Hinter dem Schlüsselwort *on* geben Sie die Exception-Klasse an, die Sie behandeln möchten. Wenn die Exception diese oder eine davon abgeleitete Klasse hat, werden die Anweisungen nach dem *do* ausgeführt. Anschließend wird der *except*-Block beendet, auch wenn sich darin noch andere *do*-Klauseln befinden, die mit der Exception übereinstimmen. Das folgende Beispiel zeigt eine *do*-Klausel, die niemals ausgeführt wird, da sie eine Unterklasse einer Klasse abfragt, die schon vorher erledigt wurde:

```
except
  on EDatabaseError do ...
  on EDBEngineError do ...
end;
```

Die beiden *on*-Behandlungen müssen also vertauscht werden, um sinnvoll zu werden. Die zweite *on*-Bedingung trifft dann nur noch die *EDatabaseError*-Exceptions, die keine *EDBEngineError*-Exceptions sind.

Weitere Ähnlichkeiten mit der *case*-Anweisung sind: Mehrere Anweisungen werden in einem *begin...end*-Block zusammengefasst, Sie können mehrere Klassen hinter einem *on* angeben und einen *else*-Zweig für den Fall bereitstellen, dass eine andere Ausnahme auftritt. Letzteres ist jedoch meistens nicht zu empfehlen, da auch Exceptions, die Ihnen vielleicht nicht bekannt sind, auf diese Weise aufgehoben werden. Dadurch ist es einem äußeren Block nicht mehr möglich, die Exception auf die angemessene Weise zu behandeln.

Abfangen des Exception-Objekts

In den obigen Beispielen wurde nur die Klasse der Exception abgefragt. Sie können auch das Objekt selbst, das in der *raise*-Anweisung über einen Konstruktor initialisiert wurde, abfragen. Dazu müssen Sie ihm lediglich einen Namen geben, den Sie hinter *on* angeben, wie bei einer Variablendeklaration:

```
  except
    on E: ETableOverflow do begin
      { E.TableSize ist ansprechbar }
    end;
  end;
```

So können Sie auf die Datenelemente des Exception-Objekts zugreifen, z. B. *Message* für alle Exceptions und *TableSize* für den in einem früheren Beispiel deklarierten Typ *ETableOverflow*.

Es gibt eine weitere Möglichkeit, dieses Objekt zu erhalten, beispielsweise im *else*-Zweig des *except*-Blocks: Die Funktion *ExceptObject* (Unit *SysUtils*) liefert Ihnen das jeweils aktuelle Exception-Objekt als *TObject*-Referenz zurück (oder *nil*, wenn keine Exception vorhanden ist).

Erneuern und Schachteln von Exceptions

Wenn Sie in einem Exception-Handler erneut von der *raise*-Anweisung Gebrauch machen, kann das zwei verschiedene Wirkungen haben:

```
try
 ...
except
  on Exception do begin
    { Alle Ressourcen freigeben, die *nur bei einer Exception*
      nicht mehr gebraucht werden, die aber bei einer fehlerfreien
      Ausführung weiter belegt bleiben sollen. }
     ...
    raise; { Weitere Behandlung vom Aufrufer verlangen. }
  end;
end;
```

Falls Sie *raise* wie in diesem Beispiel ohne Parameter verwenden, erneuern Sie lediglich die Exception, die Sie gerade behandeln. Wenn Sie den Fehlerzustand in Ihrer Funktion nur teilweise beheben können, haben Sie so die Möglichkeit, diesen Teil im *except*-Block durchzuführen und dann die Exception noch einmal weiterzugeben. Das obige Beispiel wirkt also wie ein *finally*-Block, der nur im Fall einer Exception aufgerufen wird.

Ein selten vorkommender Spezialfall bleibt noch: Wenn Sie beim Aufruf von *raise* innerhalb eines *except*-Blocks ein neues Exception-Objekt konstruieren und mit *raise* starten, so ersetzt dieses das bisherige Exception-Objekt, falls Sie es nicht noch innerhalb desselben *except*-Blocks selbst behandeln.

Das Ende einer Exception

Eine Exception gilt als behandelt, sobald ein Exception-Handler gefunden wurde, der die Exception nicht erneuert. Ein solcher Exception-Handler kann drei verschiedene Erscheinungsformen haben:

- Eine *on*-Abfrage, die mit der aktuellen Exception übereinstimmt, beendet den Ausnahmezustand wie beschrieben.
- Der *else*-Teil einer *on*-Abfrage behandelt alle übrig bleibenden Exceptions.

- Ein *except*-Abschnitt kann statt einer *on*-Verzweigungsliste auch nur eine Reihe von Anweisungen enthalten. Durch diese wird der Ausnahmezustand in jedem Fall aufgehoben, so als würde der *except*-Abschnitt statt irgendwelcher *on*-Abfragen nur einen *else*-Teil enthalten.

Funktionen, die keine Exception-Behandlung durchführen, dienen den Exceptions, ohne es zu merken, als Durchgangsstation auf ihrem Weg nach oben. Falls sich keine Behandlung für eine Exception finden lässt, gelangt diese schließlich zur Laufzeitbibliothek zurück, die eine Meldung über die Exception anzeigt, die Bearbeitung der Nachricht, bei der diese Exception aufgetreten ist, abbricht und schließlich die Programmbearbeitung fortsetzt.

Selbst wenn Sie eine Exception nicht bearbeiten, kann Ihre Anwendung auf diese Weise meistens problemlos fortgesetzt werden, wenn Sie alle kritischen Programmteile mit *try..finally* behandelt haben. Wenn der Benutzer, z.B. über einen Schalter, das Ereignis ausgelöst hat, das zur Exception geführt hat, sollte ein sicheres Programm nach der Exception wieder in einem Zustand sein, als hätte der Benutzer den Schalter nie gedrückt. Da die CLX eine Meldung über die Exception anzeigt, kann der Benutzer sich etwas einfallen lassen, wie er den Fehler umgehen kann.

2.6.5 Optionen für Exceptions

Sowohl in den Projektoptionen als auch in den Umgebungseinstellungen finden Sie Optionen, die die Exceptions betreffen. Für das Programm wichtig sind die Compilerschalter der Projektoptionen (Abbildung 1.14). Mit diesen können Sie bestimmen, ob bestimmte Routineüberprüfungen im Fehlerfall in einer Exception resultieren oder nicht. Wenn die folgenden Optionen ausgeschaltet sind, werden die entsprechenden Fehler in Ihrem Programm nicht automatisch entdeckt, wodurch eventuell Folgefehler bis zum Programmabsturz entstehen können. Bei angeschalteter Option erzeugt der Compiler zusätzlichen Code, der bestimmte Bedingungen überprüft und eine Exception erzeugt, falls ein Fehler festgestellt wird. Sie können daher die Operationen, die zu Fehlern führen können, in *try*-Blöcken unterbringen:

- BEREICHSÜBERPRÜFUNG (*$R*): Überprüft, ob die Bereiche von Arrays, Strings und Unterbereichstypen eingehalten werden mit Ausnahme der Prozeduren *Inc* und *Dec*, die durch eine Bereichsprüfung um ein Vielfaches langsamer werden würden. Die Exception-Klasse ist *ERangeError*, abgeleitet von *EIntError*.
- ÜBERLAUFPRÜFUNG (*$Q*): Fängt Überläufe bei Integer-Operationen ab (wenn Sie beispielsweise mit den *Byte*-Variablen *a*, *b* und *c* die Anweisung *a:=b*c* ausführen und *b* und *c* den Wert 100 haben, so ist das Ergebnis zu groß für eine *Byte*-Variable), nicht überprüft werden die Prozeduren *Inc* und *Dec*. Auch hierfür gibt es eine von *EIntError* abgeleitete Exception-Klasse: *EIntOverflow*.

- I/O-PRÜFUNG (*$I*): Testet verschiedene I/O-Operationen auf Fehler, beispielsweise die Standardfunktionen für Dateioperationen wie *Reset*, *Rewrite*, *BlockWrite* etc. Für I/O-Fehler definiert die Laufzeitbibliothek die Klasse *EInOutError*, deren öffentliche *ErrorCode*-Variable weitere Informationen liefert.

  Auch wenn Sie diese Option abschalten, können Sie I/O-Fehler überprüfen, indem Sie die herkömmliche Funktion *IoResult* verwenden, für die hier auf die Online-Hilfe verwiesen sei.

Sie können die genannten Optionen auch für bestimmte Teile des Programms an- und abschalten, indem Sie die Compilerschalter im Programmcode setzen (siehe Kapitel 2.1.3). Auch die Überprüfung auf einen Stack-Überlauf lässt sich an- und abschalten, allerdings führt ein Überlauf bei angeschalteter Option zu einem sofortigen Programmabbruch ohne Exceptions.

Bereichs-, Überlauf- und Stack-Prüfung sollten übrigens nur in der Testphase eines Programms eingesetzt werden, da sie das Programm verlangsamen (und auch die ausführbare Datei vergrößern).

Debugger-Optionen

Unter den Umgebungsoptionen (TOOLS | DEBUGGER-OPTIONEN... | SPRACH-EXCEPTIONS) finden Sie auch die Option BEI DELPHI-EXCEPTIONS STOPPEN, die bewirkt, dass der Debugger das Programm stoppt, sobald eine Exception auftritt. Außerdem können Sie eine Liste von »Ausnahme-Exceptions« angeben, bei denen das Programm auch bei eingeschalteter Stop-Option *nicht* angehalten werden soll.

Mit den Exceptions hängen auch die Haltepunkt-Aktionen zusammen, die das Verhalten des Debuggers bei Eintreten einer Exception automatisch und während des Programmlaufs beeinflussen können (siehe Kapitel 1.7.3).

2.6.6 Exceptions im Beispielprogramm

Als abschließendes Beispiel kommt dieser Abschnitt noch einmal auf die Exception-Behandlung des *AlarmClock*-Beispielprogramms aus Kapitel 1.8 zurück. Eine potenzielle Fehlerquelle ist dort die Bibliotheksfunktion *StrToTime*, die es mit einer *EConvertError*-Exception ahndet, wenn eine Alarmzeit ungültige Zeichen enthält oder nicht den erlaubten Formaten (00:00:00 oder 00:00) entspricht.

Wir untersuchen dies anhand einer Programmversion, die die Alarmzeiten als String speichert und bei jedem Timer-Ereignis diesen String in der Funktion *IsTimeOver* mit *StrToTime* in einen Zeitwert umwandelt, um diesen dann mit der aktuellen Zeit zu vergleichen:

```
function TAlarmForm.IsTimeOver(Alarm : string) : boolean;
var
  AlarmTime : TDateTime;
begin
  Result:=False;
  if Alarm<>'' then begin
    AlarmTime:=StrToTime(Alarm);
    Result:=Time>AlarmTime;
  end;
end;
```

Für die Exception-Behandlung kommen nun zwei Stellen in Frage: In der gezeigten Funktion *IsTimeOver* und eine Aufrufebene höher in der Funktion *TimerTimer*, an die die Exception weitergereicht wird, falls *IsTimeOver* sie nicht behandelt:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
var
  FoundAlarm : string;
begin
  TimeText.Caption:=TimeToStr(Time);
  if AlarmAktiviert then
    if IsTimeOver(Alarmzeit) then
      // Ausgabe einer Meldung etc.
      ...
end;
```

Dazu muss zuerst überlegt werden, wie die Exception überhaupt behandelt werden soll. Eine Möglichkeit wäre, den Eingabefehler einfach zu ignorieren. Dazu müsste der Aufruf von *StrToTime* lediglich in einen *try*-Block eingeklammert werden, dessen *except*-Teil leer bleibt und alle Exceptions aufhebt. Dies würde in der Funktion *IsTimeOver* stattfinden:

```
function TAlarmForm.IsTimeOver(Alarm : string) : boolean;
var
  AlarmTime : TDateTime;
begin
  Result:=False;
  if Alarm<>'' then begin
    try
      AlarmTime := StrToTime(Alarm);
    except
      // die folgende Zeile kann auch weggelassen werden,
      // dann werden alle Exceptions ignoriert:
      on E: EConvertError do ; // nur EConvertError ignorieren
    end;
    Result:=Time>AlarmTime;
  end;
end;
```

Stattdessen soll das Programm aber eine Fehlermeldung ausgeben und die Alarmfunktion abschalten, damit der Benutzer den Fehler korrigieren kann. Diese Tätigkeit

liegt nicht im Zuständigkeitsbereich der Methode *IsTimeOver*, denn diese soll lediglich entscheiden, ob die Weckzeit vorüber ist, und keine weiteren Nebeneffekte haben.

Daher behandelt *IsTimeOver* die Exception nicht; die Exception wird also eine Ebene öher zu *TimerTimer* gereicht, die sie mit dem folgenden Exception-Block behandelt:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
begin
  TimeText.Caption := TimeToStr(Time);
  try
    ...
     if IsTimeOver(AlarmEdit.Text) then
    ...
  except
    on E: EConvertError do begin
      AlarmActive.Checked := False;
      MessageDlg('Ungültige Eingabe, bitte korrigieren:'+sLineBreak
                 + E.Message, mtError, [mbOK], 0);
    end;
  end;
end;
```

Ein *finally*-Block ist hier nicht notwendig, da *TimerTimer* keine Ressourcen anfordert, insbesondere nicht vor dem Aufruf der Funktion, aus der eine Exception »herauskommen kann«, der Funktion *IsTimeOver*.

2.7 Interfaces

Dieses Kapitel widmet sich den Klassen-*Schnittstellen*, die kurz als *Interfaces* (Schnittstellen) bezeichnet werden, nicht zu verwechseln mit den Unit-Schnittstellen, die in Object Pascal ebenfalls durch das Schlüsselwort *interface* eingeleitet werden. In diesem Kapitel bezieht sich die Bezeichnung *Interface* immer auf ein Klasseninterface.

ür den Einstieg können Sie sich ein solches Interface wie eine abstrakte Klasse vorstellen, also eine Klasse, die nur abstrakte Methoden und keine Datenelemente enthält. Die aus diesen Methoden bestehende Schnittstelle kann nun an beliebige (normale) Klassen vererbt werden – zusätzlich zur einfachen Vererbung von Object Pascal.

Historische Anmerkung

Obwohl sich Interfaces hervorragend für allgemeine Zwecke verwenden lassen und eigentlich jeder objektorientierten Programmiersprache gut zu Gesicht stünden (Java gibt ein weiteres Beispiel dafür), diente die Einführung dieser Interfaces in die Sprache Object Pascal ursprünglich nicht in erster Linie dazu, ein besseres Software-Engineering zu ermöglichen. Anlass war vielmehr das Ziel von Borland, in Delphi 3 eine vollständige Unterstützung des Component Object Models (COM) von Microsoft

Windows zu bieten. Hierbei ging es darum, durch die Interfaces die Kommunikation zwischen verschiedenen Anwendungen zu ermöglichen. Da unter Linux das COM gar nicht zur Verfügung steht, können die Interfaces ihre segensreichen Dienste ganz auf die Zusammenarbeit von Klassen *innerhalb* einer Anwendung konzentrieren. Ein kleines praktisches Beispiel zur Verwendung eines Interfaces finden Sie im TreeDesigner (siehe Kapitel 6.7.5).

2.7.1 Abstrakte Basisklassen versus Interfaces

Um den Unterschied der Interfaces zur bisherigen Art der objektorientierten Programmierung in Object Pascal zu verdeutlichen, kommen wir noch einmal auf das Beispiel aus Kapitel 2.3.3 zurück, in dem der Benutzer aus mehreren Klassen, die bestimmte gemeinsame Fähigkeiten haben, eine Klasse auswählen soll, von der das Programm dann zur Laufzeit ein neues Objekt erzeugt.

Schnittstellen-Vererbung ohne Interfaces

Der Einfachheit halber verzichten wir diesmal wieder auf die in Kapitel 2.3.4 zu Hilfe genommenen Klassenreferenzen und erzeugen das Objekt mit einem normalen Aufruf des Konstruktors der gewählten Klasse:

```
var
  Objekt: AbstrakteKlasse;
begin
  case Benutzerauswahl of
    { "Benutzerauswahl" kann z.B. sein: (Sender as TButton).Tag }
     Button1: Objekt := Klasse1.Create;
     Button2: Objekt := Klasse2.Create;
  ...
     Button10: Objekt := Klasse10.Create;
```

Die gemeinsamen Fähigkeiten der verschiedenen Klassen waren in diesem Beispiel auf eine virtuelle Methode *Arbeite* beschränkt, die in einer abstrakten Basisklasse deklariert wurde:

```
type
  AbstrakteKlasse = class
  { Konstruktor Create wird von TObject geerbt }
    procedure Arbeite; virtual;
  end;
```

Nach der Initialisierung von *Objekt* in der obigen *case*-Anweisung konnten wir dieses *Objekt* mit *Objekt.Arbeite* dazu bringen, seine ganz spezielle Version der Methode *Arbeite* auszuführen, wobei zum Zeitpunkt der Kompilierung nicht feststand, ob dies nun die Methode *Klasse1.Arbeite*, *Klasse2.Arbeite* oder noch eine andere Methode sein würde.

In diesem Beispiel haben alle Klassen eine Schnittstelle gemeinsam, die Schnittstelle der Klasse *AbstrakteKlasse*, die aus der Methode *Arbeite* besteht (und diese »Schnittstelle« ist noch kein Interface im Sinne von Kylix!). Voraussetzung für das Funktionieren dieses Beispiels war allerdings, dass alle Klassen, die dem *Objekt* zugewiesen werden konnten, von einer gemeinsamen Basisklasse abgeleitet worden sind, in diesem Beispiel der Klasse *AbstrakteKlasse*.

Was ist ein Interface?

Ein Interface ist genau das, was wir im letzten Beispiel als Schnittstelle der Klasse *AbstrakteKlasse* bezeichnet haben: eine wohldefinierte Menge von Methoden. Um die Schnittstelle von *AbstrakteKlasse* nun als echtes Kylix-Interface zu deklarieren, können wir die Deklaration wie folgt umschreiben (wobei wir dem Interface einen anderen Namen geben, der mit dem für Interfaces üblichen *I* beginnt):

```
type
  IArbeite = interface
    procedure Arbeite;
    // hier können weitere Methoden folgen
  end;
```

Allerdings besteht ein Interface *nur* aus solchen Methodendeklarationen. Damit ist ein Interface »weniger« als eine Klasse, enthält es doch

- keine Variablen und Properties
- keinen Code. Die Methodendeklarationen sind damit sozusagen alle »abstrakt«, nur dass Sie hier nicht das Schlüsselwort *abstract* verwenden müssen wie bei Klassendeklarationen.

Mit den Interfaces von Object Pascal ergeben sich nun die folgenden neuen Möglichkeiten:

- Klassen mit einer gemeinsamen Schnittstelle müssen nicht mehr von einer gemeinsamen Basisklasse abstammen, damit sie ein und demselben (*polymorphen*) Objekt zugewiesen werden können (abgesehen von der Tatsache, dass in Object Pascal alle Klassen von *TObject* abstammen).
- Klassen können mehrere völlig verschiedene Interfaces unterstützen.

Um das obige Beispiel zu vervollständigen: Mit der Schnittstelle *IArbeite* ist es nun möglich, einer Variablen den Typ *IArbeite* zu geben (anstatt *AbstrakteKlasse*):

```
var
  ArbeitsfaehigesObjekt: IArbeite;
begin
  case Benutzerauswahl of
    { "Benutzerauswahl" kann z.B. sein: (Sender as TButton).Tag }
```

```
    Button1: ArbeitsfaehigesObjekt := Klasse1.Create;
...
    Button100: ArbeitsfaehigesObjekt := Klasse100.Create;
```

Der Variablen *ArbeitsfaehigesObjekt* können Sie beliebige Objekte mit einer Klasse, die die *IArbeite*-Schnittstelle unterstützt, zuweisen. Von welchen Klassen diese Klassen abgeleitet sind, spielt zumindest theoretisch keine Rolle. *Klasse1* kann zum Beispiel eine Erweiterung der vordefinierten Klasse *TStringList* sein und *Klasse100* könnte ein externer COM-Server sein, der sich in einer ganz anderen Anwendung befindet, die zufällig auch die *IArbeite*-Schnittstelle unterstützt.

In Kapitel 2.7.4 werden wir zu der Frage kommen, wie ein Interface im Unterschied zu einer Klasse überhaupt implementiert wird. Vorher werden wir mit einem kleinen Beispielprogramm beginnen (Kapitel 2.7.2) und ein paar wichtige allgemeine Dinge klären (Kapitel 2.7.3).

2.7.2 Verwendung eines Interfaces

Ein zweites Beispiel für ein Interface zeigt das folgende Listing, es stammt aus dem Beispielprogramm *InterfaceDemo*, das Sie auf der CD finden:

```
type
  IContainer = interface
    procedure AddElement(const ElementName: string);
    procedure DeleteElement(const ElementName: string);
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Dieses Interface beschreibt eine Schnittstelle, über die Sie einem Containerobjekt Elemente in Form von Strings hinzufügen können (*AddElement*), die Sie später auch wieder löschen können *(DeleteElement*). Mit den letzten beiden Methoden können Sie die Zahl der Elemente sowie den Namen des ersten Elements abfragen. Diese Schnittstelle sagt noch nichts über den Container aus, außer dass er Teile besitzt, die wie in der gezeigten Schnittstelle als Strings angesprochen werden können.

Abbildung 2.1 zeigt die Oberfläche des Beispielprogramms. Zur Laufzeit wählen Sie über die Radioschaltergruppe des Hauptfensters einen von fünf Containern aus, zu dem Sie dann über zwei Buttons das im Editierfeld angegebene Element hinzufügen oder löschen können:

- Die Elemente des Formular-Containers sind Schalter. Jedes »Element«, das mit *AddElement* hinzugefügt wird, wird in einem zweiten Formular als Schalter dargestellt. Entsprechend wird ein Schalter gelöscht, wenn die *DeleteElement*-Methode aufgerufen wird. Dieses an sich sinnlose Beispiel wurde gewählt, um einen möglichst großen Kontrast zum zweiten Containerobjekt zu geben. Zur Implementation dieses Containers kommen wir in Kapitel 2.7.4.

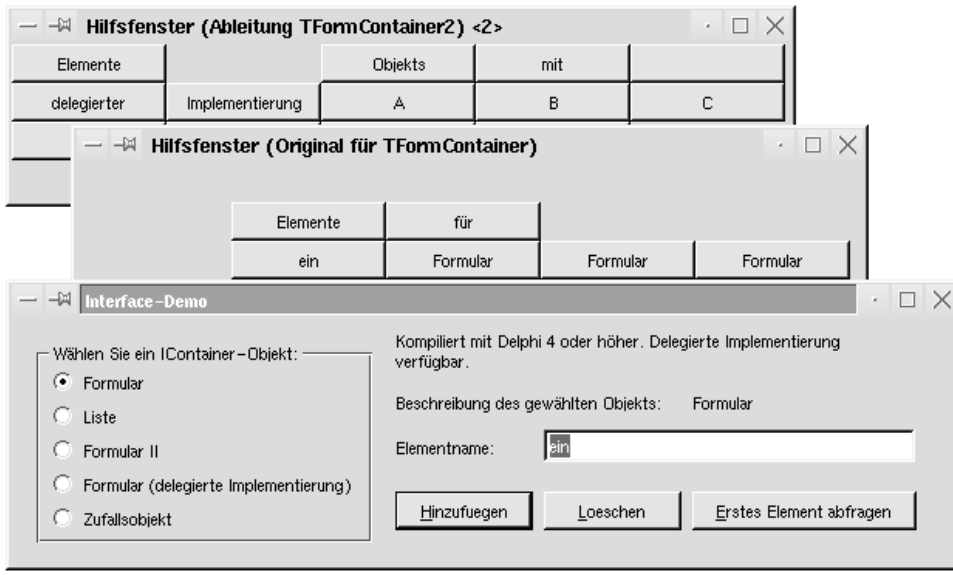


Abbildung 2.1: Die Programmoberfläche für die Verwendung mehrerer völlig verschiedener Objekte, die dasselbe Interface unterstützen (Beispielprogramm InterfaceDemo).

- Dieses zweite Containerobjekt, die »Liste«, merkt sich die mit *AddElement* hinzugefügten Objekt einfach nur, ohne sie irgendwo am Bildschirm anzuzeigen. Als »Beweis«, dass sich die Liste richtig verhält, können Sie ja die einzelnen Elemente mit *GetFirstElementName* abfragen und wieder löschen. Die Interna dieses Objekts werden Sie in Kapitel 2.7.5 kennen lernen.
- Das Containerobjekt *Formular II* verhält sich nach außen hin genauso wie das erste Objekt, ist intern ähnlich wie der Listen-Container realisiert (dazu mehr in Kapitel 2.7.5).
- Das Objekt »Formular (delegierte Implementierung)« verhält sich nach außen hin genauso wie *Formular II*, ist aber intern anders realisiert. Es verwendet die Delegationstechnik, die ebenfalls in Kapitel 2.7.5 erläutert wird.
- Für die ersten vier Containerobjekte definiert das Programm jeweils eine eigene Klasse. Mit *Zufallsobjekt* wählen Sie ein fünftes Objekt aus, das zu Programmbeginn von einer zufällig aus diesen vier Klassen ausgewählten Klasse erzeugt wird.

Nun zum Programm: Die Hauptformular-Unit greift je nach Selektion der Radioschaltergruppe auf eines der folgenden Array-Elemente zu:

```
var
  ContainerArray: array[0..4] of IContainer;
```

Dieses Array ist in einer zweiten Unit namens *Containers* deklariert, in der sich auch die eigentlichen Containerobjekte befinden. Worum es sich bei diesen Objekten handelt, lassen wir weiter offen, denn das ist eine Privatsache der Unit *Containers* (bzw. ihres Implementationsteils). Die Methoden des Formulars kennen nur die anfangs abgedruckte Schnittstellendefinition und dieses vierelementige Array, denn das genügt bereits, um die Methoden für das Hauptformular zu implementieren. Die Methoden für die beiden Schalter sehen beispielsweise wie folgt aus:

```
// ContainerTyp: TRadioGroup; (zwei Radioschalter, siehe Abbildung)
// ElementName: TEdit; (Editierfeld, siehe Abbildung)
// ContainerArray: siehe oben

procedure TForm1.HinzufuegenClick(Sender: TObject);
begin
  ContainerArray[ContainerTyp.ItemIndex].AddElement(ElementName.Text);
end;

procedure TForm1.LoeschenClick(Sender: TObject);
begin
  ContainerArray[ContainerTyp.ItemIndex].DeleteElement(ElementName.Text);
end;
```

Zur Zeit der Kompilierung steht also noch nicht fest, welche Art von Element in diesen Zeilen angesprochen wird. *ContainerArray[0..4]* sind *polymorphe Objekte*, wie wir sie auch bei Verwendung von Klassen schon erhalten haben. Ohne die Verwendung von Interfaces hätte *ContainerArray* auch wie folgt deklariert sein können:

```
var
  ContainerArray: array[0..4] of TContainer;
```

wobei *TContainer* eine Klasse sein könnte. Bei der zu Beginn gezeigten Interface-Deklaration hätten nur das Wort *interface* durch *class* ersetzt und die Methodendeklarationen durch ein *abstract* ergänzt werden müssen, und *ContainerArray* hätte genauso verwendet werden können, wie oben gezeigt.

Der Unterschied zwischen Klasse und Interface ergibt sich hier erst bei der Implementierung, für die wir uns im übernächsten Abschnitt dem Implementationsteil der Unit *Containers* zuwenden werden.

2.7.3 IUnknown, Co-Klassen und andere Begriffe

Während die Verwendung eines existierenden Interfaces so einfach ist, wie im letzten Abschnitt gezeigt, müssen Sie bei der Implementierung von Interface-Klassen und -Objekten einige weitere Begriffe kennen:

- *Interface-Vererbung*: Wie eine Klasse die Elemente einer anderen Klasse erben kann, kann ein Interface die Methoden eines anderen Interfaces erben. Die Syntax für die Vererbung stimmt mit der Vererbungssyntax bei Klassen überein, das Erb-Interface wird also in Klammern hinter dem Schlüsselwort *interface* angegeben.
- *IInterface*: So wie *TObject* der obligatorische Vorfahre aller Klassen ist, gibt es auch für die Interfaces ein Interface, das immer geerbt wird, auch wenn Sie kein Erb-Interface explizit angeben. Dieses Interface heißt *IInterface.*
- *IUnknown* ist ein Alias-Name für *IInterface*. Bis Delphi 5 gab es unter Windows sogar nur den Namen *IUnknown*, da *IInterface* genau dem grundlegenden *IUnknown*-Interface aus dem COM von Microsoft entspricht. Diese Namensgleichheit hat immer wieder zu dem Fehlurteil geführt, dass Delphis Interfaces vom COM abhängig seien. Kylix beweist das Gegenteil, da seine Interfaces unter Linux auch ohne COM sehr gut funktionieren. Die Namensänderung von *IUnknown* in *IInterface* unterstreicht diese Unabhängigkeit.
- Eine *implementierende Klasse* zu einem Interface ist eine Klasse, die die im Interface deklarierten abstrakten Methoden *implementiert*. Object-Pascal-Klassen können gleichzeitig die Methoden von mehreren Interfaces implementieren. Im COM werden solche implementierenden Klassen auch als *Co-Klassen* bezeichnet.
- Wenn Sie von einer Klasse, welche eines oder mehrere Interfaces implementiert, ein Objekt erzeugen (in Abschnitt 2.7.4 werden wir dies tun), dann erhalten Sie ein Objekt, welches Interfaces unterstützt. Unter Windows wird dieses Objekt dann ein *COM-Objekt* genannt. Die über Interfaces verfügenden Objekte einer Kylix-Anwendung unter Linux arbeiten genauso wie Delphis COM-Objekte unter Windows, nur dass sie ihre Funktionalität nicht nach außen, zu anderen Anwendungen oder anderen Rechnern exportieren können, denn hierfür ist ja unter Windows Microsofts (D)COM zuständig.
- Eine *Interface-Variable* (oben z. B. jedes Element von *ContainerArray*) entspricht in Object Pascal einer Zugriffsmöglichkeit für die in einem bestimmten Interface definierten Methoden eines bestimmten Objekts. Mit der Interface-Variablen *ContainerArray[0]* können Sie beispielsweise alle *IContainer*-Methoden des ersten Objekts aufrufen, mit *ContainerArray[1]* die *IContainer*-Methoden des zweiten Objekts. Um Methoden aufzurufen, die zum selben Objekt gehören, aber in einem anderen Interface deklariert sind, benötigen Sie eine weitere Interface-Variable.

Es stellt sich nun die Frage, wie Sie an eine solche weitere Interface-Variable für ein anderes Interface desselben Objekts kommen. Die Antwort dazu finden wir bei einer genaueren Untersuchung des *IInterface*-Interfaces in einer von dessen Methoden.

IInterface

IInterface besitzt drei Methoden, die aufgrund der automatischen Vererbung von *IInterface* in *jeder* Interface-Variablen zur Verfügung stehen. Da ist zunächst die oben bereits angekündigte Methode, mit der sich andere Interfaces desselben Objekts ermitteln lassen:

- *QueryInterface* liefert eine Interface-Variable für eine andere Schnittstelle desselben Objekts zurück. Wenn Sie beispielsweise in *Container: IContainer* die Schnittstelle eines Containerobjekts gespeichert haben und dieses Objekt auch noch eine zweite Schnittstelle unterstützt (beispielsweise *IClassDescription*), können Sie diese Schnittstelle mit *QueryInterface* anfordern: *Window := Container.QueryInterface(IClass Description)*. Besserer Object-Pascal-Stil ist es jedoch, statt *QueryInterface* den Object-Pascal-Operator *as* zu verwenden, der intern auf *QueryInterface* zurückgreift: *Window:=Container as IClassDescription*.

Voraussetzung für die Funktion von *QueryInterface* ist, dass es für das gesuchte Interface eine Kennzahl (Interface Identifier, IID) gibt. Die Angabe einer IID findet am Anfang der Deklaration eines Interfaces in eckigen Klammern wie im folgenden Beispiel statt:

```
type
  IClassDescription = interface
    ['{22910AA3-C028-11D0-9E2F-444553540000}']
    function GetClassDescription: string;
  end;
```

Fragt sich, wie man sich ein solches Ungetüm von Kennzahl ausdenken soll. Glücklicherweise kann man hier das Denken einmal an den Rechner delegieren und drückt im Editor der Kylix-IDE einfach ⇧+Strg+G (für *GUID*). Der Editor fügt dann einen komplett neuen GUID mitsamt innerer Klammerung, Hochkommata und äußerer Klammerung ein. Der GUID wird nach den Regeln des COM generiert, wonach die aktuelle Zeit und technische Daten des gerade verwendeten Computers in die Zahl einfließen. (Dies soll lediglich sicherstellen, dass der GUID weltweit eindeutig ist, es dient nicht etwa dazu, Daten des verwendeten Computers für Spionagezwecke in den GUIDs zu verschlüsseln).

Referenzzählung

Die anderen beiden Methoden von *IInterface* werden in Object Pascal immer automatisch aufgerufen. Für die interne Funktionsweise der Interfaces sind sie jedoch so wichtig, dass Sie sie kennen sollten:

- *_AddRef* benachrichtigt ein Objekt darüber, dass eines seiner Interfaces verwendet wird. Das Objekt erhöht daraufhin einen internen Referenzzähler, der angibt, wie oft das gesamte Objekt zurzeit verwendet wird.

- *_Release* ist das Gegenstück von *_AddRef*, es sagt dem Objekt, dass es seinen Referenzzähler um eins verringern kann. Ist der Zähler bei 0 angekommen, kann sich das Objekt selbst freigeben. Eine Anwendung ruft *_Release* auf, wenn sie ein Objekt nicht mehr benötigt, beispielsweise am Ende einer Prozedur, in der das Objekt als lokale Variable verwendet wurde.

Um die Hintergründe der Referenzzählung etwas verständlicher zu machen, sei es erneut erlaubt, einen Blick auf das COM von Windows zu werfen: Die Referenzzählung ist im COM deshalb so wichtig, weil COM-Objekte mehrfach in völlig verschiedenen Anwendungen verwendet werden können. Die einzelnen Anwendungen dürfen nicht alleine dafür verantwortlich sein, einmal reservierte COM-Objekte freizugeben, denn sie könnten sonst aus Versehen ein Objekt freigeben, das in einer anderen Anwendung noch verwendet wird, wodurch diese andere Anwendung möglicherweise abstürzen würde. Stattdessen ist jede Anwendung lediglich dafür verantwortlich, den Referenzzähler seiner Objekte richtig »zu bedienen«, so dass diese sich von selbst freigeben können, falls notwendig.

Wenn Sie nun Objekte mit Interfaces lediglich innerhalb einer Kylix-Anwendung einsetzen, so können Sie dort dieselben Vorteile nutzen. Wie die verschiedenen COM-Anwendungen können auch verschiedene Objekte innerhalb derselben Anwendung ein Objekt mit Interfaces verwenden, ohne sich um die Freigabe kümmern zu müssen. Der vom Kylix-Compiler erzeugte Code wird automatisch dafür sorgen, dass das Objekt freigegeben wird, sobald der letzte Nutzer des Objekts *_Release* aufruft. Wobei ja wie gesagt auch dieser Aufruf von *_Release* automatisch stattfindet (und zwar am Ende einer Methode, wenn das Objekt eine lokale Variable der Methode ist, oder bei der Freigabe eines Objekts, wenn es sich um die Variable eines Objekts handelt).

Wie unschwer zu erkennen ist, ist eine irrtümliche Freigabe eines Objekts auch mit *_AddRef* und *_Release* nicht ausgeschlossen, denn niemand hält eine Anwendung davon ab, die *_Release*-Methode öfter aufzurufen, als ihr zusteht. Auch innerhalb einer Kylix-Anwendung können Sie ja *_Release* noch manuell aufrufen und dadurch Fehler verursachen. Daher ist der vollkommen automatische Aufruf von *_AddRef* und *_Release* in Object Pascal ein sehr großer Vorteil.

2.7.4 Implementierung eines Interfaces

Nach dieser umfangreichen Theorie soll nun endlich die Frage beantwortet werden, wie Sie in Object Pascal eine Interface-Variable erzeugen. Das folgende Listing zeigt zum Vergleich noch einmal, wie einfach es ist, von einer Klasse zu einem Objekt zu kommen:

```
type TKlasse = class ... end;
var Objekt: TKlasse;
begin
  Objekt := TKlasse.Create;
```

Bei einem Interface ist das nicht so einfach, *IContainer.Create* gibt es also nicht. Zwischen dem Interface und der Interface-Variablen stehen als weitere Stationen noch die implementierende Klasse und das Erzeugen einer Instanz dieser Klasse.

TInterfacedObject als Basis für die implementierende Klasse *R156*

Das in Kapitel 2.7.2 bereits eingeführte Beispielprogramm aus Abbildung 2.1 benötigt für jedes der verschiedenen *IContainer*-Objekte eine eigene Klasse, die die *IContainer*-Schnittstelle implementiert. Das folgende Listing zeigt eine der beiden Klassen – die Klasse für das *IContainer*-Formular:

```
type
  TFormContainer = class(TInterfacedObject, IContainer)
    // FormContainer erzeugt das Formular selbst, sobald es benötigt wird.
    // Welche Formularklasse es verwenden soll, wird im Konstruktor
    // angegeben.
    HostFormClass: TFormClass; // Klasse des Formulars (siehe Konstruktor)
    HostForm: TForm; // Das dynamisch erzeugte Formular, in das die
                     // "Elemente" eingefügt werden.
    AddCount: Integer; { zählt die Zahl der Aufrufe von AddElement }
  public
    constructor Create(AFormClass: TFormClass);
    procedure AddElement(const ElementName: string);
    procedure DeleteElement(const ElementName: string);
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Die erste Zeile dieser Deklaration ist bereits die wichtigste: *IContainer* in den Klammern hinter dem *class*-Schlüsselwort bedeutet, dass diese Klasse die *IContainer*-Schnittstelle implementieren wird. Die Angabe von *TInterfacedObject* bewirkt, dass *TFormContainer* von der in Kylix vordefinierten Klasse *TInterfacedObject* abgeleitet wird. Sinn von *TInterfacedObject* ist es, eine Implementation der drei obligatorischen *IInterface*-Methoden bereitzustellen. Wenn Sie Ihre Implementationsklasse also von *TInterfacedObject* ableiten, brauchen Sie keinen Referenzzähler zu deklarieren und keine eigenen *QueryInterface-*, *_AddRef-* und *_Release*-Methoden zu schreiben.

Kommen wir nun zum »Inhalt« der Klasse aus dem obigen Listing. Außer dem eigenen Konstruktor, der nur dazu dient, den von außen angelieferten Parameter in der Objekt-internen Variablen festzuhalten, sind in der Klasse *TFormClass* nur die Methoden des *IContainer*-Interfaces deklariert. Zur Veranschaulichung sei hier die Methode *AddElement* gezeigt:

```
procedure TFormContainer.AddElement(const ElementName: string);
var
  B: TButton;
begin
  if not Assigned(HostForm) then HostForm := HostFormClass.Create(nil);
  B := TButton.Create(HostForm);
  B.Parent := HostForm;
  // Die Schalter so positionieren, dass kein Schalter verdeckt wird
  // (Lücken, die in DeleteElement entstehen, werden nicht gefüllt):
  B.Width := 100;
  B.Top := (AddCount div 5) * B.Height;
  B.Left := (AddCount mod 5) * B.Width;
  B.Caption := ElementName;
  inc(AddCount);
end;
```

Wie der Code dieser Methode hat auch der Code der anderen Methoden von *TFormClass* nichts mit unserem Thema der Interfaces zu tun, daher können wir die Implementierung der Klasse *TFormClass* hier als gegeben betrachten.

Erzeugung von Objekten mit Interfaces

Um nun ein Objekt der Klasse *TFormContainer* zu erzeugen, müssen wir nur auf das Wissen über Klassen in Object Pascal zurückgreifen:

```
var
  FormContainer: TFormContainer;
begin
  FormContainer := TFormContainer.Create(THelperForm);
```

Wenn Sie sich an Kapitel 2.7.2 erinnern, haben wir dort jedoch nicht mit einer Variablen vom Typ *TFormContainer* gearbeitet, sondern mit einer Variablen des Typs *IContainer*, welche auch das Ziel des aktuellen Abschnitts sein soll. Das Objekt *FormContainer* ist nur ein Zwischenschritt auf dem Weg zu diesem Ziel.

Die Interface-Variablen

Die folgenden Programmzeilen deklarieren eine Interface-Variable *ContainerInterface* und weisen ihr das oben erzeugte Objekt zu:

```
var
  ContainerInterface: IContainer;
begin
  ContainerInterface := FormContainer;
```

Da Sie über die Interface-Variable nicht mehr auf alle theoretisch möglichen Methoden von *FormContainer* zugreifen können, entspricht diese Zuweisung einer Typumwandlung von einer abgeleiteten in eine höhere Klasse und *ContainerInterface* entspricht einem polymorphen Objekt, dessen wahrer Typ nicht bekannt ist.

Diese Polymorphie wird in der Beispiel-Unit *Containers* deutlich, in der das schon in Kapitel 2.7.2 verwendete Array wie folgt initialisiert wird, wobei auch die in den folgenden Abschnitten besprochenen implementierenden Klassen zum Einsatz kommen:

```
var
  Zufall: byte;
initialization
  ContainerArray[0] := TFormContainer.Create(THelperForm);
  ContainerArray[1] := TListContainer.Create;
  ContainerArray[2] := TFormContainer2.Create(Screen);
  ContainerArray[3] := TFormContainerWrapper.Create(Screen);
  Randomize; Zufall := Random(4); // Zufällige Auswahl aus 4 Elementen
  case Zufall of
    0: ContainerArray[4] := TFormContainer.Create(THelperForm);
    1: ContainerArray[4] := TListContainer.Create;
    2: ContainerArray[4] := TFormContainerWrapper.Create(Screen);
    3: ContainerArray[4] := TFormContainer2.Create(Screen);
  end;
end.
```

Für den Programmcode in Kapitel 2.7.2 ist es völlig unerheblich, welche Klassen auf der rechten Seite der Zuweisung genannt werden. Der Compiler achtet lediglich darauf, dass Sie eine Klasse angeben, die die *IContainer*-Schnittstelle implementiert. Das Programm funktioniert also auch noch, wenn Sie die Objekte vertauschen oder alle vier Objekte von zufällig ausgewählten Klassen erzeugen lassen.

Delegierte Implementierung

Eine besondere Technik, mit der Interfaces implementiert werden können, ist die Delegation. Eine Klasse muss die unterstützten Interfaces nicht mehr durch eigene Methoden implementieren, sondern sie darf die Implementation an ein anderes Objekt bzw. andere Objekte delegieren. Im Beispielprogramm wird dies von der folgenden Klasse demonstriert:

```
TFormContainerWrapper = class(TObject, IContainer)
private
  FContainer: TFormContainer2;
public
  property Container: TFormContainer2
                        read FContainer implements IContainer;
  constructor Create; override;
end;
```

Auch *TFormContainerWrapper* soll das *IContainer*-Interface unterstützen. Sie soll aber dessen Methoden nicht selbst implementieren, sondern dies an eine andere *IContainer*-Klasse delegieren, und zwar an die Beispielklasse *TFormContainer2* aus dem nächsten Abschnitt (natürlich könnte ebenso gut auch die bereits bekannte *TListContainer*-Klasse verwendet werden). Ein Objekt dieser Klasse wird in *FContainer* gespeicher. *FContainer* wird im *Create*-Konstruktor von *TFormContainerWrapper* konstruiert:

```
constructor TFormContainerWrapper.Create;
begin
  FContainer := TFormContainer2.Create(Application);
end;
```

Da *FContainer* alle *IContainer*-Methoden unterstützt, kann *TFormContainerWrapper* seine *IContainer*-Verpflichtungen an *FContainer* delegieren. Das Property *Container* dient lediglich dazu, diese Delegation dem Compiler mitzuteilen. Entscheidend ist hierbei das Ende der Property-Deklaration: *implements IContainer* besagt, dass das in diesem Property gespeicherte Objekt die *IContainer*-Schnittstelle implementiert.

Das Property braucht hierfür lediglich lesbar zu sein, theoretisch wäre es auch möglich, dass das zurückgelieferte Objekt (hier also *FContainer*) mehrere Interfaces implementiert, die dann als kommaseparierte Liste hinter *implements* angegeben werden. Eine letzte Variation bestünde darin, statt an ein Objekt wie *FContainer* direkt an eine *IContainer*-Variable zu delegieren.

Das obige Beispiel ist sicher aufgrund seiner geringen Komplexität noch nicht geeignet, die Vorteile der Delegation besonders herauszuheben. Die Beispiel-Klasse *TFormContainerWrapper* kommt ja aufgrund der Delegation mit nur einer einzigen Methode aus – dem oben gezeigten Konstruktor – und wird dadurch zu einer leeren Hülle. Hätten wir es anstatt mit *TFormContainerWrapper* mit einer komplexeren Klasse zu tun, die auch noch andere Aufgaben zu erfüllen hat, so könnte die Delegation eine nützliche Entlastung darstellen.

Ein weiterer Vorteil der Delegation, auf den hier nur am Rande hingewiesen werden soll, besteht darin, dass Sie die Implementierung eines Interfaces zur Laufzeit auswechseln können. Wenn Sie im obigen Beispiel etwa zur Laufzeit ein anderes Objekt an *FContainer* zuweisen, vielleicht sogar eines mit einer ganz anderen Klasse (gefordert ist ja lediglich, dass diese auch die *IContainer*-Schnittstelle unterstützt), dann wechseln Sie quasi einen Teil der Methoden von *TFormContainerWrapper* zur Laufzeit aus.

Hinweis: Würde *TFormContainerWrapper* noch weitere Interfaces unterstützen, so könnten diese zwar theoretisch sehr schön an verschiedene Objekte delegiert werden, allerdings versteckt sich hinter dieser Idee ein größerer Nachteil: Jedes dieser Objekte würde über die Vererbung von Interfaces auch die *IInterface*-Schnittstelle unterstützen und damit eine *QueryInterface*-Methode besitzen. Diese Methode würde aber standardmäßig immer nur die anderen Interfaces desselben implementierenden Objekts zurückliefern, aber kein Interface der anderen Objekte.

2.7.5 Interface-Mix-In

Wir kommen nun zum zweiten Container-Objekt des Beispielprogramms: Dieses soll eine Liste darstellen, welche die *IContainer*-Schnittstelle implementiert, indem es etwa die mit *AddElement* hinzugefügten Strings in dieser Liste speichert. Dieses Listen-Con-

tainer-Objekt wird im Beispielprogramm durch die Klasse *TListContainer* implementiert. Der Unterschied zur im letzten Abschnitt behandelten Klasse *TFormContainer* ist folgender:

- *TFormContainer* implementiert die gesamte »Container-Funktionalität«, indem es ein anderes Objekt (ein *TForm*-Objekt) *verwendet*, um etwa bei *AddElement* einen Schalter zum Formular hinzuzufügen. Die Klasse *TForm* wurde dadurch nicht verändert. *TFormContainer* erbte lediglich die elementare Funktionalität der Klasse *TInterfacedObject*.
- *TListContainer* soll zeigen, wie Sie die Unterstützung eines Interfaces zu einer bestehenden Klasse, die schon fast die gesamte für das Objekt gewünschte Funktionalität enthält, *hinzufügen*. Für die Verwaltung einer Stringliste bietet sich die von Kylix vordefinierte Klasse *TStringList* an, von der wir *TListContainer* auch ableiten werden.

In dieser Variante zeigt sich ein wesentlicher Vorteil der Interface-Technik im Allgemeinen: Verschiedene bestehende Klassen, die überhaupt nichts miteinander zu tun haben, können für eine Unterstützung eines neuen Interfaces erweitert werden (in diesem Beispiel wird die Klasse *TStringList* mit dem Interface erweitert). Wären die Methoden des Interfaces nicht in einem Interface, sondern in einer Basisklasse deklariert, müssten die Klassen alle von dieser Basisklasse abgeleitet werden. Da aber etwa *TStringList* nicht nachträglich von einer anderen Klasse abgeleitet werden kann, wäre auf diese Weise eine Erweiterung von *TStringList* nicht möglich.

Die Klasse TListContainer *R157*

TListContainer soll nun also unsere zweite Klasse werden, die das *IContainer*-Interface unterstützt, und sie soll von *TStringList* abgeleitet werden. Dies führt zu einem kleinen Problem, und zwar kann die Klasse dann nicht mehr von *TInterfacedObject* abgeleitet werden (dies wäre ja Mehrfachvererbung). Da wir uns entschlossen haben, sie von *TStringList* abzuleiten, müssen wir die Funktionalität von *TInterfacedObject* in *TListContainer* neu implementieren.

Das folgende Listing zeigt die Deklaration der Klasse *TListContainer*. Die Klassenelemente setzen sich aus den schon von *IContainer* bekannten Methoden sowie aus den Methoden zusammen, die von der *IInterface*-Schnittstelle gefordert werden, aber diesmal nicht von *TInterfacedObject* übernommen werden können:

```
type
  TListContainer = class(TStringList, IContainer)
  private
    FRefCount: Integer;
  protected
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
```

```
    function _Release: Integer; stdcall;
  public
    procedure IContainer.AddElement = Append;
    procedure DeleteElement(const ElementName: string);
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Die in der *QueryInterface*-Deklaration verwendeten Typen *TGUID* und *HResult* bergen keine großen Geheimnisse: Für den Parameter *IID* können Sie einfach den Namen eines Interfaces angeben, das über die schon erwähnte Kennzahl (IID) verfügt, der Compiler setzt dann statt des Namens diese Kennzahl ein. Auch die innere Struktur des Typs *HResult* ist unwichtig, da ein *HResult*-Rückgabetyp üblicherweise durch »0« oder eine vordefinierte Konstante ausgedrückt wird (im Folgenden bspw. die Konstante *E_NOINTERFACE*). Der Name *HResult* selbst entstammt der COM-Programmierung, wo es sich um einen 32-Bit-Integer handelt, dessen einzelne Bits sich auf verschiedene Bedeutungsbereiche aufteilen.

IInterface-Methoden

Um die *IInterface*-Methoden neu zu implementieren, wurden im Beispielprogramm einfach die von der vordefinierten Klasse *TInterfacedObject* verwendeten Anweisungen bernommen:

```
function TListContainer.QueryInterface(const IID: TGUID; out Obj):HResult;
begin // Speichert einen Verweis auf ein anderes Interface in der
      // Interface-Variablen Obj.
  if GetInterface(IID, Obj) then Result := 0 else Result := E_NOINTERFACE;
end;

function TListContainer._AddRef: Integer;
begin // Erhöht den Referenzzähler des Objekts um eins.
  Result := InterlockedIncrement(FRefCount);
end;

function TListContainer._Release: Integer;
begin // Verringert den Referenzzähler des Objekts um eins.
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then
    Destroy; // Falls Zähler bei 0, Objekt freigeben.
end;
```

Dabei ist *GetInterface* eine der in Kapitel 2.3.6 aufgelisteten Methoden von *TObject*, also quasi eine Grundmethode von Object Pascal. *InterlockedIncrement/Decrement* sind Funktionen der Unit *System*, welche die einfachen Operationen *Inc* und *Dec* mit einer Abfrage des Wertes verbinden und gleichzeitig Konflikte zwischen mehreren Threads vermeiden. Da das Beispielprogramm kein Multithreading verwendet, hätte es statt-

dessen auch *Inc* und *Dec* verwenden können, allerdings wäre der Code dann nicht so schön kurz geworden (siehe auskommentierte längere Code-Version im vollständigen Quelltext auf der CD).

IContainer-Methoden

Bei der Implementation der *IContainer*-Methoden haben wir nun den Vorteil, dass *TListContainer* bereits selbst eine StringList *ist*. Wir müssen also kein *TStringList*-Objekt erzeugen oder freigeben und können alle *TStringList*-Methoden direkt aufrufen. Auch hier soll wieder eine Beispiel-Methode genügen:

```
procedure TListContainer.DeleteElement(const ElementName: string);
begin
  if IndexOf(ElementName)>-1 then
    Delete(IndexOf(ElementName));
end;
```

Zuordnen zwischen Interface- und Klassenmethoden

Für die Methode *AddElement* sieht die Situation besonders günstig aus. Sie müsste eigentlich nur die *Append*-Methode der Stringliste aufrufen, um den String zur Liste hinzuzufügen. Da *Append* aber dasselbe Aufrufformat hat wie die *AddElement*-Methode in *IContainer*, brauchen wir überhaupt keine neue Methode zu schreiben, sondern dem Compiler nur mitzuteilen, dass die geerbte *Append*-Methode als Implementation der *AddElement*-Methode verwendet werden soll. Dies wird in der obigen Deklaration durch die folgende Zeile erreicht:

```
procedure IContainer.AddElement = Append;
```

Diese Möglichkeit der Verknüpfung zwischen unterschiedlich benannten Interface- und Klassenmethoden erlaubt es Ihnen, in Ihren implementierenden Klassen andere Methodennamen zu verwenden als im Interface vorgegeben.

Formularklasse und IContainer-Schnittstelle

Auch der Formularcontainer aus Kapitel 2.7.4 hätte auf diese Weise implementiert werden können, und zwar als Ableitung von einer Formularklasse, die alle Methoden von *IContainer* (inklusive *IUnknown*) neu implementiert:

```
TFormContainer2 = class(THelperForm, IContainer)
private
  FRefCount: Integer; // für die IUnknown-Funktionalität
protected
  // IUnknown-Methoden:
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  ...
  (Mehr dazu im vollständigen Quelltext auf der CD.)
```

Allerdings soll hier nicht ein weiterer Nachteil dieser Vorgehensweise verschwiegen werden: Die Klasse *TFormContainer* ist zur Laufzeit flexibler als die Klasse *TFormContainer2*, da sie mit Formularen unterschiedlicher Klassen arbeiten kann (die zu verwendende Klasse wird ja dem Konstruktor als Parameter übergeben). *TFormContainer2* ist dagegen auf eine bestimmte Formularklasse festgelegt, und zwar auf die Klasse, von der sie abgeleitet ist – in diesem Beispiel also *THelperForm*.

Ob es besser ist, eine Interfaces implementierende Klasse von *TInterfacedObject* abzuleiten und durch Verwendung von Objekten anderer Klassen zu implementieren wie in Kapitel 2.7.4 oder die Interfaces zu einer bestehenden Klasse hinzuzufügen wie in diesem Kapitel, hängt also von der Situation ab: Soll die Klasse möglichst flexibel sein wie *TFormContainer* oder soll sie effektiver implementiert werden können, wie z.B. die Klasse *TListContainer*, bei der wir eine der Interface-Methoden gar nicht zu schreiben brauchten, da sie schon von *TStringList* zur Verfügung gestellt wurde?

Hinweis: Versuchen Sie nicht, eine Klasse direkt von *TForm* abzuleiten, denn *TForm* versucht eine Formulardatei aus der fertig kompilierten Anwendung zu laden, und eine solche Formulardatei erhalten Sie nur, wenn Sie ein neues Formular und damit eine neue Formularklasse im Formulardesigner erzeugen.

ergleich mit der Mehrfachvererbung

Im Zusammenhang mit Object Pascal wird oft die Frage gestellt, ob mit den Interfaces nun die Mehrfachvererbung unterstützt wird. Wenn man unter Mehrfachvererbung die Tatsache versteht, dass dort, wo man in früheren Turbo-Pascal-Versionen nur eine Elternklasse angeben konnte, jetzt mehrere Bezeichner stehen dürfen, dann wäre die Frage mit »ja« zu beantworten, ansonsten wäre die logische Antwort »nein«, denn das letzte Beispiel hat gezeigt, dass eine Klasse auch heute nur von *einer* anderen Klasse abgeleitet werden kann und nur die Variablen und den Code von dieser einen Klasse erben kann. So muss das Beispielprogramm *InterfaceDemo* in zwei Fällen die *IInterface*-Methoden neu implementieren, weil es eine Klasse nicht gleichzeitig von THelperForm *und* von TInterfacedObject ableiten kann.

Auch für Interfaces gilt, dass sie nur von jeweils einem anderen Interface erben können. Wir haben in Object Pascal also die Einfachvererbung von Klassen und die Einfachvererbung von Interfaces vorliegen. Klassen können allerdings *mehrere* Interfaces implementieren, aber dabei *erben* sie von diesen Interfaces überhaupt nichts, sie müssen sich mit ihren Methoden lediglich an die vom Interface vorgegebenen Regeln halten. Erben können sie diese Methoden allerdings von einer Basisklasse (wie z.B. von *TInterfacedObject*).

2.8 Funktionsbereiche der Laufzeitbibliothek

Da sich die in den Kapiteln 2.1 bis 2.7 besprochenen Spracheigenschaften alleine noch kaum sinnvoll einsetzen lassen, ist die Laufzeitbibliothek in Object Pascal, wie in anderen Sprachen auch, besonders wichtig. Daher ist ihr der Abschluss des zweiten Kapitels gewidmet, auch wenn sie nicht zur eigentlichen Sprache Object Pascal gehört.

Da die Bildschirmausgabe und die Eingaben des Benutzers bereits von der CLX abgedeckt werden, erwarten wir von der Laufzeitbibliothek hauptsächlich Dinge wie Funktionen zum Lesen, Schreiben und Verwalten von Dateien, Informationen über die Uhrzeit, Funktionen, die das Arbeiten mit Strings und anderen Datentypen vereinfachen, und mathematische Funktionen.

Natürlich ist es in diesem Kapitel nicht möglich, alle Funktionen der Laufzeitbibliothek auch nur knapp zu beschreiben, daher gibt dieses Kapitel nur einen Überblick über die verschiedenen Funktionsbereiche und erklärt die meisten der in diesem Buch verwendeten Funktionen.

2.8.1 Dateiverwaltung

Zu der Dateiverwaltung gehören unter anderem Funktionen, mit denen Sie

- Verzeichnisse erstellen, löschen und wechseln können (*MkDir*, *RmDir* und *ChDir*)
- Informationen über Dateien erhalten, beispielsweise über Existenz, Größe, Datum und Alter (*FileExists*, *FileSize*, *FileGetDate* und *FileAge*)
- Attribute oder den Namen einer Datei ändern können (z. B. *RenameFile* und *FileSetDate*)
- Strings, die Dateinamensangaben enthalten, manipulieren können: Beispielsweise können Sie die Pfadangabe, den Namen oder die Namenserweiterung extrahieren (mit *ExtractFilePath* und *Application.ExeName* wird beispielsweise in Kapitel 4.6.2 der Pfad ermittelt, in der sich die ausführbare Datei der Anwendung befindet)
- wie im Beispielprogramm aus Kapitel 7.3.6 Verzeichnisse rekursiv auflisten.

Dateiauflistungsfunktionen *R119*

Zum Finden aller Dateien, die bestimmte Kriterien erfüllen, gibt es in Kylix die beiden Routinen *FindFirst* und *FindNext*. Zunächst geben Sie die Kriterien, bestehend aus Dateimaske und Dateiattributen, an die Funktion *FindFirst* und übergeben als dritten Parameter eine Struktur des Typs *TSearchRec*, die das Ergebnis aufnimmt, zu dem der Dateiname und einige wichtige Attribute der Datei gehören:

```
FindFirst('*', faAnyFile, SearchRec);
```

Die Maske * und das Attribut *faAnyFile* sorgen dafür, dass die Funktion die erste aller Dateien zurückliefert. Statt *faAnyFile* können Sie auch *faDirectory* angeben, falls Sie nur an Verzeichnissen interessiert sind. Die anderen *fa...*-Konstanten, die in der Unit *SysUtils* zu finden sind, sind nur unter Windows wirksam, also nicht portabel nutzbar, da sie auf Eigenschaften des DOS-Dateisystems abzielen (simpel schreibgeschützte Dateien, versteckte Dateien, Systemdateien etc). *FindFirst* liefert ein Ergebnis ungleich Null, wenn sie keine Datei finden konnte.

Alle weiteren Dateien suchen Sie mit *FindNext*. Sie brauchen und können die Dateiauswahl-Kriterien nicht noch einmal angeben, *FindNext* schreibt die nächste passende Datei in den *TSearchRec*-Parameter und liefert ein Ergebnis wie *FindFirst*. Nach dem letzten Aufruf von *FindNext* sollten Sie mit *FindClose* dafür sorgen, dass die intern zur Dateisuche verwendeten Ressourcen wieder freigegeben werden.

Die Auflistung aller Dateien eines Verzeichnisses kann wie folgt stattfinden:

```
var
  F: TSearchRec;
  Error: Integer;
begin
  Error := FindFirst('*.*', faAnyFile, F);
  while (Error = 0) do begin
    ... Bearbeitung der Datei "F.Name" ...
    Error := FindNext(F);
  end;
  FindClose(F);
```

Ein Beispiel, das auch etwas mit den aufgelisteten Dateien anstellt, finden Sie wie schon erwähnt in Kapitel 7.3.6.

Hinweis: Die genannten Funktionen heißen zwar so, als wären sie zum Suchen da, werden aber durch Konkurrenz einer anderen Funktion auf ihr Spezialgebiet der Dateiauflistung zurückverwiesen. Diese Funktion heißt *FileSearch* und kann die Dateien sogar in einer Liste von Verzeichnissen suchen.

2.8.2 Dateieingabe und -ausgabe

In Kylix stehen – abgesehen von den Datenbankfunktionen – vorwiegend zwei Systeme zum Speichern von Daten auf Datenträgern zur Auswahl: die in Kapitel 4.3 beschriebenen objektorientierten Streams und die nicht objektorientierten Funktionen der Unit *System*, die noch aus alten Pascal-Zeiten stammen und die an dieser Stelle kurz angesprochen werden sollen.

Die *System*-Funktionen eignen sich für alle Zwecke gut, bei denen die gespeicherten Daten nicht polymorph sind, also nicht aus verschiedenen nicht vorhersagbaren Datentypen einer Klassenhierarchie bestehen.

Besondere Vorteile gegenüber den Streams weisen die Textdateien auf. Sie sind die einzigen Dateien, die die Datei nicht binär, sondern wie eine normale Textdatei, beispielsweise einen Pascal-Quelltext, ansehen und in denen Sie mit *writeln* und *readln* komfortabel auf einzelne Textzeilen zugreifen können.

Selbst wenn Sie für normale Dateioperationen mit Datenträgern nur Streams verwenden, erweisen sich die Textdateien beim Drucken von Text mit Hilfe des Objekts *Printer* als sehr nützlich (siehe Kapitel 6.6.3).

Dateitypen im Einsatz *R113*

Object Pascal kennt die folgenden drei Dateitypen: Textdatei, typisierte Datei (entspricht einem größenmäßig nahezu unbegrenzten Datenarray) und binäre/untypisierte Datei (in der die Daten von Struktur und Inhalt her völlig beliebig sein können).

Das folgende Beispiel zeigt die Unterschiede zwischen den drei Typen anhand des Einlesens einer kompletten Datei (der Datei-Inhalt wird nicht komplett im Hauptspeicher abgelegt, sondern nur kurz in Puffern gespeichert, die immer wieder überschrieben werden):

```
var
  ZeilenPuffer: string;
  Data: LongInt;
  Result: Word;
  LesePuffer: array[0..1023] of Byte;

{ Textdatei: }
  while not eof(Datei) do begin
    { jeweils eine Zeile lesen }
    readln(Datei, ZeilenPuffer);

{ typisierte Datei, bestehend aus LongInts: }
  while not eof(Datei) do begin
    { jeweils einen LongInt lesen }
    read(Datei, X);

{ untypisierte Datei, die mit reset(Datei, 1) geöffnet wurde: }
  repeat
    { jeweils ein kByte lesen }
    blockread(Datei, LesePuffer,
                             1024, { 1024 Datensätze zu je 1 Byte }
                             Result);
  until Result < 1024;
```

Bevor Sie die Dateien wie gezeigt verwenden können, müssen Sie die Datei öffnen und natürlich auch eine Variable deklarieren: Die Dateivariablen werden nach dem folgenden Muster deklariert:

```
var
  StartDatei: System.Text;  { Textdatei }
  DateiVoller32Bit: file of LongInt; { typisierte Datei }
  BinaerDatei: file; { untypisierte Datei }
```

ffnen und Schließen von Dateien

Die grundsätzlichen Schritte zum Öffnen und Schließen einer Datei sind bei jedem der drei Dateitypen gleich: Zuerst müssen Sie die Dateivariable mit einer Datei verbinden *AssignFile*), dann die Datei öffnen (*Reset*, *Rewrite* oder *Append*) und, nachdem die Ein- und Ausgabe abgeschlossen ist, dieselbe wieder mit *Close* schließen:

```
Assign(StartDatei, 'recording.102');
Reset(StartDatei); { je nach Dateityp }
...
Close(StartDatei);
```

Das Öffnen selbst unterscheidet sich wieder von Typ zu Typ:

- Textdateien können mit *Reset* zum ausschließlichen Lesen und mit *Rewrite* zum ausschließlichen Schreiben geöffnet werden. *Rewrite* löscht den bisherigen Inhalt der Datei und erwartet, wie der Name schon sagt, dass Sie den gesamten Inhalt aktualisieren, indem Sie ihn neu schreiben. Um eine Textdatei zu beschreiben, ohne sie zu löschen, verwenden Sie die Prozedur *Append*. Eine mit dieser Prozedur geöffnete Datei ist ebenfalls nur beschreibbar, jedoch werden die neuen Daten an die bestehende Datei angehängt.
- Bei typisierten Dateien entfällt die Prozedur *Append*, da sie nicht mehr notwendig ist: Mit *Reset* geöffnete typisierte Dateien können nämlich auch beschrieben werden. Der einzige Unterschied zwischen *Reset* und *Rewrite* besteht bei dieser Dateiart darin, dass *Rewrite* den bisherigen Dateiinhalt löscht (um Daten an eine mit *Reset* geöffnete Datei anzuhängen, müssen Sie den Positionszeiger der Datei nach dem Öffnen zuerst auf das Dateiende setzen: *Seek(FileSize(F))*).
- Für untypisierte Dateien gilt dasselbe wie für typisierte Dateien, allerdings können Sie den Funktionen *Rewrite* und *Reset* einen zweiten Parameter übergeben. Während bei typisierten Dateien nämlich die Größe eines Datensatzes, der mit *Read* gelesen werden kann (die Blockgröße), schon in der Deklaration der Dateivariablen durch den Basistyp angegeben ist, werden untypisierte Dateien per Voreinstellung in Datensätzen zu je 128 Bytes ausgelesen/beschrieben. Um beim Aufruf der Funktionen *blockwrite* und *blockread* die genaue Byte-Zahl angeben zu können, die gelesen bzw. geschrieben werden soll, müssen Sie den zweiten Parameter von *Reset* auf 1 setzen (Datensatzgröße=1 Byte). Wie im oben gezeigten Beispiel für *blockread* können Sie dann immer noch auch große Datenblöcke am Stück auslesen.

Lesen und Schreiben

Für weitere Informationen zum Schreiben und Lesen der Dateien muss an dieser Stelle auf die Online-Hilfe verwiesen werden (neben den im obigen Beispiel verwendeten Routinen *read*, *readln* und *blockread* benötigen Sie die Routinen *write*, *writeln* und *blockwrite*).

Fehlerbehandlung

Das Öffnen von Dateien ist eine sehr fehleranfällige Operation: Dateien können z.B. schreibgeschützt oder nicht vorhanden, die Pfadangabe kann ungültig sein. Es gibt in Kylix zwei Methoden, diese Fehler abzufangen: über Exceptions und über die Funktion *IoResult* (gegenüber Borland Pascal wurde die strenge Laufzeitfehler-Methode also durch die behandelbaren Exceptions ersetzt). Welche der beiden Methoden verwendet wird, hängt vom Compilerschalter *$I* ab (siehe Kapitel 2.6.5).

2.8.3 Zeitformat und Zeitfunktionen

Kylix speichert Datums- und Zeitangaben in Variablen des Typs *TDateTime*, der intern lediglich eine Fließkommazahl ist, die eine Anzahl von Tagen angibt. So meint beispielsweise die Zahl 1,75 die Zeitspanne von einem Tag und 18 Stunden.

Wenn ein *TDateTime*-Wert ein festes Datum angibt, enthält er die Anzahl der Tage, die seit dem Datum 30.12.1899 vergangen sind. Enthält der Wert eine Zeit, so enthält der Teil nach dem Komma die Uhrzeit als Bruchteil von 24 Stunden.

Hinweis: Um OLE-Kompatibilität herzustellen, wurde das ursprüngliche Format von Delphi 1, bei dem vom Jahre 1 an mit der Zählung begonnen wurde, auf dieses neue Format umgestellt. *TDateTime*-Werte von 16-Bit-Delphi müssen daher um 693594 Tage verringert werden, um in der aktuellen Kylix-Version dasselbe Datum zu erhalten.

TDateTime-Rechnungen

Datum und Zeit können beliebig miteinander kombiniert und sogar verrechnet werden. Ergibt beispielsweise der Ausdruck *EndZeit-StartZeit* den Wert 14, so lagen zwei Zeitmessungen um zwei Wochen auseinander, wobei die Tageszeit nicht berücksichtigt wurde (oder zufällig in beiden Messungen genau übereinstimmte). Ein Ergebnis von 0.000116 gibt einen Zeitraum von ca. 10 Sekunden an.

Zeitroutinen

Im einem Beispielprogramm aus Kapitel 1 wurde bereits die Funktion *Time* verwendet, die die aktuelle Uhrzeit liefert. Entsprechend liefert *Date* das aktuelle Datum und *Now*

die Summe aus beiden. Mit den Funktionen *StrToTime* und *TimeToStr* wurde eine Zeitangabe ebenfalls bereits in Kapitel 1 aus einem String gebildet und in einen String zurücküberführt.

Weitere Funktionen wie *EncodeTime*, *DecodeTime*, *EncodeDate* und *DecodeDate* stellen sicher, dass Sie, ohne mit dem Fließkommawert rechnen zu müssen, auf einzelne Bestandteile der Zeitangabe (Monat, Sekunde etc.) zugreifen können.

Jahrhundertwechsel-feste Programmierung

Bekanntlich bereitete die Umstellung auf das Jahr 2000 mancher Software große Probleme. Der Typ *TDateTime* kann alle vierstelligen Jahreszahlen der Zukunft speichern, daher ist bei der Programmierung Vorsicht nur dann geboten, wenn Sie den Benutzer Jahreszahlen in einem zweistelligen Format angeben lassen, denn dann ist es Interpretationssache, ob sich eine Angabe von »20« nun auf das Jahr 1920, auf 2020 oder vielleicht auf 2120 bezieht (dieser Fall kann ganz unverhofft eintreten, wenn man Ihr Programm länger verwendet als von Ihnen erwartet). Per Voreinstellung würde die CLX in der Funktion *StrToDate* eine »20« bis zum 31.12.2070 als 2020, danach als 2120, und die Jahreszahl »96« noch bis zum Jahr 2046 als »1996« interpretieren.

Sollten Sie mit dieser voreingestellten Interpretationsweise nicht einverstanden sein, önnen Sie sie über die Variable *TwoDigitYearCenturyWindow* ändern, die auf den Wert 50 voreingestellt ist. Ist der Wert dieser Variablen 0, erweitert die CLX eine zweistellige Jahreszahl immer auf das aktuell in der Systemzeit eingestellte Jahrhundert. Bei einem von Null verschiedenen Wert wird unter Umständen auch auf das vorherige oder ächste Jahrhundert erweitert. Nach Möglichkeit wird die Zahl der Vergangenheit zugerechnet, wobei *TwoDigitYearCenturyWindow* angibt, wie viele Jahre die erweiterte Jahreszahl maximal in die Vergangenheit reichen darf. Ein Wert von 1 in *TwoDigitYearCenturyWindow* bewirkt beispielsweise, dass im Jahr 2003 die Angabe »1.1.02« noch als 1.1.2002, der »1.1.01« bereits als 1.1.2101 interpretiert wird.

2.8.4 Formatierungs-/Stringfunktionen

Dieses Kapitel befasst sich mit einer kleinen (überwiegend in den Programmen dieses Buchs verwendeten) Auswahl von Routinen zum Umgang mit Strings.

ormatierung von Strings

Kylix stellt neben *Format* auch einige andere Routinen zur Stringformatierung zur Verügung (beispielsweise für die Verwendung von nullterminierten Strings oder vorgegebenen String-Puffern), allen Funktionen gemeinsam ist jedoch die Formatierung des Strings über einen *Formatstring*. In *Format* geben Sie diesen als ersten Parameter an, beispielsweise:

```
Label.Caption := Format('Testlauf %3d - Diagnose: %s.',[Count, Diag]);
```

Der Formatstring besteht aus einem Textgerüst, das auf jeden Fall in den Ergebnisstring aufgenommen wird, und Platzhaltern, in die weitere Parameter in einem bestimmten Format eingefügt werden. Ein Platzhalter beginnt mit einem »%« und endet mit einem Zeichen, das den Typ des einzufügenden Parameters angibt, beispielsweise »d« für eine Integer-Zahl, »e« für eine Fließkommazahl und »s« für einen String. Zwischen diesem Zeichen und dem »%« kann sich unter anderem eine Angabe befinden, wie viele Zeichen der eingefügte Parameter in Anspruch nehmen soll.

Nach dem Formatstring geben Sie ein in eckigen Klammern eingefasstes Array an, in das Sie für jeden Platzhalter im Formatstring einen Parameter schreiben, der den einzufügenden Wert angibt. Im obigen Beispiel muss es sich bei *Count* um einen Integer-Wert, bei *Diag* um einen String handeln.

Der Ergebnisstring könnte im obigen Beispiel etwa »Testlauf 33 – Diagnose: keine Fehler.« lauten.

Formatieren von Fließkommazahlen und Datumsangaben

Mit den Funktionen *FormatDateTime*, *FormatFloat* und anderen Routinen können Sie genaue Kontrolle darüber ausüben, wie Zahlenwerte und Datumsangaben ausgegeben werden. Die beiden genannten Funktionen arbeiten prinzipiell genauso wie *Format*, erhalten aber einen Formatstring, der nach einem anderen Muster aufgebaut ist, und nur einen einzelnen zu formatierenden Wert (statt des Arrays).

Funktionen für Standard-Strings

Die folgenden Funktionen werden bei einfachen Aufgaben besonders häufig benötigt und stehen teilweise mit *Format* in Konkurrenz: So können Sie mit *IntToStr* eine Zahl schneller in Text umwandeln als mit *Format*; je mehr Zahlen jedoch aneinander gereiht werden, desto umständlicher wird der Umgang mit *IntToStr*, die für jede Zahl einmal aufgerufen werden muss.

Funktion in Kurzschreibweise	**Aufgabe**
function AnsiCompareStr(s1, s2): Integer;	vergleicht zwei Strings unter Berücksichtigung der Groß- und Kleinschreibung und gibt 0 zurück, falls die Strings übereinstimmen. Das Ergebnis ist kleiner als 0, falls *s1* kleiner als *s2* ist, und größer als 0, falls umgekehrt (berücksichtigt wie alle *Ansi...*-Funktionen auch länderspezifische Umlaute).
function AnsiCompareText(s1, s2): Integer;	vergleicht wie *CompareStr*, betrachtet aber Groß- und Kleinschreibung als identisch.
procedure AnsiLowerCase(Str);	wandelt einen String in Kleinbuchstaben um.
procedure AnsiUpperCase(Str);	wandelt einen String in Großbuchstaben um.

Funktion in Kurzschreibweise	Aufgabe
function Copy(Str, Index, Count);	kopiert einen Teil aus *Str* in einen neuen String und liefert diesen als Ergebnis zurück. *Copy* kopiert *Count* Zeichen ab der Position *Index*.
ocedure Delete(Str, Index, Count);	löscht *Count* Zeichen von *Str* ab der Position *Index*. *Delete* ist keine Funktion, sondern ändert den Var-Parameter *Str* direkt.
ocedure Insert(Teilstring, Zielstring, Zielposition);	fügt einen *Teilstring* in einen *Zielstring* ein, dabei gelangt das erste Zeichen des einzufügenden Strings an den Index *Zielposition* im Zielstring.
function IntToHex(Int): String;	wandelt den Zahlenwert in einen String um, wobei sie das hexadezimale Zahlensystem verwendet.
function IntToStr(Int): String;	wandelt den Zahlenwert in einen String um.
function Length(Str): Byte;	liefert die Länge von *Str*.
function Pos(SubStr, Str): Byte;	sucht *SubStr* im String *Str* und liefert die Position des gefundenen Teilstrings oder 0, falls *SubStr* nicht in *Str* enthalten ist.
ocedure SetLength(Str, Laenge);	ändert bei einem langen String die Größe des reservierten Speichers und die Längenangabe selbst; bei einem String im alten 16-Bit-Format passt es nur das Längenbyte an (siehe Kapitel 2.4.4).
function StrToInt(Str): LongInt;	wandelt den String in einen Zahlenwert um und erzeugt eine *EConvertError*-Exception, falls der String fehlerhafte Zeichen enthält.

Neben den Routinen *AnsiCompareStr*, *AnsiCompareText*, *AnsiLowerCase* und *AnsiUpperCase* gibt es noch vier weitere Routinen für die gleichen Aufgaben, die jedoch kein Ansi« im Namen haben. Diese Routinen berücksichtigen die Umlaute der Sprache, auf die das System konfiguriert ist, nicht.

Funktionen für nullterminierte Strings

Die folgende Tabelle listet nur die elementarsten Routinen für nullterminierte Strings auf, mit denen Sie unter anderem einen Übergang zu den Pascal-Strings herstellen önnen:

Funktion in Kurzschreibweise	Beschreibung
function StrLen(Str): Word;	ermittelt die Länge des Strings.
function StrCat (Dest, Source): PChar;	hängt *Source* an *Dest* an.
function StrUpper(Str): PChar;	wandelt einen String in Großbuchstaben um.
function StrLower(Str): PChar;	wandelt einen String in Kleinbuchstaben um.
function StrPCopy (Dest, PasStr): PChar;	kopiert den Pascal-String *PasStr* in den nullterminierten Stringpuffer *Dest*, der vorher bereits reserviert oder als festes Array deklariert worden sein muss.

Wenn eine Stringfunktionen einen *PChar*-Zeiger zurückliefert, ist das meistens kein neuer String (wie bei den Routinen der herkömmlichen Strings), sondern einer der schon als Parameter übergebenen Strings. Sie können die Funktionsaufrufe auf diese Weise einfach verschachteln. (Neue nullterminierte Strings werden von den Funktionen *StrNew* und *StrAlloc* erzeugt, siehe Online-Hilfe.)

2.8.5 Sonstige Funktionen

Die abschließende Tabelle fasst einige ständig benötigte Funktionen der Unit *System* zusammen. Sie sind in den meisten Fällen so einfach, dass der Compiler statt eines Funktionsaufrufs höchstens ein paar Maschinenbefehle zu generieren braucht:

Funktion	**Aufgabe**
function Addr(x): Pointer;	liefert die Adresse einer Variablen, wie der Adressoperator @.
function Assigned(x): Boolean;	liefert *True*, falls der als Parameter übergebene Zeiger nicht *nil* ist, bzw. falls das übergebene Objekt reserviert wurde und die Objektvariable nicht aus einem *nil*-Zeiger besteht.
function Chr(x: Byte):Char;	funktioniert wie die Typenumwandlung Char(x).
procedure Dec(var x);	verringert x um eins.
procedure Dec (var x, n: LongInt);	verringert x um *n*.
procedure FillChar (var x; Count; Val);	füllt *Count* Bytes von *x* mit dem Byte-Wert *Val*.
function High(x);	liefert den höchsten Index eines offenen Arrays oder den größten Wert eines Aufzählungstyps.
procedure Inc(var x);	erhöht x um eins.
procedure Inc (var x, n: LongInt);	erhöht x um *n*.
function IncludeTrailingPathDelimiter(s): String	in den Beispielprogrammen dieses Buchs häufig verwendete plattformunabhängige Funktion, die einen Pfadbegrenzer (unter Windows »\«, unter Linux »/«) an eine Pfadangabe anhängt, falls noch kein solcher vorhanden ist.
function Low(x);	liefert den zu *High(x)* gehörenden niedrigsten Wert.
function Ord(x): LongInt;	liefert die Ordinalzahl des Parameters und kommt einer Typenumwandlung mit *LongInt(x)* gleich.
function ParamCount: Word;	liefert die Zahl der Parameter, die die Anwendung als Start- oder Befehlszeilenparameter erhalten hat. (Startparameter in der Kylix-IDE: START | PARAMETER ...)
function ParamStr(i): String;	liefert einen bestimmten Befehlszeilenparameter als String zurück.
function Random(Grenze): Word;	liefert eine nicht negative ganze Zufallszahl, die kleiner als *Grenze* ist.

Funktion	Aufgabe
ocedure Randomize;	initialisiert den Zufallszahlengenerator.
function Round(r): LongInt;	rundet einen Fließkommawert auf eine ganze Zahl.
function SizeOf(x);	liefert die Größe einer Variablen oder eines Typs.
function Trunc(r): LongInt;	schneidet den Nachkommateil einer Fließkommazahl ab und gibt das Ergebnis als LongInt zurück.
function UpCase(c): Char;	wandelt das Zeichen *c* in einen Großbuchstaben um (ohne Umlaute).

3 Die Komponentenbibliothek CLX

Die CLX (*X* für *cross*-platform Component Library) ist unter Linux der Nachfolger der VCL (Visual Component Library), die unter Windows fünf Delphi-Generationen lang *die* visuelle Komponentenbibliothek war. Unter Windows kommt die CLX als Ergänzung zur VCL hinzu – für die Entwickler, die mit Delphi 6 plattformunabhängige Programme schreiben wollen. Dieses Kapitel konzentriert sich ausschließlich auf die CLX. Wenn Sie die VCL bereits kennen, wird Ihnen sehr vieles schon bekannt vorkommen, denn die CLX weicht nur in ihrem internen Aufbau sehr von der VCL ab. Die äußere Schnittstelle hat Borland so weit wie möglich an die VCL angepasst, um eine einfache Portierung von der VCL zur CLX zu ermöglichen.

Man kann die CLX auf verschiedene Weise definieren: Ihrem Namen der »Komponentenbibliothek« nach handelt es sich um eine Sammlung von Komponenten. Alle grundlegenden Komponenten aus dem Lieferumfang von Kylix sind der CLX zuzurechnen, wobei hier drei Kernbereiche zu nennen sind: die *VisualCLX* (allgemeine Komponenten), *DataCLX* (Komponenten für Datenbankanwendungen) und die *NetCLX* (Internet-Komponenten). Die CLX ist jedoch weit mehr als eine normale austauschbare Komponentensammlung, denn unter der Oberfläche definiert sie die allgemeine Funktionsweise der Komponenten, einer Kylix-Anwendung und der Einbindung der Komponenten in die Kylix-IDE (dieser Teil der CLX wird auch als *BaseCLX* bezeichnet). Während Sie also die Komponenten der Komponentenpalette alle austauschen könnten, bleibt der Kern der CLX ein grundlegender Baustein einer jeden Kylix-Anwendung.

Die Aufteilung in die vier Bereiche *BaseCLX*, *VisualCLX*, *DataCLX* und *NetCLX* ist übrigens auch technisch sichtbar: Jeder Bereich lässt sich optional über ein eigenes Runtime-Package zur Anwendung hinzubinden (wobei hier auch noch das Package *netdataclx* hinzugehört, für die internetfähigen Datenbankkomponenten).

Kapitelüberblick

Dieses Kapitel beschäftigt sich sowohl mit den allgemeinen Grundlagen der CLX und einigen interessanten Aspekten ihrer internen Funktionsweise als auch mit speziellen CLX-Komponenten, die Sie in der Komponentenpalette finden: Der Übergang vom Allgemeinen zum Speziellen vollzieht sich dabei wie folgt:

- Die Abschnitte 3.1 bis 3.3 beschreiben Grundlagen der CLX wie die Beziehungen zwischen den Komponenten und die interne Verarbeitung der Nachrichten. Auch einige grundlegende Klassen werden näher betrachtet: die nicht-visuellen Klassen *TApplication* und *TScreen* und die abstrakten Klassen, die ihre Methoden, Properties und Ereignisse an die Komponenten der Komponentenpalette vererben.
- In den Abschnitten 3.4 und 3.5 geht es dann um den visuellen Grundbaustein jeder Kylix-Anwendung: das Formular. Während Kapitel 3.4 sich mit der Klasse *TForm* befasst und beschreibt, wie Sie mit den Objekten (Formularen) dieser Klasse umgehen (z.B. wie Sie sie dynamisch erzeugen und modal anzeigen können), erläutert Kapitel 3.5 allgemeine Techniken, mit denen Sie Formulare aufbauen können: Gruppieren von Steuerelementen, Verteilen der Komponenten auf mehrere Seiten etc. Kapitel 3.5 schließt mit der Verwendung der Standarddialoge, die Ihnen die Herstellung von alltäglichen Formularen ersparen.
- Der Abschnitt 3.6 beschäftigt sich mit einigen komplexeren Komponenten, die in Kapitel 1.8 noch nicht ausreichend gewürdigt werden konnten.
- Abschnitt 3.7 schließlich befasst sich mit Vereinfachungen des Formular-Entwurfs und behandelt verschiedene Techniken, mit denen gemeinsame Komponentengruppen in Formularen definiert werden können, hauptsächlich die Formularvererbung und die Frames.

Das Kapitel soll Ihnen einerseits in einer an der Vererbungshierarchie ausgerichteten Struktur die Elemente der CLX vorstellen, die besonders wichtig sind, als auch Ihnen einen Überblick über die seltener benötigten Funktionsbereiche der CLX geben, so dass Sie danach in der Lage sind, auch diese Bereiche schnell mit Hilfe der Online-Referenz zu erschließen.

Die CLX und die X Window-Bibliothek

Grundlage der grafischen Benutzeroberfläche unter Linux ist die X Window-Bibliothek. Da diese Bibliothek – im Gegensatz etwa zum Windows-API – keine Steuerelemente definiert, setzen die meisten GUI-Programme unter Linux nicht direkt auf X Window auf, sondern verwenden als Hilfsmittel ein Toolkit, welches mit Hilfe der von X Window bereitgestellten Grundfunktionen zumindest die wichtigsten Steuerelemente definiert. Ein relativ neues und modernes Toolkit ist die Qt-Bibliothek von Trolltech, die Borland als Fundament für die CLX verwendet.

Wichtige Gründe für die Wahl von Qt waren, dass Qt plattformunabhängig und beispielsweise auch für Windows verfügbar ist und dass Qt nach Prinzipien entworfen wurde, die den Entwicklern von Borland sympathisch waren (wer die technische Qualität der VCL kennt, ahnt vielleicht, dass Borland eine möglichst hochwertige Bibliothek mit hervorragendem Design als Grundlage für die CLX nehmen würde).

Außerdem spielt Qt unter Linux bereits eine wichige Rolle, denn es dient auch dem populären KDE als Toolkit (ist jedoch wie Kylix von diesem unabhängig).

Die CLX und Qt

Auch wenn bereits die Qt objektorientiert ist und Komponenten in Form von Klassen bereitstellt, können diese Komponenten nicht ohne weiteres als Kylix-Komponenten verwendet werden. Zum einen ist Qt in C++ geschrieben und von daher nicht ohne weiteres von Object Pascal heraus ansprechbar, zum anderen soll die Komponentenbibliothek von Kylix unter Linux ja weitgehend mit der unter Windows bewährten VCL kompatibel bleiben. Von der VCL-Kompatibilität hätten nicht nur die Entwickler von Windows-Programmen große Vorteile, da sie ihre Anwendungen leicht nach Linux portieren könnten. Auch Borland selbst könnte dadurch erheblichen Neuentwicklungsaufwand sparen, denn die Entwicklungsumgebung von Delphi selbst, ihre gesamte RAD-Architekur zur »schnellen Anwendungsentwicklung« mit den »Two-Way-Tools« hängt ja unter Windows auch von der VCL ab und müsste neu geschrieben werden, wenn die VCL durch etwas völlig Andersartiges ersetzt werden würde.

So genießt also die VCL-Kompatibilität der CLX einen höheren Stellenwert als die Wiederverwendung der Qt-Klassen und daher hat Borland auch einige Funktionsbereiche, die bereits von Qt bereitgestellt werden, in der CLX neu implementiert. Was die Komponentenebene betrifft, stellt die CLX für manche Qt-Komponenten nicht viel mehr als eine Hüllkomponente zur Verfügung (z.B. im Falle von *TTextViewer* und *TTextBrowser*), viele CLX-Komponenten sind jedoch mehr oder weniger neu in Object Pascal geschrieben und verwenden lediglich die plattformunabhängigen Grafikausgabe-Routinen von Qt, um sich am Bildschirm zu zeichnen (z.B. *TToolBar* und *TStatusBar*). Die »zentrale Infrastruktur« der Nachrichtenschleife der Qt (siehe Kapitel 3.1.4) und damit auch die grundlegenden Qt-Klassen wie *QWidget*, *QEvent*, *QObject* werden aber unter Kylix genutzt.

Wenn Sie nun also eine Anwendung mit Kylix entwickeln, machen Sie sich bewusst oder unbewusst alleine im Bereich der Bildschirmdarstellung drei Schichten von Bibliotheken zunutze, von denen jede Schicht neue Annehmlichkeiten mit sich bringt:

- Zunächst stellt X Window die grundlegenden Funktionen für Grafikausgabe bereit und gestattet dabei z.B., dass die Anwendung auf einem anderen Rechner läuft als dem Rechner, auf dem die Bildschirmausgabe stattfindet (die Bildschirmausgabe kann auf ein beliebiges X Terminal umgeleitet werden, was allerdings auf der typischen Kylix-Plattform, die wohl ein einzelner Intel-kompatibler PC sein wird, nicht ausprobiert werden kann).
- Die Qt-Bibliothek baut aus simplen Grafikausgabefunktionen komplette Steuerelemente zusammen, die als Bauelemente für Fenster, allgemein für die Oberfläche von Anwendungen, dienen können.

- Die CLX schließlich formt diese Bauelemente in RAD-taugliche Komponenten um, die sehr leicht zu programmieren und kompatibel mit der bewährten VCL sind.

Die folgende Tabelle zeigt den Zusammenhang zwischen CLX-Komponenten und Qt-Widgets. Die von der CLX selbst gezeichneten Elemente basieren entweder auf der Klasse *TCustomControl* oder auf *TGraphicControl* und überschreiben die jeweilige *Paint*-Methode. Den auf *TCustomControl* basierenden Komponenten steht auf der Seite der Qt-Bibliothek zumindest ein *QWidget* gegenüber, für die *TGraphicControl*-Komponenten gibt es überhaupt kein *QWidget* (abgesehen vom *QWidget* einer Elternkomponente, auf deren Fläche sich die *TGraphicControl*-Komponente selbst zeichnet).

Wenn Sie Qt kennen, können Sie aus der Tabelle schließen, was Sie von den einzelnen CLX-Komponenten erwarten können und wie sich die Benennung dieser Komponenten von Ihren gewohnten Qt-Klassen unterscheidet. Wenn Sie mit einer Komponente etwas vorhaben, was nicht in der Online-Hilfe dokumentiert ist, zeigt Ihnen diese abelle evtl., an welche Qt-Klasse Sie sich durch direkte Qt-Funktionsaufrufe wenden önnen.

Auf welchen Widgets eine Komponente basiert, lässt sich im Falle der CLX aus ihrer *CreateWidget*-Methode schließen (die evtl. auch unverändert von der Basisklasse übernommen wird). Dort finden Sie einen Aufruf der *Create*-Funktion aus *qtintf.so*, z. B. welche ihrerseits den C++-Konstruktor für die gewünschte Klasse aufruft. Der Name der *Create*-Methode gibt Aufschluss über die verwendete Qt-Klasse:

```
procedure TCustomGroupBox.CreateWidget; // aus QStdCtrls.pas
begin
  FHandle := QGroupBox_create(ParentWidget, nil);
end;
```

Wenn in der Tabelle bei einem Qt-Widget der Zusatz *»modifiziert«* angegeben ist, dann verwendet die CLX nicht direkt eine Klasse von Qt, sondern eine durch die *libqtintf.so*-Bibliothek bereitgestellte, von der Qt-Klasse abgeleitete Klasse, etwa *QClxLineEdit* statt *QLineEdit*.

CLX-Komponente	Zugrunde liegendes Widget	CLX-gezeichnet
Unit QControls		
TGraphicControl	-	X
TWidgetControl	QWidget	
TCustomContro	QWidget	X
TFrameControl	QFrame	
Unit QStdCtrls		
T(Custom)GroupBox	QGroupBox	
T(Custom)Label	QLabel	

CLX-Komponente	Zugrunde liegendes Widget	CLX-gezeichnet
T(Custom)Edit	QLineEdit (modifiziert)	
TButton	QPushButton	
T(Custom)ComboBox	QOpenComboBox	
T(Custom)CheckBox	QCheckBox	
T(Custom)ListBox	QListBox	
T(Custom)Memo	QOpenMultiLineEdit	
TRadioButton	QRadioButton	
TScrollBar	QScrollBar	
T(Custom)LCDNumber	QLCDNumber	
Unit QExtCtrls		
T(Custom)Panel	QFrame	X
T(Custom)ControlBar	QWidget	X
T(Custom)RadioGroup	QButtonGroup	
TBevel	-	X
TShape	-	X
TImage	-	X
TSplitter	-	X
Unit QComCtrls		
TCustomViewControl T(Custom)TreeView T(Custom)ListView	QListView	
TTrackBar	QSlider (modifiziert)	
T(Custom)HeaderControl	QHeader	
TSpinEdit	QSpinBox (modifiziert)	
T(Custom)TextViewer	QTextView	
T(Custom)TextBrowser	QTextBrowser	
T(Custom)IconView	QIconView	
T(Custom)TabControl	QWidget	X
TStatusBar	QWidget	X
TProgressBar	-	X
TToolBar	QWidget	X
Unit QForms		
TScrollingWidget T(Custom)Form	QOpenScrollView	

X Window und Qt unter Kylix ansprechen

Schon unter Windows war die VCL dafür bekannt, dass sie zwar das komplizierte Windows-API hervorragend kapselte, aber dem Entwickler trotzdem die Freiheit ließ, direkt auf dieses Windows-API zuzugreifen und so alle Funktionen von Windows nutzen zu können. Diesem Prinzip bleibt Kylix auch unter Linux treu: Auch hier können Sie direkt auf die darunter liegenden Bibliotheken zugreifen. Die X Window-Funktionen sind normale C-Funktionen und können wie Object Pascal Prozeduren/Funktionen aufgerufen werden. Ihre Deklarationen stehen zu diesem Zweck in der Unit *Xlib* zur Verfügung. Diese Möglichkeit ist jedoch eher theoretischer Natur, denn ein direkter Zugriff auf X Window würde die Plattformunabhängigkeit zunichte machen und außerdem selten einen funktionellen Vorteil bringen – die Funktionen von X Window werden durch Qt sehr gut abgedeckt und sogar stark erweitert. Sollte also einmal eine gewünschte Funktion in der CLX fehlen, ist es wahrscheinlicher, dass Sie schon bei der Qt fündig werden, und nicht erst in der X Window-Bibliothek. Die Verwendung der Qt ist in Kapitel 5.4 beschrieben, hier in Kapitel 3 konzentrieren wir uns nun auf die CLX.

3.1 Überblick über die CLX

Die CLX besteht aus einer Vielzahl von Units, von denen einige die Grundstruktur und eine Reihe abstrakter Klassen bereitstellen und einige andere sich auf die Implementie- ung von speziellen Komponenten konzentrieren. Wichtiger als die Aufteilung in Units ist die Klassenhierarchie, die sich durch die Zusammenführung aller Units ergibt.

Die Grundlage der Hierarchie sind abstrakte Klassen wie *TObject* und *TComponent*. Von diesen werden Sie wohl nie eine Instanz direkt herstellen, aber die meisten CLX-Klassen erben von ihnen wichtige Eigenschaften. Jede der abstrakten Klassen beschreibt ein Konzept der CLX auf kompakte Weise. Wollen Sie sich beispielsweise über das Wesen der Komponenten informieren, so gibt Ihnen die Klasse *TComponent* eine genaue Definition, durch welche Eigenschaften sich eine Komponente auszeichnet und welche ähigkeiten alle Komponenten haben. Interessieren Sie sich als weiteres Beispiel für die Gemeinsamkeiten von Editierfeldern und Memos, so finden Sie diese in der abstrakten Klasse *TCustomEdit*.

Abbildung 3.1 zeigt die wichtigsten Klassen der CLX-Hierarchie und die Vererbungslinie, der die nächsten Kapitel folgen werden. Die Klassen in dieser Linie sind die wichtigsten, da sie durch die Vererbung zum Wesen der Klasse *TForm* und *TWidgetControl* beitragen, wobei *TWidgetControl* die Gemeinsamkeiten aller interaktiven Steuerelementkomponenten beschreibt und *TForm* mit dem Formular das visuelle Fundament einer Kylix-Applikation legt (das nicht-visuelle Fundament ist das Objekt *Application*, ein Objekt der Klasse *TApplication*, siehe hierzu Kapitel 3.2.3).

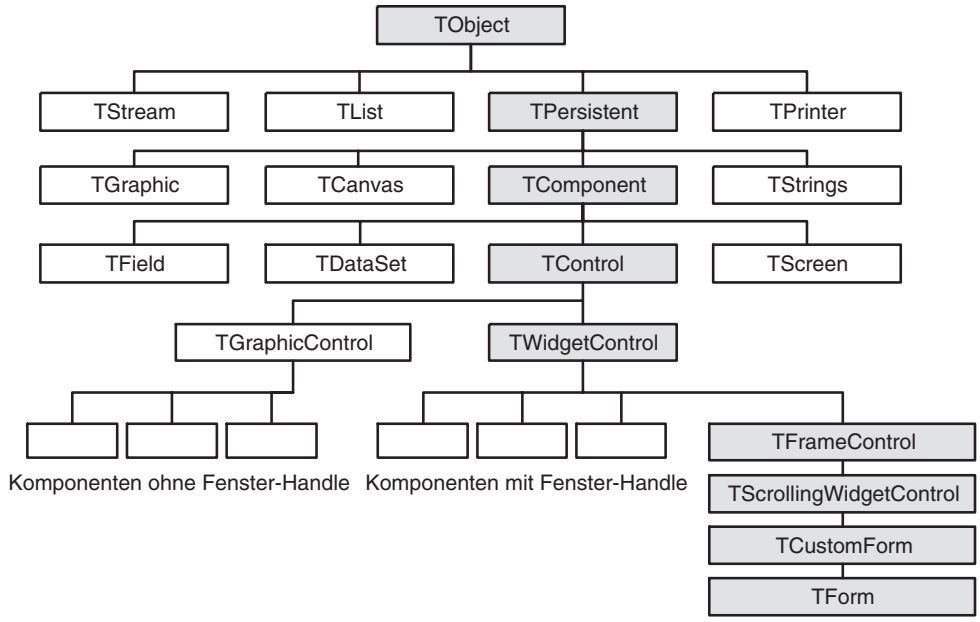


Abbildung 3.2: Das Gerüst der CLX-Klassenhierarchie von Kylix (detaillierte Grafiken siehe Anhang B)

3.1.1 Die grundlegenden Klassen

Wir beginnen in diesem Abschnitt mit grundlegenden Klassen, die im Hierarchiediagramm als oberste Klassen angesiedelt sind.

TObject

TObject ist die Wurzel des Hierarchiebaums. Von dieser Klasse wird jede neue Klasse automatisch abgeleitet, bei der nicht explizit eine andere Basisklasse angeben ist. *TObject* ist so grundlegend, dass sie zum innersten Teil der Laufzeitbibliothek, der Unit *System* gehört. *TObject* hat große Verdienste auf dem Gebiet der Polymorphie: Da selbst die verschiedensten Klassen einen gemeinsamen Kern haben, nämlich die Elemente, die in *TObject* deklariert sind, können Methoden und Funktionen geschrieben werden, die mit allen nur denkbaren Objekten umgehen können. Das häufigste Beispiel dafür sind sicher die Ereignisbearbeitungsmethoden, deren Parameter *Sender* vom Typ *TObject* ist.

Die wohl wichtigsten Methoden von *TObject* sind der Konstruktor *Create* zum Initialisieren des Objekts und die Prozedur *Free*, die das Objekt per Destruktoraufruf freigibt (Allgemeines zur Initialisierung und Freigabe von Objekten finden Sie in Kapitel 2.3). In der CLX tritt der *Create*-Konstruktor von *TObject* häufig in den Hintergrund, da die abgeleiteten Klassen ihn mit eigenen, zum Teil virtuellen Konstruktoren überschreiben, die allerdings intern auch *TObject.Create* aufrufen.

Hinzu kommen einige grundlegende Methoden zur Laufzeit-Typinformation, die in Kapitel 2.3.6 beschrieben sind (*ClassName*, *ClassType* etc.) und die im alltäglichen Einsatz sehr nützlich sein können. Von den weiteren *TObject*-Methoden sollen hier noch drei grundlegende Methoden für die Speicherverwaltung erwähnt werden, die bei der alltäglichen Programmierung weniger gebraucht werden, aber dennoch interessant sind, da sie im Hintergrund wichtige Dienste verrichten:

virtuelle Methode	**Aufgabe**
destructor Destroy	übernimmt alle Aufgaben eines Destruktors und wird häufig überschrieben, jedoch normalerweise nicht direkt aufgerufen (statt *Destroy* sollten Sie *Free* aufrufen).
function NewInstance: TObject	reserviert die von *InstanceSize* zurückgegebene Speichermenge und gibt einen Zeiger darauf zurück. Durch Überschreiben dieser Methode kann man tief in die Speicherverwaltung einer Klasse eingreifen, so etwa wie in C++ durch Überschreiben des *new*-Operators. Überschreiben wäre z. B. dann sinnvoll, wenn viele kleine Objektinstanzen sich einen großen Speicherblock teilen sollen, so dass nicht für jedes Objekt eine eigene Speicherreservierung vorgenommen werden muss.
ocedure FreeInstance	gibt den von *NewInstance* reservierten Speicher frei.

Object-Ableitungen

Da *TObject* die voreingestellte Basisklasse für alle Klassen ist, befinden sich unter ihren Ableitungen viele Klassen, die ansonsten sehr wenig miteinander zu tun haben wie B. *TPrinter* und *TList*. Ohne *TObject* wären sie völlig eigenständige Klassen ohne Verwandtschaft. Allerdings sind viele dieser Klassen nicht immer ansprechbar – manche esidieren in speziellen Units, die Sie extra einbinden müssen, z. B. die Klasse *TClipboard* in der Unit *QClipbrd*. (Die Units, in denen sich die visuellen Komponenten befinden, werden von Kylix automatisch zur *uses*-Anweisung hinzugefügt, wenn Sie die entsprechende Komponente in ein Formular einfügen.)

Manche *TObject*-Ableitungen verfügen über einige wenige Nachkommen, wobei besonders die *TFiler*- und *TStream*-Klassen (siehe Kapitel 4.3 und 4.3.4) wichtig sind. Alles in allem enden die meisten Vererbungslinien von *TObject* bereits nach spätestens zwei Schritten und sind damit leicht überschaubar.

Zwei *TObject*-Ableitungen bilden eine Ausnahme von dieser Regel: Die Klassen *Exception* und *TPersistent* sind ihrerseits die Wurzel eines sehr großen Ableitungsbaums. Da die Exceptions neben ihrer Klasse kaum weitere wichtige Informationen enthalten (siehe Kapitel 2.6.2), braucht diese Hierarchie nicht genauer untersucht zu werden.

Anders verhält es sich mit der *TPersistent*-Hierarchie: Fast alle häufig verwendeten Klassen sind von *TPersistent* abgeleitet, darunter auch alle Komponenten.

Die abstrakte Klasse TPersistent

Die Klasse *TObject* ist so allgemein gehalten, dass sie noch nicht einmal die Unterstützung von Dateioperationen beinhaltet. Diese kommt erst in der Klasse *TPersistent* dazu, allerdings ist sie dort nicht für allgemeine, sondern nur für interne Nutzung in der CLX selbst (beim Speicher- und Ladeprozess von Formularen) gedacht. Die Persistenz, die der Name andeutet, bezieht sich auf die Lebenszeit der *TPersistent*-Objekte: Sie kann durch das Speichern in einer Datei länger dauern als die Programmlaufzeit.

Natürlich können Sie auch Klassen, die nicht von *TPersistent* abstammen, in Dateien speichern; das besondere der *TPersistent*-Klassen ist, dass sie im Zusammenhang mit den *TFiler*- und *TStream*-Klassen zu polymorphen Dateioperationen fähig sind, was eben gerade beim Speichern eines Formularentwurfs wichtig ist.

Assign und DefineProperties

Die Klasse *TPersistent* enthält drei neue virtuelle Methoden, von denen nur eine dazu gedacht ist, direkt von Ihnen aufgerufen zu werden: Mit *Assign* weisen Sie einem Objekt ein anderes Objekt zu. Während Sie mit

```
Objekt1 := Objekt2;
```

lediglich zwei Variablen erhalten, die auf dasselbe Objekt weisen, kopiert *Assign* den *Inhalt* eines Objekts in ein anderes:

```
Objekt1.Assign(Objekt2);
```

Allerdings muss *Objekt1* vorher per Konstruktoraufruf initialisiert worden sein, da *Assign* kein neues Objekt *Objekt1* erstellt, sondern den Inhalt des bestehenden *Objekt1* ersetzt. Die genaue Funktionsweise von *Assign* hängt von den abgeleiteten Klassen ab. So können Sie beispielsweise der Methode *TClipboard.Assign* Grafikobjekte der Klassen *TBitmap*, *TIcon* und *TMetafile* übergeben, die diese dann in die Zwischenablage kopiert.

Die zweite Methode, *AssignTo*, wird intern von *Assign* verwendet, so dass nur noch die dritte Methode bleibt: *DefineProperties* wird normalerweise nur von der CLX aufgerufen und kann bei der Programmierung eigener Komponenten überschrieben werden (siehe Kapitel 8.4.2 und 8.6.6).

TPersistent-Ableitungen

Zu den Erben von *TPersistent* gehören neben der abstrakten Stringliste *TStrings* und deren Nachkommen einige Klassen, die mit Grafik zu tun haben, wobei in diesem Buch die folgenden Klassen behandelt werden:

- *TGraphicsObject* als Klasse für Zeichenwerkzeuge, mit denen Sie auf Zeichenflächen zeichnen können (siehe Kapitel 4.4)

- *TCanvas*, die allgemeine Zeichenfläche, die sowohl in den Fenstern des Bildschirms als auch in fensterunabhängigen Bitmaps (*TBitmap*) zu finden ist (ebenfalls Kapitel 4.4)
- *TGraphic* als allgemeine Klasse für die drei Grafiktypen Icon, Bitmap und Metadatei, denen das Kapitel 4.5 gewidmet ist
- die Klasse *TPicture*, die zur leichteren Verwaltung von *TGraphic* gedacht ist (ebenfalls Kapitel 4.5) und
- darüber hinaus alle Komponenten (Basisklasse *TComponent*)
- *TStrings*, die Basisklasse für Stringlisten (beschrieben in Kapitel 4.1.1)
- *TClipboard*, die Klasse für die Zwischenablage (behandelt in Kapitel 4.7.2).

3.1.2 Komponenten

Nachdem wir auch für die meisten *TPersistent*-Unterklassen bereits ein Kapitel gefunden haben, das sich diesen vollständig widmen kann, bleibt wieder eine Klasse übrig, die so viele Nachkommen hat, dass ein weiterer Überblick fällig ist. Wir folgen also der in Abbildung 3.1 hervorgehobenen Linie und setzen den Überblick mit der Klasse *TComponent* fort.

Die Klasse TComponent

Zu den grundlegenden Eigenschaften dieser Klasse gehört vor allem die Art, wie mehrere Komponenten miteinander verknüpft werden. Diese wird in Kapitel 3.2 beschrieben.

TComponent ist die erste Klasse in der Hierarchie, die Properties veröffentlicht. Die folgenden zwei Properties sind somit für alle Komponenten das Minimum der im Objektinspektor editierbaren Eigenschaften:

- In *Name* befindet sich der Name, unter dem die Komponente im Object-Pascal-Code angesprochen werden kann. Sie können ihn auch zur Laufzeit ändern, wobei dann dieselben Regeln gelten wie zur Entwurfszeit: Sie dürfen einer Komponente keinen Namen geben, den bereits eine andere Komponente besitzt, sonst verursachen Sie eine Exception.
- Das Property *Tag* ist ein Allzweck-Property, in dem Sie nach Belieben einen 32 Bit großen Wert ablegen können. Ein häufiger Verwendungszweck ist die Unterscheidung zwischen verschiedenen Komponenten in einer einzigen Ereignisbearbeitungsmethode (siehe beispielsweise Kapitel 3.5.2).

Die folgende Übersicht zeigt weitere Elemente, auf die Sie nur zur Laufzeit zugreifen können; die meisten davon betreffen die Besitzverhältnisse, die in Kapitel 3.2.1 besprochen werden:

Element	Bedeutung
Components (Property) ComponentCount(Property) ComponentIndex(Property) Owner	Properties, die die Besitzverhältnisse der Komponenten definieren (Kapitel 3.2.1)
ComponentState (Property)	Menge von Flags, die über einige Vorgänge Bescheid geben, in die die Komponente gerade eingebunden ist. Neben vielen anderen gehören dazu als besonders wichtige Flags: *csLoading* (die Komponente wird aus einer Formulardatei geladen), *csDestroying* (wird unmittelbar vor dem Ablauf des Destruktors gesetzt), *csDesigning* (die Komponente ist Teil eines Formulars, dass gerade in der Kylix-IDE bearbeitet wird).
FindComponent, InsertComponet, RemoveComponent (Methoden)	Methoden, die mit der Komponentenliste *Components* arbeiten (Kapitel 3.2.1)

Verschiedene Arten von Komponenten

Um es gleich vorwegzunehmen: Auch auf dieser Ebene der Klassenhierarchie gibt es wieder eine Klasse, die fast alle weiteren Nachkommen an sich zieht: *TControl*. Neben *TControl* gibt es eine noch relativ große Gruppe von Klassen, die von *TCommonDialog* ausgeht, und einige kleinere Gruppen oder einzelne Klassen wie *TTimer*, *TScreen*, *TMenuItem* und *TMenu*, die alle in einem eigenen Kapitel dieses Buchs behandelt werden. Auch viele wichtige nicht-visuelle Datenbankklassen sind Komponenten (*TField*, *TDataSet*, *TDataSource* etc.).

Alle *TComponent*-Typen außer *TControl* sind »nicht-visuell«, d.h. sie werden zur Entwurfszeit nur als Icon dargestellt und können dort nur verschoben, aber nicht anderweitig manipuliert werden. Das heißt jedoch nicht, dass von diesen Klassen zur Laufzeit des Programms nichts zu sehen ist, wie einige Beispiele zeigen sollen:

- *TMenuItem* und *TMenu* bezeichnen zwar Objekte, die zur Laufzeit des Programms sichtbar sind (Menüpunkte und ganze Menüs), sie werden jedoch nicht innerhalb des Formulars, sondern in einem eigenen Editor editiert.
- *TTimer* ist ein Beispiel für eine vollständig nicht-visuelle Komponente, sie ist also zur Progammlaufzeit nicht sichtbar und wird beim Formularentwurf durch ein Icon repräsentiert.

- *TScreen* schließlich gehört zu den Komponenten, die beim Oberflächenentwurf in der IDE überhaupt nicht zugänglich sind. Sie repräsentiert den Bildschirm, dessen Sichtbarkeit sich jedoch normalerweise nicht bestreiten lässt. Mehr zu *TScreen* finden Sie in Kapitel 3.2.4.

3.1.3 Visuelle Komponenten

Es bleibt nun die größte Untergruppe von *TComponent*, der Hierarchiebaum von *TControl*. Die Eigenschaft, in der sich *TControl* von den anderen direkten *TComponent*-Nachfahren unterscheidet, ist, dass *TControl*-Elemente zur Entwurfszeit innerhalb des Formulars manipuliert werden können.

Wie Sie aus Abbildung 3.1 erkennen können, sind die Dialogelement-Komponenten noch nicht direkt von *TControl* abgeleitet, *TControl* dient lediglich als Basisklasse für zwei verschiedene Typen von Dialogelementen: *TWidgetControl* und *TGraphicControl*. Der Kompatibilität mit Windows-Delphi wegen können Sie *TWidgedControl* in Kylix auch unter dem Alias-Namen *TWinControl* ansprechen.

Die Unterschiede zwischen TWidgetControl und TGraphicControl

Der grundlegende Unterschied zwischen diesen beiden Klassen ist, dass eine *TWidgetControl*-Instanz mit einem Qt-Widget verbunden ist (sozusagen ein von Qt verwalteter Fensterbereich), während *TGraphicControl*-Instanzen von der CLX alleine verwaltet werden. Während Sie *TGraphicControl*-Komponenten in Ihrem Object-Pascal-Code mit bestimmten Einschränkungen wie ein *TWidgetControl* ansprechen können, ist eine *TGraphicControl*-Komponente aus der Sicht von Qt nur ein nicht abgegrenzter, beliebiger Bereich innerhalb eines »richtigen« Widgets.

Dies hat einige weitere Folgen:

- *TWidgetControl*-Elemente können den Tastatureingabefokus erhalten, Tastatureingaben in *TGraphicControl*-Elemente sind nur auf Umwegen möglich.
- *TWidgetControl*-Elemente können andere Komponenten enthalten und damit zum Eltern-Fenster werden.
- Andererseits benötigen *TGraphicControl*-Elemente unbedingt ein »Wirt-Fenster« (ein *TWidgetControl*), durch das sie erst sichtbar werden.

Das Handle eines WidgetControls

Normalerweise brauchen Sie sich nicht weiter um das *Handle* eines *WidgetControls* zu ümmern, wenn Sie aber Funktionen der Qt-Bibliothek direkt aufrufen wollen, die ein *QWidgetH (*allgemein *QxxxH*) als Parameter erwarten, benötigen Sie auch eine Komponente, die ein solches Handle besitzt, und das sind eben die *TWidgetControl*-Ableitun-

gen. Das Handle liegt günstigerweise in Form eines Properties vor und ist damit immer gültig. Wenn Sie also versuchen, *Handle* auszulesen, wenn das Widget noch nicht existiert, kümmert sich die CLX zuerst um die Erstellung des Widgets, bevor sie den gültigen Wert von *Handle* zurückgibt. (Diese Versprechung sollten Sie allerdings nicht allzu sehr auf die Probe stellen, da es durchaus Zeitpunkte im Auf- und Abbauprozess von Formularen gibt, in denen der Versuch, ein Widget anzusprechen, zu einer Schutzverletzung führen kann – was vielleicht teilweise auch am noch relativ jungen Entwicklungsstand der CLX liegt).

Grafikausgabe in TGraphicControl

Anstelle des Handles hat die Klasse *TGraphicControl* einen anderen wichtigen Vorteil: Sie verfügt über ein *Canvas*-Property, das die Ausgabe von Grafik im Fensterbereich, den die Komponente belegt, stark vereinfacht (siehe Kapitel 4.4). Dazu gehört die Methode *Paint*, die Sie in eigenen abgeleiteten Komponenten überschreiben können, und das Ereignis *OnPaint*, das es Ihnen ermöglicht, die Zeichenfunktionalität vom Objektinspektor aus in einer Formularmethode unterzubringen.

Die Klassen, die von *TGraphicControl* abgeleitet sind, nutzen diese Eigenschaften bereits und geben damit gute Beispiele für eigene *TGraphicControl*-Ableitungen. Sie sind von sich aus nicht interaktiv, können aber durch Bearbeitung der entsprechenden Ereignisse so gemacht werden.

Grafikausgabe in TWidgetControl

Der grundsätzliche Unterschied zwischen den beiden Control-Typen bedeutet jedoch nicht, dass Sie nicht auch in *TWidgetControl*-Elementen Grafik ausgeben können. Wenn Sie eine eigene Steuerelementkomponente erstellen wollen, die auch über ein Fensterhandle verfügt, sollten Sie sie von *TCustomControl* ableiten, einer Spezialisierung von *TWidgetControl*, die wie *TGraphicControl* über ein *Canvas*-Property verfügt. Für die Grafikausgabe vom Formular aus sollten Sie die Komponente *TPaintBox* verwenden, die von *TGraphicControl* abgeleitet ist.

3.1.4 Der Nachrichtenfluss

Ereignisbearbeitung sieht ganz einfach aus, wenn Sie im Formular-Designer eine Komponente über den Objektinspektor mit einer Ereignisbearbeitungsmethode verknüpfen. Normalerweise müssen Sie sich nicht darum kümmern, wie diese Methode aufgerufen wird, denn dafür sorgt die CLX automatisch. Dieses Kapitel wirft einen kleinen Blick hinter die Kulissen und erläutert, welche Stufen ein Ereignis durchläuft, bevor die von Ihnen geschriebene Methode aufgerufen wird. Das Verständnis dieser Vorgänge kann Ihnen nützlich sein,

- wenn Sie Ereignisse bearbeiten wollen, die in der CLX nicht berücksichtigt wurden,
- wenn Sie Ereignisse bearbeiten wollen, die nicht an bestimmte Komponenten gebunden sind, sondern global für die gesamte Anwendung gelten,
- wenn Sie eigene Komponenten schreiben und Ereignisse auf eine andere Art verarbeiten müssen, als Sie es beim Entwurf eines Formulars gewohnt sind.

Abbildung 3.2 zeigt die Verarbeitung der Nachrichten in einer Kylix-Anwendung unter Linux (bei Windows-Anwendungen, die statt der CLX die VCL verwenden, läuft diese Verarbeitung völlig anders ab!).

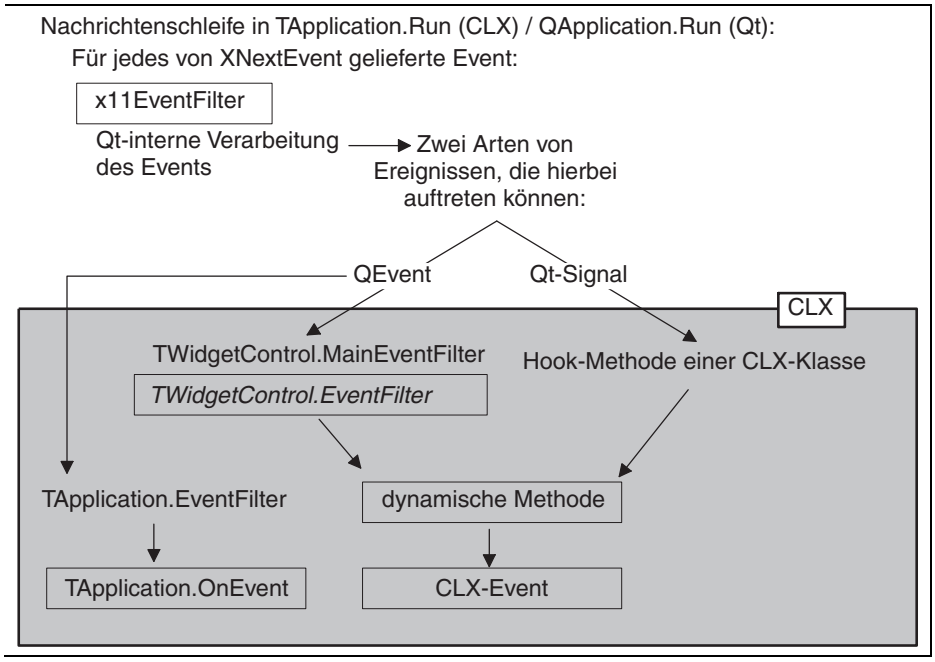


Abbildung 3.3: Die Wege einer Nachricht in einer Kylix-Anwendung. An den eingerahmten Positionen (z.B. bei OnEvent) gibt Ihnen die CLX die Möglichkeit, in das Geschehen einzugreifen.

Die Nachrichtenschleife

Unabhängig vom verwendeten Betriebssystem besteht das »Hauptprogramm« einer ereignisgesteuerten Anwendungen meistens aus einer so genannten Nachrichtenschleife. Diese Schleife fragt so lange neue Nachrichten vom Betriebssystem ab (unter X Window z.B. durch Aufruf der Funktion *XNextEvent*), bis eine Nachricht vom Ende des Programms eintritt. Wenn ein neues Ereignis eintritt, der Benutzer also z.B. die Maus bewegt, generiert das Betriebssystem von sich aus eine Nachricht und schreibt sie in eine Nachrichtenwarteschlange[1]. Diese Warteschlange ist erforderlich, damit die

Anwendung ihren Verarbeitungsrhythmus selbst bestimmen kann. So können beispielsweise während der Bearbeitung einer Nachricht bereits neue Nachrichten eintreffen, die dann in der Schlange warten.

Auf den von Delphi unterstützten Systemen besteht eine Nachricht zumindest aus einer Kennzahl für den Typ des Ereignisses (z. B. Tastendruck oder Mausbewegung), aus einer Referenz auf das Widget oder Fenster, für welches das Ereignis bestimmt ist, sowie weiteren ereignisspezifischen Informationen (bei einer Mausbewegung z. B. die Position der Maus).

- *Von der Qt zur CLX*

Wenn eine Anwendung eine Nachricht vom Betriebssystem erhält, muss sie zunächst den Ereignistyp prüfen, um dann über die weitere Bearbeitung entscheiden zu können. In objektorientierten Systemen ist es außerdem wichtig, dass die Nachricht von der globalen Nachrichtenschleife an eines von eventuell mehreren Fensterobjekten weitergeleitet wird (die Fensterinformation, die das Betriebssystem als Teil der Nachricht sendet, gibt kein Objekt an, sondern nur eine Fenster-ID).

In einer CLX-Anwendung ist die Qt-Bibliothek für diese erste Weiterleitung zuständig. In der Nachrichtenschleife ermittelt *QApplication* das zu einer Fenster-ID des Betriebssystems gehörige *QWidget*-Objekt und ruft dessen *event*-Methode auf, von der aus nach Bedarf in weitere Qt-interne Methoden verzweigt wird.

In diese Qt-interne Bearbeitung klinkt sich nun die CLX per »Hook« ein; für den Aufbau der Verbindung definiert *TWidgetControl* die virtuelle Methode *HookEvents*, die von den meisten CLX-Komponenten überschrieben wird, um spezielle Hooks zu verwenden. Die Hooks setzen an zweierlei von der Qt vordefinierten Stellen an:

- Qt erlaubt die Definition von Filter-Methoden, die über alle Ereignisse für ein bestimmtes Widget informiert werden, bevor diese Ereignisse weiter bearbeitet werden. Diese allgemeinen Ereignisse werden durch eine *QEvent*-Struktur beschrieben. Die Basisklasse aller Widget-Komponenten, *TWidgetControl*, definiert eine Filter-Methode mit dem Namen *MainEventFilter*. Die eigentliche Verknüpfung gestaltet sich komplizierter und läuft über Borlands Bibliothek *qtintf.so* ab, genauer gesagt über deren Prozedur *Qt_hook_hook_events*, die Sie im Quelltext der CLX in vielen *HookEvents*-Methoden aufgerufen finden.
- Ein weiteres Grundprinzip der Nachrichtenbearbeitung von Qt sind die *Signals* und *Slots*. Wir können die Signals hier einfach als Ereignisse betrachten, was sie prinzipiell auch sind. Der spezielle Charakter dieser Signals (das Signal/Slot-Prinzip) ist ohnehin nur zu erkennen, wenn man Qt von C++ aus verwendet, denn da

1. Unter X11 ist das nicht unbedingt für jedes Ereignis der Fall, da man beim Programmstart uninteressante Nachrichten von vornherein ausfiltern kann bzw. umgekehrt zuerst die Nachrichten angeben muss, die überhaupt gesendet werden sollen.

das Signal/Slot-Prinzip nicht direkt von C++ auf Object Pascal übertragen werden kann, wurde es von Borland in ein »Hook-Prinzip« übersetzt. In den bereits erwähnten *HookEvents*-Methoden verbinden einige CLX-Komponenten Methoden mit speziellen Qt-Signals (siehe z.B. *QStdCtrls*, Methode *TCustomListBox.HookEvents*).

Während die *QEvent*-Ereignisse des ersten Punkts allgemeiner Natur sind, sind die *Signals* speziell und je nach Art des Widgets sehr verschieden. Ein Beispiel: Es gibt ein allgemeines *Paint*-Ereignis, das als *QEvent* an jedes Widget gesendet wird, das sich neu zeichnen soll. Die CLX sieht dieses Ereignis in ihren *MainEventHandler*-Methoden. Erhält nun eine besitzergezeichnete Listbox eine solche *Paint*-Aufforderung, erzeugt sie »kleinere« Paint-Ereignisse für jeden einzelnen Listeneintrag. Diese Ereignisse sind nun nicht mehr als *QEvent*, sondern als *Signal* definiert. Die CLX fängt sie in einer eigenen Methode *TCustomListBox.ItemPaintHook* ab (die Verknüpfung des Hooks geschieht in der schon erwähnten Methode *TCustomLixBox.HookEvents*) und erzeugt darin das *OnDrawItem*-Event, das Sie über den Objektinspektor in einer eigenen Methode verknüpfen können.

on der CLX zur Formularmethode

Entsprechend den beiden beschriebenen Ansatzpunkten der Ereignis-Hooks gelangen Qt-Ereignisse also auf zwei Wegen in die CLX: Alle *QEvent*-Ereignisse gehen an die CLX-Methode *TWidgetControl.MainEventFilter*, die *Signals* gehen an spezielle CLX-Methoden.

TWidgetControl.MainEventFilter ist eine statische Methode, die nur wenige ganz grundlegende Aufgaben verrichtet wie einen Exception-Behandlungsrahmen um die gesamte Ereignisverarbeitung zu legen und zur Entwurfszeit einen Ansatzpunkt (auch wieder *Hook* genannt) für den Formulardesigner der Delphi-IDE zu bieten. Die eigentliche Ereignisbearbeitung findet in der Methode *EventFilter* statt, die von *MainEventFilter* aufgerufen wird. *EventFilter* ist virtuell, kann also von den von *TWidgetControl* abgeleiteten Klassen überschrieben werden, so etwa von der Methode *TForm.EventFilter*. Wie es sich für eine virtuelle Methode meistens anbietet, ruft *TForm.EventFilter* neben den eigenen Operationen auch die von der Elternklasse geerbte *EventFilter*-Methode auf, welche wiederum ihre Elternklasse zum Zug kommen lässt, so dass schließlich auch *TWidgetControl.EventFilter* ausgeführt wird. In dieser führt *TWidgetControl* ganz zentrale CLX-Aufgaben durch wie etwa Nachrichten an *TGraphicControl*-Komponenten weiterzuleiten, die im *TWidgetControl* als Kindelemente enthalten sind *TGraphicControl*-Komponenten sind nicht mit Qt-Widgets verbunden, daher kann Qt die Ereignisse nicht direkt an sie weiterleiten).

Abbildung 3.2 zeigt, wie der Weg der Ereignisse über die *EventFilter*-Methoden letztlich in denselben Alternativen mündet wie der Weg von den Signal-Hook-Methoden:

- In beiden Fällen werden häufig dynamische Methoden (eine besondere Form virtueller Methoden) aufgerufen, die für die Bearbeitung eines ganz speziellen Ereignisses gedacht sind. Dieser Schritt ist aus folgenden Gründen sinnvoll: *EventFilter* wird ja noch für *alle* Ereignisse aufgerufen und kann daher schon aus Gründen der Übersichtlichkeit nicht die komplette Behandlung selbst übernehmen. Die Signal-Hook-Methoden dagegen, die zwar schon auf ein Ereignis spezialisiert sind, sind aber statische Methoden und können als solche nicht überschrieben werden. Erst der Aufruf der dynamischen Methoden gestattet es abgeleiteten Klassen, ihre eigene Logik mit einzubringen. Beispiele für dynamische Methoden sind *MouseDown*, *KeyPress*, *Resize*, *DblClk*.
- Dynamische Methoden sind hauptsächlich für Komponentenprogrammierer interessant. Doch auch der Formularentwickler möchte von zahlreichen Ereignissen der Komponenten unterrichtet werden. Daher gibt es schließlich noch die CLX-Events, und nach der Begriffsvereinbarung dieses Buchs bezieht sich dies ja auf die Art von Ereignissen, die Sie im Objektinspektor aufgelistet finden. CLX-Events können zwar theoretisch überall entstehen, werden aber normalerweise erst von den dynamischen Methoden generiert, daher ist in Abbildung 3.2 nur dieser Ablauf eingezeichnet. Während die dynamische Methode noch Teil der Komponente ist, gehört die Methode, die durch das Event aufgerufen wird, normalerweise zum Formular (falls das Event wie üblich über den Objektinspektor mit einer Formularmethode verknüpft wurde). Wie dieser letzte Schritt genau abläuft, ist wieder für Komponentenprogrammierer wichtig und wird in Kapitel 8.3.2 erläutert.

Anwendung

In den beschriebenen Ablauf des Nachrichtenflusses können Sie auf folgende Weise eingreifen:

- Die speziellen Ereignismethoden der Komponenten (wie etwa *MouseMove*) sind virtuell und können daher leicht in eigenen Komponenten überschrieben werden (siehe hierzu Kapitel 8.4.1).
- *QEvent*-Ereignisse, die an eine bestimmte Komponente gesendet werden, können Sie durch eine überschriebene *EventFilter*-Methode abfangen, wozu Sie allerdings eine neue Komponentenklasse ableiten müssen (auch hierzu finden Sie in Kapitel 8.4.1 ein Beispiel).
- *QEvent*-Ereignisse, die an das Formular-Widget gesendet werden, können natürlich auch in einer *EventFilter*-Methode abgefangen werden, nur ist es leichter, diese im Formular zu definieren, da Sie in der Formular-Unit sowieso schon eine abgeleitete Formularklasse vorliegen haben.

- Alle in einer Anwendung auftretenden *QEvent*-Ereignisse, egal für welche Komponente, können Sie abfangen, indem Sie das Ereignis *TApplication.OnEvent* bearbeiten.
- Für den Fall, dass Sie selbst direkt ein *QEvent*-Ereignis erzeugen wollen, etwa um eine Benutzereingabe zu simulieren, sei auf R79 Seite 736 verwiesen.
- Während alle bisher genannten Eingriffsmöglichkeiten auch mit der CLX unter Windows verfügbar sein werden, gibt es noch eine spezielle Option, die nicht portabel ist: Sie können auch einen Event-Filter für Nachrichten auf X11-Ebene definieren. Dies sollte jedoch normalerweise nicht notwendig sein und dieses Buch wird sich nicht weiter damit befassen. Wenn Sie jedoch einen praktischen Eindruck von X11-Nachrichten erhalten oder ein Experiment damit machen wollen, so finden Sie unter dem Namen *MessageDemo* auf der CD ein Projekt, das einen X11-Event-Filter definiert, der die Namen der X11-Ereignisse auf der Konsole (oder in einer Log-Datei) ausgibt.

3.2 Die Beziehungen der Komponenten

Während das letzte Kapitel die Vererbungshierarchie untersuchte, beschäftigt sich dieses mit der Besitzhierarchie. Normalerweise besitzt ein Formular alle Komponenten, die von Kylix automatisch zu der Klassendeklaration des Formulars hinzugefügt werden, jedoch ist das Hinzufügen zur Klassendeklaration noch nicht ausschlaggebend dafür, ob eine Komponente im Besitz einer anderen ist oder nicht.

Die Unterschiede zwischen dieser Besitzhierarchie und der Fensterhierarchie sowie die Art, wie diese Beziehungen in der CLX dargestellt werden, sind Thema dieses Kapitels.

3.2.1 Besitzhierarchie und Komponentenlisten

Zwar können Sie die Komponenten, die sich im Besitz eines Formulars befinden, jederzeit direkt mit dem Namen ansprechen, in manchen Fällen ist es jedoch wünschenswert, alle »besessenen« Komponenten in einer Schleife behandeln zu können, z. B., um sie in einer Datei zu speichern, wie die CLX es beim Speichern des Formulars tun muss. Zu diesem Zweck enthält die Klasse *TComponent* eine Liste, die den gesamten Komponentenbesitz, der normalerweise schon in Formularvariablen zu finden ist, auf eine zweite Weise speichert.

Eine wichtige Bedeutung des Komponentenbesitzes liegt darin, dass die Besitzerkomponente normalerweise dafür verantwortlich ist, die ihr gehörenden Komponenten
eizugeben, wenn auch sie selbst freigegeben wird.

Properties von TComponent zur Besitzverwaltung

Die Organisation dieser Komponentenliste ist dabei unwichtig, da sie in Form eines indizierten Properties vorliegt. Damit jede Komponente nicht nur weiß, was sie besitzt, sondern auch, wem sie gehört, verfügt *TComponent* außerdem über das Property *Owner*. Diese und zwei weitere Properties sind in der folgenden Tabelle zusammengefasst – die Properties sind allesamt nur zur Laufzeit ansprechbar:

TComponent-Property	Typ	Bedeutung	Bemerkung
Components	array of TComponent	Liste der Komponenten, die der Komponente gehören	nur lesen
ComponentCount	Integer	Zahl der Komponenten in der Liste *Components*	nur lesen
ComponentIndex	Integer	Index der Komponente in der *Components*-Liste des Besitzers	lesen und schreiben
Owner	TComponent	weist auf den Besitzer der Komponente	nur lesen

Veränderung der Komponentenliste zur Laufzeit

Wenn Sie zur Laufzeit eine neue Komponente erzeugen, sorgen Sie für die richtige Verknüpfung zu seiner Besitzerkomponente, indem Sie diese als Parameter für den *Create*-Konstruktor der neuen Komponente angeben. Zu diesem Zweck erwarten alle Komponentenkonstruktoren einen Parameter *AOwner* des Typs *TComponent*, dem Sie auch den Wert *nil* übergeben können, falls die Komponente keinen Besitzer haben soll. Auf diese Weise können Sie Komponenten erzeugen, die völlig unabhängig von allen anderen Komponenten sind.

Der Komponentenkonstruktor setzt also die Properties *Owner* und *ComponentIndex* der eigenen Komponente und sorgt dafür, dass die Properties *Components* und *ComponentCount* der Besitzerkomponente angepasst werden. Obwohl die Properties *Components* und *Owner* nicht beschreibbar sind, können Sie die Komponenten-Besitzverhältnisse zur Laufzeit auch auf andere Weise beeinflussen:

- Wenn Sie eine Komponente nachträglich (also nach der Konstruktion mit *Create*) in den Besitz einer anderen Komponente einfügen wollen, rufen Sie beim neuen Besitzer die Methode *InsertComponent* auf, die auch vom *Create*-Konstruktor verwendet wird.
- Um eine Komponente aus dem Besitz einer anderen Komponente zu löschen, müssen Sie, statt *Owner* auf *nil* zu setzen, die Methode *RemoveComponent* des Besitzers aufrufen, z.B. *Owner.RemoveComponent(self)*. Dadurch wird *Owner* indirekt auf *nil*

gesetzt. Die *Components*-Liste funktioniert genauso wie *TList*, füllt also Lücken, die durch Entfernen von Elementen entstehen, sofort durch Nachschieben aller folgenden Elemente auf.

- Mit dem einzigen beschreibbaren Property aus der obigen Liste, *ComponentIndex*, können Sie schließlich die Position der Komponente in der Liste des Besitzers verändern.

Wenn Sie eine Komponente zur Laufzeit löschen, sorgt der Destruktor selbstverständlich automatisch dafür, dass sie aus der Liste des Besitzers gestrichen (mit *RemoveComponent*) und dass ihr eigener Komponentenbesitz gelöscht wird.

Herkunft der Components-Liste

Interessant ist auch, wie die CLX die Komponenten eines Formulars zur Laufzeit wiederherstellt. Dazu genügt es nämlich nicht, einfach alle Komponenten wiederherzustellen und dabei die *Components*-Listen aufzubauen. Wenn eine Komponente dem Formular gehört, so ist in dessen Klassendeklaration höchstwahrscheinlich eine Objektvariable enthalten, die ebenfalls auf diese Komponente weist. Auch diese Tatsache ist in der `xfm`- bzw. in der ausführbare Datei gespeichert. Wenn also eine solche Variable existiert, muss die CLX auch diese auf die neu erzeugte Komponente setzen. Enthält das Formular beispielsweise eine Komponente mit dem Namen *LogoImage*, setzt die CLX sowohl *LogoImage* selbst als auch einen Listeneintrag von *Components* auf diese Komponente (eine `xfm`-Datei enthält zu diesem Zweck den Namen der Variablen, die auf die neue Komponente weisen sollen. Um von diesem Namen auf die Adresse der Variablen schließen zu können, verwendet die CLX hier die Methode *TObject.FieldAddress*).

3.2.2 Die Fensterhierarchie

Bei einfachen Formularen ist es klar, dass die im letzten Abschnitt beschriebene Besitzhierarchie auch der Fensterhierarchie entspricht: Das Elternfenster ist also nicht nur »Vater und Mutter«, sondern auch Besitzer der Komponenten. In manchen Fällen ist es jedoch sinnvoll, diese Beziehungen etwas zu differenzieren. Ein sehr häufiges Beispiel dafür sind die Panels und Gruppen von Steuerelementen.

Haben Sie beispielsweise mehrere Radioschalter, die Sie zu einer Gruppe zusammenfassen wollen, so sollten Sie sie in einer *TPanel*-Komponente platzieren. Auf diese Weise wird das Panel zum Elternfenster der Radioschalter, obwohl das Formular deren Besitzer wird.

Properties zur Fensterhierarchie

Auch die untergeordneten Fenster müssen oft so effektiv über eine Liste angesprochen werden können wie die Komponenten der *Components*-Liste. So kann ein Fenster beispielsweise die Liste seiner Unterfenster nach einem bestimmten Komponententyp durchsuchen (um beispielsweise alle Aktionsschalter in einer Schleife zu deaktivieren).

Da jedoch die Klasse *TComponent* noch nichts mit Fenstern zu tun hat, enthält sie auch keine Elemente zur Speicherung der Fensterhierarchie. Die Klassen, die diese Elemente einführen, sind *TControl* und *TWidgetControl*. *TWidgetControl* verfügt über eine Liste aller enthaltenen Steuerelemente (Unterfenster) und *TControl* besitzt das entsprechende *Parent*-Feld:

Property	Typ	ab Klasse	Bedeutung	Bemerkung
Controls	array of TControl	TWidgetControl	Liste der untergeordneten *TControl*-Elemente	nur lesen
ControlCount	Integer	TWidgetControl	Zahl der Elemente in *Controls*	nur lesen
Parent	TWidgetControl	TControl	übergeordnetes Fensterelement eines *TControl*-Elements	beschreibbar

Dies ist ein weiterer Unterschied zur Besitzhierarchie: Nicht alle Komponenten, die Kinder sein können, können auch Eltern sein. Alle grafischen Steuerelemente, deren Klassen von *TGraphicControl* abgeleitet sind, können lediglich in einem anderen Fenster enthalten sein, aber keine Unterfenster mehr besitzen. Dies ist eine Fähigkeit der Vertreter des *TWidgetControl*-Teilbaums der Klassenhierarchie. Da *TWidgetControl* natürlich das *Parent*-Feld erbt, können *TWidgetControl*-Komponenten beliebig geschachtelt sein.

Veränderungen zur Laufzeit

Für die Komponenten, die Sie schon zur Entwurfszeit in das Formular einfügen, sorgt die CLX automatisch für den Aufbau der Eltern-Kind-Beziehungen. Wenn Sie eine Komponente zur Laufzeit selbst erzeugen, müssen Sie sie explizit mit einem Elternfenster verbinden, indem Sie ihr *Parent*-Property auf das Elternfenster setzen, sonst erhält sie überhaupt kein Elternfenster. Eine Ausnahme sind MDI-Kindformulare. Zu diesen sucht die CLX automatisch das MDI-Hauptfenster (von dem es nur ein einziges geben kann) aus den bestehenden Formularen als Elternfenster heraus – wenn Sie hier das *Parent*-Property selbst setzen, kommt es unter Umständen zu merkwürdigen kleinen Fehlern (siehe Kapitel 6.7.4). Bei Formularen, die kein übergeordnetes Formular haben, bleibt das *Parent*-Property *nil*.

Den Methoden *InsertComponent* und *RemoveComponent* entsprechen die *TControl*-Methoden *InsertControl* und *RemoveControl*. Anders als *Owner* ist *Parent* jedoch ein beschreibbares Feld, so dass Sie nicht die Methode *InsertControl* verwenden müssen, um diese Verbindung nachträglich herzustellen. Ebenso sparen Sie sich einen Aufruf der Methode *RemoveControl*, indem Sie *Parent* einfach auf *nil* setzen. Ein weiterer Unterschied zur Komponentenliste ist schließlich, dass Sie die Position eines Elements innerhalb von *Controls* nicht beliebig festlegen können, ein Property *ControlIndex* gibt es also nicht. Indirekt können Sie auf den Index im *Controls*-Array Einfluss nehmen, indem Sie eine Komponente mit *SendToBack* in den Hintergrund setzen oder mit *BringFront* in den Vordergrund holen (siehe *Z-Reihenfolge von Komponenten* in Kapitel 3.3.1).

In Kapitel 6.7.5 finden Sie ein Beispiel, bei dem das *Parent*-Property mehrfach zur Laufzeit geändert wird.

3.2.3 Die oberste Komponente: TApplication

Nachdem wir nun die Komponentenverknüpfungen innerhalb von Formularen genau durchleuchtet haben, bleibt die Frage offen, wer Besitzer und Elternfenster des Formulars sind. Während das *Parent*-Property eines Hauptfensterformulars *nil* bleibt, gibt es eine Komponente, die das Formular besitzt:

Sie wird in der Unit *Forms* deklariert und erzeugt, hat den Typ *TApplication* und heißt *Application*. Da die Unit *Forms* automatisch in jede Formular-Unit eingebunden wird, können Sie in Ereignisbearbeitungsmethoden ohne weitere Vorbereitung auf das *Application*-Objekt zugreifen. *TApplication* definiert viele Methoden und Properties, die globale Bedeutung haben und ohne OOP in globalen Funktionen oder globalen Variablen implementiert werden müssten. Zu den Aufgaben von *Application* gehören neben der Nachrichtenbearbeitung z. B. die Anzeige von Hinweisfenstern, der Aufruf der Hilfefunktion und Informationen über die unter der CLX liegenden Software-Schichten der Qt-Bibliothek und von X Window.

Wie andere Komponenten auch, benachrichtigt Sie *Application* über Events von seinen eigenen Aktionen und lässt Sie in gewissem Umfang Einfluss auf diese Aktionen nehmen, teilweise mit Properties, teilweise über die Events. Die Anzeigen der Hinweisfenster, die Sie auch in der Kylix-IDE erhalten (wenn Sie die Maus über die Schalter unter der Menüzeile halten), wird beispielsweise grob durch die Properties *ShowHint*, *HintColor* und *HintPause* gesteuert (wenn Sie das Event *OnShowHint* bearbeiten, können Sie noch mehr Einfluss ausüben). Die folgende Tabelle listet zunächst die wichtigsten Properties von *Application* auf:

Property	Typ	Bedeutung
Active	Boolean	gibt an, ob ein Fenster dieser Anwendung aktiv ist; wenn *False*, ist eine andere Anwendung aktiv.
Desktop	QWidgetH	gibt den Desktop als Qt-Widget zurück, kann über Qt-Funktionen zur Manipulation des Desktops oder zur Abfage von Informationen darüber verwendet werden (zu dieser Informationsgewinnung eignet sich auch *TScreen*).
Display	PDisplay	stellt einen Zeiger auf die Display-Information von X Window dar. Hiermit können auf niedriger Ebene und auf nicht portable Weise X Window-Funktionen aufgerufen werden, die mit einem Display als Parameter arbeiten (zum »Display« zählt X Window sowohl die Bildschirmausgabe als auch die Tastatureingabe).
EnableQtAccelerators	Boolean	wird normalerweise automatisch gehandhabt, damit die Behandlung von Tastenkombinationen in Qt-Dialogen auch von Qt übernommen wird. Bei direktem Aufruf eines Qt-Dialogs ohne eine CLX-Komponente sollte vorher *EnableQtAccelerator* manuell auf *True* gesetzt werden.
ExeName	String	Name der ausführbaren (*exe*cutable) Datei der Anwendung
Font	TFont	gibt die Standardschrift der Anwendung an, die automatisch an alle Formulare »weitervererbt« wird, deren *ParentFont*-Property *True* ist.
Handle	QApplicationH	Wie die CLX mit *TApplication*, so kapselt auch Qt seine Sicht der Anwendung in einem Objekt. *Application.Handle* gibt nun in der CLX ein Handle auf dieses *QApplication*-Objekt an.
HelpFile	String	Name der Hilfedatei, auf die sich die *HelpContext*-Werte in anderen Properties und an anderen Stellen des Quelltextes beziehen
Hint	String	der Hinweistext, der gerade zu dem Element, über dem sich die Maustaste befindet, angezeigt werden soll. Zur Verwendung mit dem Ereignis *OnHint*, siehe Tabelle unten.
HintColor	TColor	Farbe des Hinweistextes
HintHidePause	Integer	Anzahl der Millisekunden, nach denen das Hinweisfenster ausgeblendet wird (Voreinstellung 2,5 Sekunden).
HintPause	Integer	Anzahl der Millisekunden, die die CLX wartet, bis sie ein Hinweisfenster anzeigt (Standard ist 500).
HintShortCuts	Boolean	Wenn *True*, werden die Hinweistexte in den Hinweisfenstern automatisch um das mit der Komponente verknüpfte Tastenkürzel ergänzt.
HintShortPause	Integer	Wenn bereits ein Hinweis eingeblendet wurde, gibt dieses Property die Wartezeit an, bis ein anderer Hinweis eingeblendet werden kann (Standard: 50 ms).

Property	Typ	Bedeutung
Icon	TIcon	Icon, das die Anwendung als Ganzes repräsentiert.
eyState	TShiftState	gibt den Zustand der Tasten ⇧, Strg und Alt während des letzten vom *Application*-Objekt empfangenen Tastaturereignisses an.
MainForm	TForm	weist auf das Hauptformular der Anwendung.
ShowHint	Boolean	schaltet die Hinweisfunktion für die gesamte Anwendung ein oder aus. Ein Hinweis zu einer Komponente wird immer dann angezeigt, wenn sowohl das *ShowHint*-Property der Anwendung als auch das *ShowHint*-Property der Komponente den Wert *True* haben (bzw. wenn *ParentShowHint* und die Elternkomponente es erlauben).
ShowMainForm	Boolean	Hiermit können Sie die automatische Anzeige des Hauptformulars beim Starten der Anwendung verhindern und Ihre Anwendung so im Hintergrund laufen lassen.
Style	QApplicationStyle	bestimmt den Stil, in dem die Qt-Widgets der Anwendung gezeichnet werden (siehe das dieser Tabelle folgende Beispiel).
rminated	Boolean	wird von der CLX auf *True* gesetzt, wenn die Anwendung beendet wird.
Title	String	Beschriftung des Anwendungs-Icons

Anwendungs-Stile

Das Property *Application.Style* weist auf ein Objekt, durch das Sie das optische Erscheinungsbild Ihrer Anwendung bzw. der darin verwendeten Qt-Steuerelemente grundlegend beeinflussen können. In *Application.Style* enthalten sind weitere Properties, mit denen Sie etwa die Größe von Radioschalter-Knöpfen (*RadioSize*) oder die Länge von Slidern (*SliderLength*) bestimmen können. Vor allem aber können Sie über 20 Events bearbeiten, um die grundlegenden Teile von Widgets selbst zu zeichnen (z. B. *DrawMenuCheck*, *DrawMenuFrame*, *DrawRadio*). Für eine Liste aller Elemente sei hier auf den Quelltext *QStyle.pas* verwiesen, für eine Qt-Dokumentation der Klasse *QStyle* (die Online-Hilfe der ersten Kylix-Version schweigt sich zu diesem Thema nämlich noch aus; beachten Sie jedoch, dass die Benennungen in Kylix dem Namensschema von Kylix angepasst wurden und daher von denen der Klasse *QStyle* abweichen; so wurde beispielsweise aus dem *QStyle::exclusiveIndicatorSize* von Qt in der CLX der Bezeichner *TStyle.RadioSize*).

ür normale Anwendungen dürfte der Hauptreiz von *Application.Style* jedoch nicht im Neuerfinden der Grafikausgabe liegen, sondern in dem sehr einfach möglichen Umschalten zwischen verschiedenen vordefinierten Zeichenstilen. Das Beispielprogramm *ApplicationAndBorderStyles* (Abbildung 3.3) ermöglicht Ihnen in seinem Haupt-

fenster, per Radioschalter-Gruppe zwischen den Stilen *dsWindows*, *dsMotif*, *dsQtSGI* und weiteren umzuschalten. Um die Umschaltung zu implementieren, ist nur eine Zeile Code für das *OnClick*-Ereignis der Schaltergruppe erforderlich:

```
procedure TStyleDemoMainForm.RadioGroup1Click(Sender: TObject);
begin
  Application.Style.DefaultStyle := TDefaultStyle(RadioGroup1.ItemIndex);
end;
```

Voraussetzung ist allerdings, dass die einzelnen Schalter genau der Reihenfolge der Stilkonstanten in der Deklaration von *TDefaultStyle* entspricht, denn nur so kann der *ItemIndex* oben korrekt in einen *TDefaultStyle*-Wert umgewandelt werden.

```
// aus QStyle.pas:
TDefaultStyle = (dsWindows, dsMotif, dsMotifPlus, dsCDE, dsQtSGI,
                 dsPlatinum, dsSystemDefault);
```

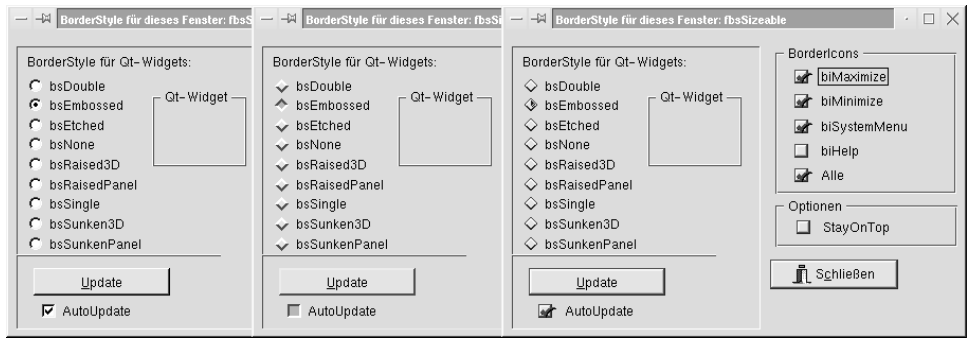


Abbildung 3.4: Das gleiche Fenster in drei verschiedenen Stilen: dsWindows, dsMotif und dsQtSGI.

Ereignisse

Die interessantesten Methoden von *TApplication* sind:

Methode	**Beschreibung**
BringToFront	macht die Anwendung zur aktiven Anwendung, bringt also ihre Fenster in den Vordergrund.
Minimize	minimiert alle Fenster der Anwendung zu einem Symbol, so als wäre das Hauptfenster der Anwendung minimiert worden.
NormalizeTopMosts	setzt zeitweise die Wirkung des Formular-Stils *fsStayOnTop* für alle Fenster der Anwendung außer dem Hauptfenster aus.
ProcessMessages	ist eine von mehreren Möglichkeiten, kooperatives Multitasking zu realisieren (siehe Kapitel 4.8.1). Für ein einfaches Beispiel siehe Kapitel 7.3.7.
Restore	macht *Minimize* wieder rückgängig.

Methode	Beschreibung
RestoreTopMosts	reaktiviert das mit *NormalizeTopMosts* deaktivierte Flag.
ShowException(E: Exception)	meldet dem Benutzer des Programms eine Exception und deren Fehlermeldungs-Inhalt (Property *Message*).
rminate	beendet die Ausführung des Programms.

Weitere Gruppen von Methoden befassen sich unter anderem mit dem Aufrufen der Hilfefunktion (*HelpContext*, *HelpCommand* und *HelpJump*), mit dem manuellen Anzeigen von Hinweisfenstern (*ActivateHint* und *CancelHint*) sowie mit dem manuellen Aufufen von Aktionen (*ExecuteAction*).

Ereignisse

Application ist eine der Klassen, die nicht im Objektinspektor bearbeitet werden können, die aber dennoch Events zur Verfügung stellen. Sie können diese Events mit Methoden Ihres Formulars oder eines anderen Objekts verknüpfen, müssen diese Methoden aber selbst in die Deklaration des Formulars bzw. der Klasse des anderen Objekts einfügen und auch deren Gerüst (Methodenkopf, *begin...end*) selbst erstellen. Ein Beispiel dafür finden Sie in Kapitel 1.8.5, R59. Die folgende Tabelle gibt eine Übersicht über die zur Wahl stehenden Ereignisse:

Event	Parameter für die Methode	Beschreibung
OnActionExecute, OnActionUpdate	Action: TBasicAction; var Handled: Boolean	treten für jede aufgerufene Aktion auf, die nicht bereits in einer *OnExecute*- bzw. *OnUpdate*-Methode der Aktionsliste erledigt wurde.
OnActivate	Sender: TObject	tritt auf, wenn die Kontrolle von einem Fenster aus einer anderen Anwendung zu einem Fenster dieser Anwendung übertragen wird.
OnDeactivate	Sender: TObject	tritt auf, wenn ein Fenster dieser Anwendung deaktiviert und dafür eine andere Anwendung aktiviert wird.
OnEvent	Sender: QObjectH; Event: QEventH; var Handled: Boolean	wird für jede Nachricht aufgerufen, die von der Anwendung empfangen wird.
OnException	Sender: TObject; E: Exception	wird von der Methode *HandleException* erzeugt.
OnHelp	HelpType, HelpContext, HelpKeyword, HelpFile, Handled	wird aufgerufen, bevor das *Application*-Objekt auf automatischem Wege die Hilfedatei aufruft.
OnHint	Sender: TObject	tritt auf, wenn die Maus sich über einer Komponente befindet, zu der ein Hinweistext angezeigt werden *könnte*. Siehe Beispiel in Kapitel 1.8.5, R59.

Event	Parameter für die Methode	Beschreibung
OnIdle	Sender: TObject; var Done: Boolean	wird in nachrichtenfreien Zeiten aufgerufen, siehe auch Kapitel 4.8.1.
OnMinimize	Sender: TObject	tritt bei jeder Minimierung des Hauptfensters bzw. der Anwendung auf.
OnRestore	Sender: TObject	tritt bei jeder Wiederherstellung eines minimierten Hauptfensters bzw. der Anwendung auf.
OnShortCut	var Msg: TMsg; var Handled: Boolean	tritt bei jeder Tastatureingabe vor dem Erzeugen des *OnKeyDown*-Ereignisses auf, eignet sich also zur Realisierung anwendungsweiter Tastenkürzel.
OnShowHint	var HintStr: string; var CanShow: Boolean; var HintInfo: THintInfo	tritt auf, wenn ein Hinweistext in einem kleinen Hinweisfenster gezeigt werden soll.

Viele weitere Themen von anwendungsglobaler Bedeutung werden nicht vom *Application-*, sondern vom *Screen*-Objekt behandelt, dem das folgende Kapitel gewidmet ist.

3.2.4 TScreen

Neben *Application* enthält die Unit *QForms* noch ein weiteres wichtiges globales Objekt: *Screen* hat die Klasse *TScreen* und befasst sich mit Dingen, die auf dem Bildschirm sichtbar sind bzw. mit den Eigenschaften des Bildschirms selbst. Die meisten Properties von *Screen* dienen nur der Information und sind somit nur lesbar. Beschreibbar sind lediglich die Properties *Cursors* und *Cursor* sowie das Property *HintFont*, mit dem Sie ein kleines Detail des Aussehens Ihrer Anwendung anpassen können.

Property	Typ	Beschreibung
ActiveControl	TWidgetControl	bezeichnet die Komponente, die innerhalb der Kylix-Anwendung den Tastaturfokus hat bzw. hatte, bevor die Anwendung deaktiviert wurde. Bei einem Fokuswechsel wird *ActiveControl* gesetzt, bevor das *OnExit*-Event der verlassenen Komponente aufgerufen wird.
ActiveCustomForm und ActiveForm	TCustomForm bzw. TForm	bezeichnet das oberste aller Formulare innerhalb der Kylix-Anwendung. Wenn die Kylix-Anwendung aktiv ist, ist *ActiveCustomForm* bzw. *ActiveForm* das aktive Fenster auf dem Desktop.
ActiveWidget	QWidgetH	Gibt das aktive Fenster im Gegensatz zu *ActiveForm* als Qt-Widget-Handle zurück.

Property	Typ	Beschreibung
Cursor	TCursor	gibt einen Cursor für alle Fenster der Anwendung an und ist dafür gedacht, zeitweise alle anderweitigen Einstellungen (*Cursor*-Properties der Komponenten) zu überschreiben. Wenn *Cursor* auf *crDefault* gesetzt ist, bleiben die *Cursor*-Properties der Komponenten gültig.
Cursors	Cursor-Handle-Array, indiziert über *TCursor*	enthält die Cursor-Ressourcen zu jeder *cr...*-Konstante, siehe Kapitel 3.3.3.
CustomForms/ CustomForm-Count	Integer/array of TCustomForm	beziehen sich auf die allgemeineren *TCustomForms*, ansonsten wie die Properties *Forms* und *FormCount*.
DataModule-Count	Integer	Zahl der in der Anwendung verwendeten *TDataModule*-Objekte
DataModules	array of TData-Module	enthält Verweise auf die einzelnen *TDataModule*-Objekte.
ormCount	Integer	Zahl der gerade bestehenden »normalen« Formulare der Anwendung (die Formulare, die von *TForm* abstammen)
orms	array of TForm	enthält Verweise auf die bestehenden *TForm*-Formulare.
onts	TStrings	Stringliste, die alle im System installierten Schrifttypen beim Namen nennt
Height	Integer	Höhe des Bildschirms in Pixeln
HintFont	TFont	beschreibbares Property, gibt die in Hinweisfenstern verwendete Schriftart an
HintFont	TFont	Schriftart für die Tooltip-Hinweisfenster
PixelsPerInch	Integer	Auflösung des Bildschirms in Punkten pro Zoll, eingestellt vom Anwender oder eingeschätzt vom Betriebssystem, unter Windows typischerweise 96, unter Linux 75 ppi
Width	Integer	Breite des Bildschirms in Pixeln

Ereignisse

Schließlich finden Sie in *TScreen* auch noch zwei Events: *OnActiveControlChange* und *OnActiveFormChange*. Diese werden jedes Mal ausgelöst, wenn der Tastaturfokus innerhalb eines Formulars bewegt wird oder wenn ein anderes Formular aktiviert wird. Während die Komponentenereignisse wie *OnEnter* und *OnActivate* nur bei Aktivierung bestimmter Komponenten oder Formulare eintreten, gelten diese Ereignisse global für die gesamte Kylix-Anwendung. So können Sie z. B. bei jeder *OnActiveControlChange*-Nachricht den Hinweistext zum aktivierten Steuerelement in einer Statuszeile anzeigen und hätten so nicht nur für die Maus-, sondern auch für die Tastaturbedienung eine Hinweisfunktion.

3.3 Grundlegende Gemeinsamkeiten von Steuerelementen

Die verschiedenen Komponententypen haben viele übereinstimmende Properties, Methoden und Ereignisse, die auf die Basisklassen *TControl* und *TWidgetControl* zurückzuführen sind. Nach der allgemeinen Vorstellung dieser Klassen in Kapitel 3.1.3 beschäftigt sich dieser Abschnitt genauer mit deren Fähigkeiten.

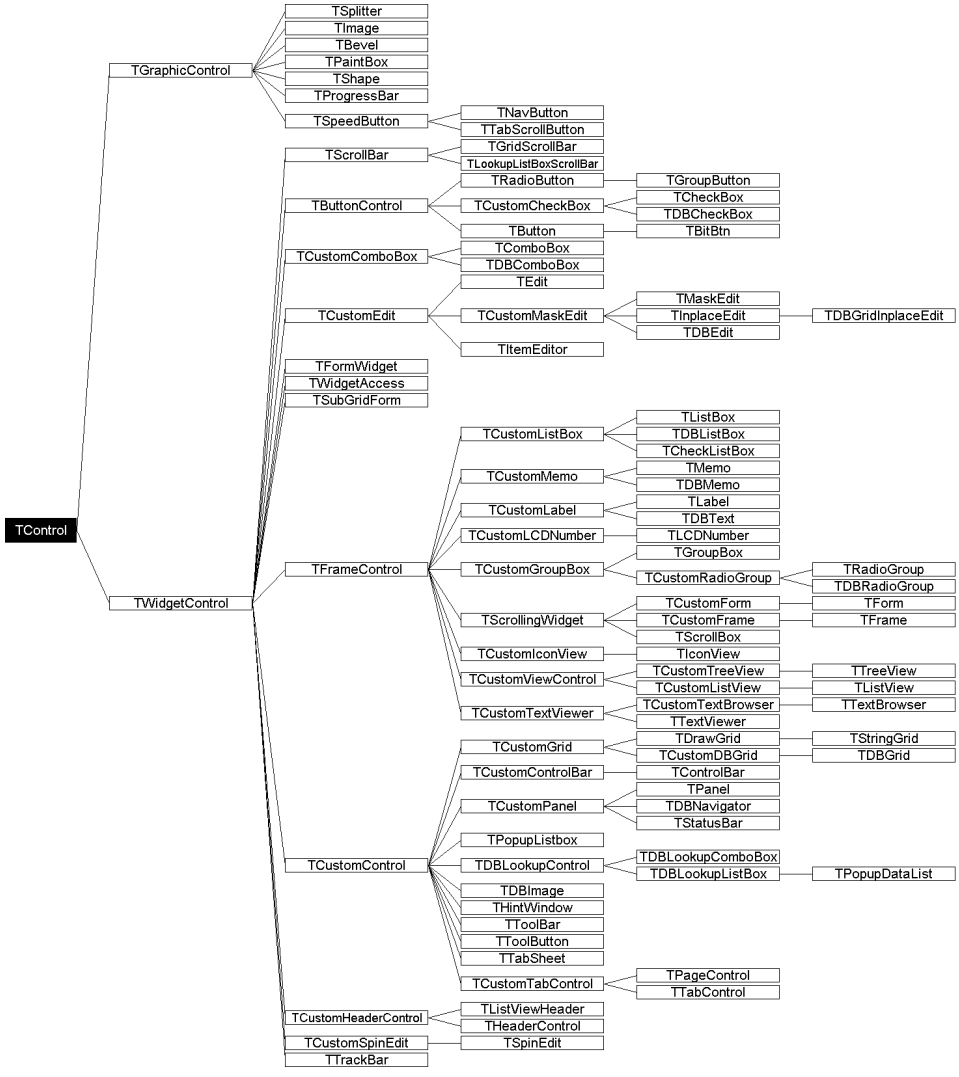


Abbildung 3.5: Die visuellen Komponenten befinden sich unter den von TControl abgeleiteten Klassen

Reichweite der TControl- und TWidgetControl-Properties

TControl und *TWidgetControl* definieren einige Properties, die zwar in den Steuerelementen immer vorhanden, aber nicht immer veröffentlicht sind. So enthält *TControl* B. ein Property *Font* für die Schriftart, ohne dies zu veröffentlichen. Sowohl *TShape* als auch *TSpeedButton* erben *Font* von *TControl*, aber nur *TSpeedButton* holt die Veröffentlichung nach, da *TShape* mangels Text gar keine Anwendungsmöglichkeit für eine Schriftart hat. Das hat zur Folge, dass Sie im Objektinspektor nur das *Font*-Property von *TSpeedButton* verändern können, nicht aber das von *TShape*. Im Programmcode önnten Sie auch *TShape.Font* verändern, würden damit aber keine Wirkung erzielen.

Die meisten Properties, die zwischen verschiedenen Komponenten übereinstimmen, sind in *TControl* oder *TWidgetControl* definiert. Nur wenige Properties, die einen gleichen Namen haben, sind nicht auf eine gemeinsame Basisklasse zurückzuführen. So haben z.B. alle Steuerelemente ein *Caption*-Property, das sie von *TControl* erben. Da die Klasse *TMenuItem* jedoch nicht von *TControl* abgeleitet ist, muss sie für ihre Menüpunkte ein eigenes *Caption*-Property einführen.

3.3.1 Grundlegende Eigenschaften

Die Vielzahl an Properties, Methoden und Ereignissen, die bereits zum Standardvorrat aller (Fenster-)Elemente gehört, wird erst dann einigermaßen übersichtlich, wenn man sie nach bestimmten Funktionsbereichen gruppiert. Da Sie eine alphabetische Übersicht über *alle* Properties einer Komponente auch in Kylix entweder im Objektinspektor oder in der Online-Hilfe bekommen können, verzichtet dieses Kapitel auf eine tabellarische Aufstellung. Es berücksichtigt neben Events und Properties aber auch die im Objektinspektor nicht sichtbaren Methoden – je nachdem, was für einen bestimmten Zweck am ehesten benötigt wird.

Aktiviert und deaktiviert

Alle Steuerelemente sind per Voreinstellung aktiviert, Sie können sie jedoch auch deaktivieren, indem Sie das Property *Enabled* verwenden. Ein deaktiviertes Element *Enabled=False*) ist zwar sichtbar, kann aber keine Eingaben vom Benutzer empfangen. Deaktivierte Schalter können also nicht gedrückt werden und deaktivierte Eingabefelder können nicht editiert werden.

Meistens dient die Deaktivierung von Elementen zur Vereinfachung der Bedienung für den Anwender, denn normalerweise zeigen die Elemente ihre Deaktivierung durch ein graues Erscheinungsbild an. So deaktiviert Kylix z.B. den Mauspalettenschalter zum Anhalten des Programms, wenn das Programm gar nicht läuft.

Wie in diesem Beispiel vermeidet die Deaktivierung unerlaubter Schalter oft unnötige Frustration des Benutzers, denn wenn der Schalter nicht gedrückt werden darf, müsste eine Zuwiderhandlung gegen diese Regel mit einer belehrenden Meldung geahndet werden.

Sichtbar und unsichtbar

Statt ein Steuerelement zu deaktivieren, könnten Sie es auch völlig unsichtbar machen, indem Sie das Property *Visible* auf *False* setzen. Allerdings dient das Unsichtbarmachen meistens anderen Zwecken, so zum Verstecken von Steuerelementen oder Fenstern, die zurzeit nicht gebraucht werden oder die der Benutzer nicht sehen *will*. So kann das Verstecken einer Mauspalette z.B. dazu dienen, mehr Arbeitsfläche auf dem Bildschirm zu schaffen. Derartige Maßnahmen mit der Mauspalette werden normalerweise auf Anweisung vom Benutzer getroffen.

Ein programmgesteuertes Verstecken und Sichtbarmachen von Komponenten wird z.B. dann benötigt, wenn ein Fenster zwar ständig existieren, aber nur bei Gelegenheit sichtbar sein soll. Und es geht schneller, ein Fenster zu verstecken und es später wieder anzuzeigen, als es freizugeben und später wieder ganz neu zu erzeugen.

Als Alternative zum Setzen des *Visible*-Properties können Sie sich auch der Methoden *Hide* und *Show* bedienen. Während *Hide* nichts anderes bewirkt als die Zuweisung *Visible:=False*, sorgt *Show* auch durch einen Aufruf der Methode *BringToFront* dafür, dass das Steuerelement in den Vordergrund geholt wird.

Hinweis: Ein Element, dessen *Visible*-Property *True* ist, kann auch unsichtbar sein, wenn das *Visible*-Property eines übergeordneten Elements gesetzt ist. Über die *Showing*-Methode eines *TControls* können Sie abfragen, ob nicht nur das Control selbst, sondern auch alle übergeordneten Elemente *Visible* sind. *Showing* gibt in diesem Fall *True* zurück, was aber noch nicht bedeutet, dass der Benutzer das Element wirklich sieht, denn es kann ja immer noch von anderen Komponenten oder Fenstern verdeckt sein.

Z-Reihenfolge von Steuerelementen *R22*

Ob sich eine Komponente im Vordergrund oder im Hintergrund befindet, hängt von der im Englischen als *Z-Order* bezeichneten Reihenfolge bezüglich einer gedachten Z-Achse ab, die aus dem Bildschirm herauszeigt und an der sich übereinander liegende Fenster auftürmen. Wie diese Reihenfolge zustande kommt, können Sie an den praktischen Erfahrungen beim Formularentwurf erkennen: Wenn Sie mehrere *TButton*-Komponenten überlappend einzeichnen, so liegt der zuletzt eingezeichnete Schalter immer oben. Ändern können Sie diese Reihenfolge nur mit den lokalen Menübefehlen NACH HINTEN SETZEN und NACH VORNE SETZEN. Dieselbe Wirkung wie diese beiden Menü-

befehle haben die Methoden *TControl.BringToFront* und *TControl.SendToBack*. Diese Methoden funktionieren auf jeder Ebene von Komponenten, also auch für ganze Fenster. Mit *Form1.SendToBack* senden Sie ein ganzes Fenster hinter alle anderen Fenster auf dem Desktop, mit *Komponente.BringToFront* holen Sie eine einzelne Komponente vor alle seine Geschwisterkomponenten (die Komponenten, die dieselbe Elternkomponente bzw. dasselbe Elternfenster haben).

oordinaten

Um die Koordinaten von Steuerelementen festzustellen oder zu ändern, haben Sie mehrere Möglichkeiten:

- Die Einzelwerte stehen in den Properties *Left*, *Top*, *Width* und *Height* zur Verfügung (*Left/Top* gibt die Koordinaten der linken oberen Ecke des Elements an, relativ zum übergeordneten Element, *Width* und *Height* enthalten die Höhe des Elements).
- Um alle vier Werte auf einmal zu ändern, brauchen Sie nicht umständlich vier Zuweisungen zu schreiben, sondern Sie können die Werte als Liste an die Methode *SetBounds* übergeben.
- Im *Bounds*-Property sind diese vier Einzelwerte zu einer *TRect*-Struktur zusammengefasst.
- Sofern Sie sich nur für den Arbeitsbereich eines Steuerelements interessieren, verwenden Sie statt *Height* und *Width* die Properties *ClientHeight* und *ClientWidth*. Oft stimmen diese mit erstgenannten Properties überein, da der Arbeitsbereich die gesamte Komponente umfasst. Formulare sind ein Beispiel, wo *ClientHeight/Width* kleiner sind als *Height/Width*, denn hier werden Titelzeile, Rahmen, Menü und eventuell weitere Teile des Fensters vom Arbeitsbereich abgezogen.
- Die Position des Arbeitsbereichs relativ zum übergeordneten Fenster ist im Property *ClientOrigin* als *TPoint* gespeichert. Im Gegensatz zu den bisher genannten Properties kann dieses nur gelesen werden.
- Im *ClientRect*-Property sind die Einzelwerte *ClientHeight* und *ClientWidth* in einer *TRect*-Struktur zusammengefasst, die für die linke obere Ecke immer 0/0 angibt, da die linke obere Ecke des Steuerelements *relativ zu sich selbst* gemeint ist. *ClientRect* kann nicht beschrieben werden, Änderungen sind nur indirekt durch Änderung der Position und Größe des gesamten Fensters oder durch Zuweisungen an *ClientHeight* und *ClientWidth* möglich.

Hinweis: Bedingt durch die vielfältige Landschaft an Fenster-Managern und die Prinzipien der Fenster-Manager an sich ist es unter X Window manchmal nicht leicht, die genaue Position eines Fensters zu ermitteln. (Die Anwendungen machen dem Fenster-Manager lediglich einen Vorschlag für Position und Größe ihrer Fenster, der Manager entscheidet dann selbst, wie die Fenster platziert werden. Bei vielen Managern kann der Benutzer außerdem noch zwischen verschiedenen Rahmentypen wählen, die dann ebenfalls ohne Einfluss der Anwendungen um die Fenster herumgebaut werden und die Ausmaße der Fenster verändern). Dies betrifft auch die CLX, die in der ersten Version von Kylix manchmal falsche Fensterpositionen zurückliefert (getestet mit KDE 2). Auf Positionsangaben von Komponenten innerhalb von Formularen sollte dagegen Verlass sein.

TRect und TPoint

Die Strukturen *TRect* und *TPoint* werden bei verschiedenen Gelegenheiten verwendet, um Koordinatenangaben zusammenzufassen. So erhält beispielsweise die Grafikfunktion *PolyLine* ein *TPoint*-Array. Während *TPoint* einfach aus den *X*/*Y*-Koordinaten besteht, enthält *TRect* zwei Varianten, so dass Sie es als vier Einzelkoordinaten oder als zwei diagonal gegenüberliegende Punkte ansprechen können. Die beiden Typen sind wie folgt deklariert:

```
type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;
  TRect = record
    case Integer of
      0: (Left, Top, Right, Bottom: Integer);
      1: (TopLeft, BottomRight: TPoint);
  end;
```

Manchmal ist es nützlich, *TPoint*- und *TRect*-Strukturen temporär zu konstruieren, ohne eine Variable dafür deklarieren zu müssen. Dies können Sie mit den Funktionen *Point* und *Rect*, denen Sie die Einzelkoordinaten übergeben und die eine *TPoint* bzw. *TRect*-Struktur daraus machen und zurückliefern. Ein Beispiel dazu folgt im nächsten Abschnitt.

Koordinatensysteme *R16*

Wie die oben genannten Properties zeigen, verfügt jedes Steuerelement über ein eigenes Koordinatensystem, das einfach dadurch definiert ist, dass die linke obere Ecke des Elements die Koordinaten 0/0 erhält und die Koordinaten nach rechts bzw. nach unten ansteigen. Mit Hilfe der bereits genannten Properties können Sie leicht Koordinatenangaben vom System eines Kindelements in das System eines Elternelements umrech-

nen. Meistens ist es aber wichtiger, dass Sie die Koordinaten zwischen einem Element-Koordinatensystem und dem Bildschirm-Koordinatensystem umrechnen können. Hierfür stellt *TControl* zwei Methoden zur Verfügung:

- *ClientToScreen* erwartet als Parameter eine Koordinatenangabe als *TPoint*-Struktur und wandelt diese in Bildschirmkoordinaten um.
- *ScreenToClient* arbeitet umgekehrt und wandelt Bildschirmkoordinaten in Steuerelement-Koordinaten um.

Wenn Sie beispielsweise wissen wollen, an welcher Position des Bildschirms sich die linke obere Ecke einer Komponente befindet, können Sie *ClientToScreen* aufrufen:

```
BildschirmPos := ClientToScreen(Point(Komponente.Left, Komponente.Top));
// was im Prinzip dasselbe ist wie:
BildschirmPos := ClientToScreen(Point(Komponente.Bounds.TopLeft));
```

extinhalt der Steuerelemente

iele Steuerelemente haben einen Textinhalt, der als Überschrift dienen (z.B. bei *TGroupBox*), das eigentliche Element ausmachen (z.B. bei *TLabel*) oder auch zur Benutzereingabe dienen kann (z.B. bei *TEdit*). Trotz der Verschiedenheit der Steuerelemente müssen Sie sich für den Textinhalt nur die ersten beiden der folgenden Properties merken:

- Wenn der Text nicht vom Benutzer geändert werden kann, steht er im Property namens *Caption*. Dies ist z.B. der Fall bei den Komponenten *TLabel*, *TButton*, *TPanel* und auch bei Formularen und Menüpunkten. Üblicherweise können Sie im *Caption*-Text erreichen, dass eines der Zeichen unterstrichen dargestellt wird, indem Sie ein »&« vor dem zu unterstreichenden Zeichen angeben.
- Vom Benutzer änderbarer Text ist über das Property *Text* zugänglich, z.B. in *TEdit* und *TComboBox*.

Align

Jedes Fenster der CLX verfügt über eine Funktion zum automatischen Anordnen seiner Unterfenster bzw. Steuerelemente bei jeder Änderung seiner eigenen Größe. Die Anordnung der Steuerelemente ist von den Einstellungen der *Align*-Properties abhängig. Per Voreinstellung haben diese den Wert *alNone*, was bedeutet, dass die Steuerelemente ihre Position und Größe nicht verändern.

Die anderen Werte von *Align* funktionieren wie folgt: Bei der Erstellung des Formulars und bei jeder nachfolgenden Größenänderung ordnet das Formular zuerst die Elemente, die die Werte *alLeft*, *alRight*, *alTop* oder *alBottom* haben, an seinen Rändern an (links, rechts, oben oder unten). Dabei werden Komponenten, die den gleichen *Align-*

Wert haben, nebeneinander bzw. untereinander angeordnet. Der Bereich, der dann noch frei bleibt, wird den Komponenten mit der Ausrichtung *alClient* zugeordnet. Normalerweise ist das nur eine Komponente, da sich mehrere Komponenten sonst gegenseitig überdecken würden. Jedes Fenster, das Elternfenster ist, ordnet seine Unterfenster auf dieselbe Weise an, daher können Sie auch die Elemente innerhalb eines Panels auf diese Weise ausrichten.

Verwendung von Align zur automatischen Größenanpassung *R17*

Viele Beispielprogramme dieses Buchs verwenden das *Align*-Property zur automatischen Anpassung von Komponenten an die Größenänderung des Hauptfensters, z. B. das ListView-Demoprogramm in Kapitel 3.6.2 (Abbildung 3.9) und die zweite Version des Bashrc-Editors in Kapitel 1.5.6. Sie verfügen jeweils über ein oder mehrere Panels, die mit *alTop* (oder *alLeft*, *alRight*, *alBottom*) ausgerichtet sind. Diese Panels verändern ihre Höhe (Breite) nicht, passen ihre Breite (Höhe) aber der des Fensters an. Ein Panel, das beliebig vergrößert werden kann, belegt den Hauptteil der Fenster und ist mit *alClient* ausgerichtet. Innerhalb einiger dieser Panels befindet sich als Kindfenster des Panels eine Komponente, die mit *alClient* ausgerichtet ist, die dann den gesamten Bereich des Panels einnimmt und damit alle Breiten- bzw. Größenänderungen desselben mitmacht. Um schließlich einen Rand um diese Komponenten zu lassen, wurde das *BorderWidth*-Property des Panels auf einen höheren Wert gesetzt.

Hinweise: Sie können die automatische Ausrichtung von Komponenten zeitweise deaktivieren und »von Hand« steuern, um beispielsweise während einer größeren programmgesteuerten Veränderung im Fensteraufbau eine chaotische Bildschirmanzeige zu verhindern. Siehe hierzu die Online-Hilfe zu den Methoden *TWidgetControl.EnableAlign*, *TWidgetControl.DisableAlign* und *TWidgetControl.Realign*.

Wenn Sie eine ganz andere als die vorgegebene Ausrichtungsfunktion benötigen, können Sie das Ereignis *OnResize* des Formulars selbst bearbeiten und die Größen seiner Elemente wie gewünscht anpassen, wie etwa in R18 beschrieben.

TSplitter *R3*

Eine sehr nützliche Komponente ist *TSplitter*; sie sorgt dafür, dass Sie die Größe eines mit *alLeft*, *alTop*, *alRight* oder *alBottom* ausgerichteten Fensterbereichs zur Laufzeit so flexibel verändern können wie zur Entwurfszeit. Wie schon erwähnt, kann von einer am Fensterrand (oder am Rand einer anderen Komponente) ausgerichteten Komponente immer nur eine Seite verschoben werden. *TSplitter* besteht nun aus einem Balken, der genau an dieser Seite der Komponente angelegt wird.

Nachdem Sie die in der Größe zu verändernde Komponente in das Formular eingefügt haben, fügen Sie einfach ein Exemplar von *TSplitter* hinzu (Seite *Zusätzlich* der Komponentenpalette) und setzen sein *Align*-Property auf denselben Wert wie bei der Komponente, die verändert werden soll. Dadurch erhalten Sie bereits den gewünschten Fensteraufbau, alles weitere erledigt die CLX zur Laufzeit automatisch.

Der Komplexität der Fensterstruktur, die Sie mit *TSplitter*-Komponenten einteilen können, sind theoretisch keine Grenzen gesetzt, denn jeder durch eine *TSplitter*-Komponente geteilte Fensterbereich kann horizontal und vertikal in weitere Teile geteilt werden (falls es sich bei diesem Bereich um eine Komponente handelt, die Kindelemente aufnehmen kann, wie z. B. eine *TPanel*-Komponente oder einfach einen Teilbereich des Formulars).

Drei besondere Properties runden die Fähigkeiten der Splitter-Komponente ab: Durch *ResizeStyle* können Sie unter anderem erreichen, dass die veränderten Fensterteile während des Verschiebens des Splitters ständig neu gezeichnet werden (*ResizeStyle = rsUpdate*). Das Property *AutoSnap* bewirkt, wenn es eingeschaltet ist, Folgendes: Wird eine bestimmte Minimalgröße der mit *alLeft*, *alTop*, *alRight* oder *alBottom* ausgerichteten Komponente unterschritten, so verschwindet diese Komponente ganz, indem ihre Größe auf Null gesetzt wird. Die Minimalgröße wird über das dritte Property angegeben: *MinSize*.

TSplitter wird z. B. für den Andockbereich an der rechten Seite des TreeDesigners-Hauptfensters verwendet (Kapitel 6.8.2).

3.3.2 Maus- und Tastatureingaben

In einfachen Dialogen müssen Sie sehr selten Maus- und Tastatureingaben direkt bearbeiten, da die Komponenten bereits alle wichtigen Eingaben zu *OnClick*- und *OnChange*-Ereignissen umformen. Dieses Kapitel beschreibt die acht Standard-Events, mit denen Sie Maus- und Tastatureingaben direkt abfragen können; Praxisbeispiele dazu gibt Kapitel 6.8.1 mit dem TreeDesigner.

Der Tastaturfokus

Während die Maus sich sehr frei über den Bildschirm bewegen kann, ist die Tastatur darauf angewiesen, dass ihre Eingaben immer an einen klar definierten Ort gelangen. Für den Benutzer ist dieser Ort meistens an der blinkenden Eingabemarkierung zu erkennen (auch Cursor oder *Caret* genannt). Im Programm wird diese Zielrichtung der Tastatureingabe als *Tastaturfokus* bezeichnet. Das Fenster oder das Steuerelement, das die Eingaben erhält, *besitzt* den Tastaturfokus. In der Terminologie der CLX wird dieses Steuerelement auch als *aktives Element* bezeichnet, es befindet sich beispielsweise in den *ActiveControl*-Properties seines übergeordneten Formulars und des *Screen*-Objekts.

Wie bereits in Kapitel 3.1.3 erwähnt, können nur die Komponenten, die von *TWidgetControl* abstammen, den Tastaturfokus erhalten. Jedes Mal wenn der Tastaturfokus zu einem anderen Element übergeht, sei es durch einen Mausklick, durch die Tastaturbedienung oder durch die Steuerung des Programms, erhält die Komponente, die den Tastaturfokus verliert, ein *OnExit*-Event; die Komponente, die ihn erhält, wird mit einem *OnEnter*-Ereignis benachrichtigt.

Es zählt nicht als Wechsel des Tastaturfokus, wenn Sie zwischen verschiedenen Formularen wechseln. Solange ein Formular nicht aktiv ist, hat sein aktives Element (*ActiveControl*) zwar nicht den Tastaturfokus, sobald das Formular aber wieder aktiv wird, erhält es diesen zurück. Daher löst der reine Wechsel zwischen verschiedenen Formularen keine *OnEnter*/*OnExit*-Ereignisse beim Steuerelement, sondern nur *OnActivate*/*OnDeactivate*-Events beim Formular aus.

Tastatureingaben *R85*

Die CLX erlaubt Ihnen eine sehr genaue Abfrage der Tastatureingaben. So können Sie nicht nur mit *OnKeyPress* auf normale Zeicheneingaben reagieren, sondern mit den Ereignissen *OnKeyDown* und *OnKeyUp* sogar das Herunterdrücken und das Loslassen jeder einzelnen Taste erfahren. Der Unterschied wird deutlich, wenn wir untersuchen, welche Ereignisse bei der Eingabe des Großbuchstabens »A« auftreten:

- Sofern der Benutzer nicht die Feststelltaste verwendet, muss er zunächst ein *OnKeyDown* mit ⇧ auslösen.
- Für den Druck auf A gibt es ein weiteres *OnKeyDown*-Ereignis.
- Aus dem A und dem Zustand der Umschalt-Tasten kann die CLX bzw. Qt nun ein *OnKeyPress*-Ereignis machen für das ANSI-Zeichen »A«.
- Danach erfolgen noch zwei *OnKeyUp*-Ereignisse in einer vom Tastaturbenutzer abhängigen Reihenfolge.

Mit *OnKeyPress* können Sie alle Eingaben, die aus einem ANSI-Zeichen bestehen, bearbeiten. Die Bearbeitungsmethode für dieses Ereignis erhält dieses Zeichen im Parameter *Key*. Wenn Sie allerdings Funktionstasten, andere Steuertasten oder gar ⇧, Strg und weitere umschaltbare Tasten abfragen wollen, müssen Sie die anderen beiden Ereignisse bearbeiten.

OnKeyDown und *OnKeyUp* erhalten jeweils die gleichen Parameter:

- In *Key* ist diesmal kein ANSI-Zeichen, sondern ein 16 Bit großer Zeichencode gespeichert. Die Liste der möglichen Codes finden Sie in der Online-Hilfe unter dem Stichwort *Tastencodes*.

- In *Shift* ist der Status der Tasten ⇧, Alt und Strg gespeichert. *Shift* besteht aus einer Menge, die die folgenden Elemente enthalten kann: *ssShift*, *ssAlt*, *ssCtrl*, *ssRight*, *ssLeft*, *ssMiddle*, *ssDouble*. Die letzten vier geben den Status der Maustasten an, wobei *ssDouble* nur bei den Mausereignissen verwendet wird.

Um also beispielsweise die Tastenkombination ⇧+F3 abzufangen, können Sie die folgende Abfrage verwenden:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Key = Key_F3) and (ssShift in Shift) then
    ...
```

Solche Abfragen dürften jedoch nur in Ausnahmefällen notwendig sein, da die CLX und Qt derartige Tastenkombinationen sowohl für Menüpunkte und Aktionen (siehe *Shortcut*-Property) als auch für die normale Navigation in einem Dialog selbst abfragen.

Mauseingaben

Wie bereits in Kapitel 1.5.1 demonstriert, gibt es »komplexe« Mausnachrichten, die besagen, dass die Maustaste einmal gedrückt und wieder losgelassen wurde (*OnClick*) oder gleich zweimal innerhalb kurzer Zeit hintereinander (*OnDblClk*), und einfache Mausnachrichten, die besagen, dass eine Taste gedrückt *oder* losgelassen wurde *OnMouseDown* und *OnMouseUp*) oder dass die Maus bewegt wurde (*OnMouseMove*).

Die letzteren drei Ereignisse geben Ihnen auch Informationen über die Position der Maus und über den Status einiger wichtiger Tasten (der Parameter *Shift* stimmt mit dem *Shift*-Parameter der Tastaturereignisse überein). Ein Beispiel für *OnMouseMove* folgt auf Seite 332, ein Beispiel für die Abfrage aller drei einfachen Mausnachrichten finden Sie in Kapitel 6.4.1.

Weitere Mausfunktionen

In die Klasse *TControl* eingebaut ist auch Unterstützung für weitere wichtige Mausfunktionen:

- Popup-Menüs, die automatisch angezeigt werden können, wenn der Benutzer die rechte Maustaste drückt (Property *TControl.PopupMenu*).
- Der Start ist auch bei Drag&Drop automatisch möglich, im weiteren Verlauf müssen Sie aber ein paar Ereignisse bearbeiten. *TControl* besitzt zwei Properties (*DragCursor* und *DragMode*) und ein paar Methoden (*Dragging*, *BeginDrag* und *EndDrag*), die mit dem einfachen Drag&Drop in Zusammenhang stehen.

Beide Themen werden an anderer Stelle dieses Buchs ausführlich und mit Beispielen behandelt: Popup-Menüs in Kapitel 6.2.4, Drag&Drop in Kapitel 6.8.3.

3.3.3 Aktionen beim Bewegen der Maus

Benutzerfreundliche Anwendungen warten nicht erst darauf, dass der Anwender irgendwelche Eingaben tätigt oder Elemente anklickt, sie reagieren bereits auf die *Bewegung* der Maus. Neben dem automatischen Anpassen der Mauszeigerform unterstützt die CLX automatisch angezeigte Hinweisfenster.

Mauszeigerformen

Auf grafischen Benutzeroberflächen ist es üblich, dass der Mauszeiger sensibel auf den Fensterbereich, auf den er zeigt, reagiert. Im Luftraum des Kylix-Editors nimmt er beispielsweise die Form des üblichen Textcursors *crIBeam* an, während er sich in einen doppelseitigen Pfeil verwandelt, sobald er über einem veränderbaren Fensterrahmen schwebt.

Die Komponenten der CLX machen es Ihnen besonders leicht, die Mauszeigerform zu verändern. Bereits die Klasse *TControl* deklariert das Property *Cursor*, in dem Sie eine Mauszeigerform angeben können, die immer dann aktiv wird, wenn sich die Maus über dem Steuerelement befindet. Der normale Mauszeiger und die Standardeinstellung des *Cursor*-Properties hat den Namen *crDefault*. Auch die anderen Mauszeigerformen werden über eine *cr...*-Konstante angesprochen, z. B. *crHourGlass* für die Sanduhr-Form. Diese Konstante weist in das *Cursors*-Array des *Screen*-Objekts.

Das Cursor-Array

Die verschiedenen verwendeten Mauszeigerformen schwirren in einer Kylix-Anwendung nicht als lose Einzelteile herum, sondern sind an einem passenden Ort aufbewahrt und zu einem Array zusammengefasst. Wie in Kapitel 3.2.4 schon erwähnt, handelt es sich um das Property *Cursors* der *TScreen*-Klasse, zu der es ein globales Objekt namens *Screen* gibt. *Cursors* enthält zu jeder *cr...*-Konstante, die Sie den *Cursor*-Properties zuweisen können, den Cursor, der für diese Konstante verwendet wird. Die Elemente des *Cursors*-Arrays sind nun Handles auf von Qt verwaltete Cursor-Objekte. Noch ohne von diesen Handles Gebrauch zu machen, können Sie z. B. die Bedeutung der Cursor-Konstanten *crIBeam* wie folgt verändern:

```
Screen.Cursors [crIBeam] := Screen.Cursors [crSize];
```

Nach der Ausführung dieser Zeile verwendet Ihr Programm immer, wenn Sie *crIBeam* angeben, den Cursor, der auch unter der Bezeichnung *crSize* ansprechbar ist. Eine sinnvollere Anwendungsmöglichkeit von *Screen.Cursors* ist jedoch die Definition eigener Cursorformen.

Eigene Mauszeigerformen *R15*

Manche Anwendungen begnügen sich nicht mit den vordefinierten Cursorn, sondern definieren ihre eigenen Mauszeigerformen, die dem Benutzer besonders deutlich machen, was er mit der Maus gerade tun kann. Typischerweise ist das in Zeichenprogrammen der Fall.

Um eine eigene Mauszeigerform verwenden zu können, müssen Sie zuerst eine Bitmap-Ressource erstellen (siehe Kapitel 1.4.9) und mit der *$R*-Compileranweisung in Ihre Anwendung einbinden. Dann bietet sich das *OnCreate*-Ereignis des Hauptformulars an, um aus dieser Ressource ein Qt-Cursor-Objekt zu erzeugen und einen Verweis darauf im *Cursors*-Array zu speichern, und zwar an einem von Ihnen gewählten positiven Index (die vordefinierten Cursorform-Konstanten *cr...* sind negative Zahlen).

Um den Cursor auch über einer Komponente anzeigen zu lassen, geben Sie genau diesen Index in deren *Cursor*-Property an. Ein Beispiel hierfür gibt der TreeDesigner aus Kapitel 6, dessen vier Programmstellen, die sich mit der Cursordefinition befassen, im Folgenden zusammengefasst sind:

```
(* Ausschnitt aus der Projektdatei TreeDsgn.dpr *)
{$R CursorBitmap.res} // Einbinden der Ressource
(* Ausschnitt aus dem Interface-Teil von DocForm.pas *)
const // Positiver Index für den Cursor im Cursors-Array:
  crDrawCursor = 1;
(* Ausschnitt aus TDocumentForm.FormCreate: *)
  Screen.Cursors[crDrawCursor] :=
    LoadCursor('DRAWCURSOR', Point(15, 16));
(* Wenn der Benutzer nun als Zeichenwerkzeug den Auswahlzeiger
   anklickt, wird der Mauszeiger der Zeichenfläche wie folgt auf die
   selbst definierte Zeigerform gesetzt: *)
  PaintBox.Cursor := crDrawCursor;
```

Kleiner Nachteil an obigem Listing ist, dass es in Qt keine vordefinierte *LoadCursor*-Funktion gibt, wie sie oben verwendet wird. Daher finden Sie auf der CD in der Unit *SystemApiWrappers* die folgende *LoadCursor*-Funktion, die auch wie oben gezeigt vom reeDesigner verwendet wird. Der erste Parameter gibt den Namen der Ressource an (dies ist nicht der Name der Ressourcen-*Datei*, sondern der Name der Bitmap innerhalb dieser Datei, den man im Ressourcen-Editor festgelegt hat). Der zweite Parameter gibt den »heißen Punkt« innerhalb der Bitmap an, der sozusagen als Spitze des Mauszeigers dient – genauer gesagt den Punkt, der immer über der exakten Mauszeigerposition zu liegen kommt:

```
function LoadCursor(ResourceName: String; HotSpot: TPoint): QCursorH;
var
  Bitmap1 : TBitmap;
  Mask: QBitmapH;
begin
  Bitmap1 := TBitmap.Create;
```

```
    try
      Bitmap1.LoadFromResourceName(HInstance, ResourceName);
      Mask := QBitmap_create;
      try
        QPixmap_createHeuristicMask(Bitmap1.Handle, Mask, false);
        Result := QCursor_create(QBitmapH(Bitmap1.Handle),
                                 QBitmapH(Mask), Hotspot.X, Hotspot.Y);
      finally
        QBitmap_destroy(Mask);
      end;
    finally
      Bitmap1.Free;
    end;
  end;
```

Qt definiert Cursor-Formen wie X Window durch eine zweifarbige 32x32-Pixel Bitmap (je nach Systemfähigkeiten sind auch 16x16, 48x48 und 64x64 möglich) und eine Maske in gleicher Größe:

- Die Bitmap definiert, welche Pixel des Cursors schwarz und welche weiß sein sollen. (X11 erlaubt auch andere Farben als schwarz und weiß, die Schnittstelle von Qt nimmt darauf jedoch keine Rücksicht.)
- Würde alleine die Bitmap für die Erzeugung des Cursors herangezogen, wären alle Mauszeiger quadratische Flächen. Tatsächlich gibt es aber neben schwarzen und weißen Pixeln auch noch transparente Flächen im Cursor, in denen der Bildschirminhalt sichtbar bleibt, die also nicht durch die Cursor-Form abgedeckt werden. Die Maske dient dazu, diese Flächen zu identifizieren.

Bequemerweise ist es aber nicht immer notwendig, die Maske selbst zu erstellen, denn Qt enthält eine Methode, die zu einer gegebenen Mauszeigerform selbst eine Maske zu erraten sucht: *QPixmap::createHeuristicMask*. Grob gesagt ermittelt diese Methode die Randfarbe der Cursor-Bitmap und maskiert dann von dieser Farbe so viel aus, dass gerade noch eine ein Pixel breite Umrandung dieser Farbe um den »Kern« des Cursors bleibt. Dabei erwartet *createHeuristicMask*, dass es genau einen solchen »Kern« in der Mitte des Bitmaps gibt, was natürlich niemand garantieren kann. Aber es funktioniert im Fall eines einfachen Kreuz-Cursors wie des Zeichen-Cursors des TreeDesigners sehr gut.

Alle weiteren Details zu obigen Listings entnehmen Sie bei Interesse bitte der Qt-Online-Dokumentation zu *QPixmap* und *QCursor* bzw. der CLX-Online-Hilfe zu *TBitmap*.

Hinweis zur LoadCursor-Methode: Sie können die Bitmap für den Cursor natürlich auch in einer eigenen Datei speichern, diese Bitmap dann mit *Bitmap.LoadFromFile* laden und daraufhin wie oben gezeigt ein Cursor-Objekt machen. Durch die eigene Datei vermeiden Sie eventuelle Schwierigkeiten mit dem Bereitstellen einer Ressourcen-Datei, die ja bei Kylix noch das Windows-Format haben muss. Aufgrund der Verwendung der Qt-Methoden ist *LoadCursor* auch nach Windows portabel, wo dann nicht mehr die von Windows definierte *LoadCursor*-Funktion verwendet werden braucht.

Hinweis für Windows-Umsteiger: Die »invertierende Farbe« der Windows-Mauszeigerformen (die unter Windows z. B. für den *crIBeam* verwendet wird, der auf weißem Grund schwarz, auf scharzem Grund weiß erscheint) wird von X11 nicht unterstützt. Damit der X11-Cursor auf allen Hintergrundfarben gut zu erkennen ist, sollte ein weißer Cursor mit einer schwarzen Umrandung versehen sein und ein schwarzer Cursor mit einer weißen Umrandung. Letztlich wird der Cursor dadurch noch »sicherer« als ein mit Invertierung arbeitender Windows-Cursor, denn auf bestimmten Hintergrundmustern kann ein nur aus Invertierung bestehender Cursor nahezu unsichtbar werden.

Weitergehende Cursor-Differenzierung *R14*

Nicht immer ist die Zuordnung »eine Komponente – eine Cursorform« differenziert genug. Dies lässt sich schon beim Formularentwurf in der Kylix-IDE erkennen – wenn Sie den Mauszeiger dort über einen der schwarzen Markierungspunkte am Rande einer markierten Komponente halten, ändert sich die Form des Mauszeigers ebenfalls.

Wenn Sie – wie im genannten Beispiel – die Maus in verschiedenen Bereichen *einer* Komponente verschieden aussehen lassen wollen, müssen Sie die *OnMouseMove*-Nachrichten selbst bearbeiten und je nach Mausposition selbst die Cursorform anpassen. Wenn beispielsweise der Mauszeiger bis zu einer Entfernung von 100 Pixeln vom linken Rand des Formulars die Form eines Fadenkreuzes haben soll, im Rest des Formulars aber die Standardform, könnten Sie die folgende *OnMouseMove*-Bearbeitung schreiben:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X,  Y: Integer);
begin
  if x < 100 then Cursor := crCross
    else Cursor := crDefault;
end;
```

Hinweisfenster

Eine weitere Funktion, die mit der Bewegung der Maus zusammenhängt, sind die Hinweisfenster, die auch die Kylix-IDE verwendet, um Ihnen kurze Informationen über die Bedeutung der Schalter unter der Menüzeile zu geben. Auch diese Funktion kön-

nen Sie bereits durch Setzen von nur zwei Properties einschalten. Geben Sie dazu lediglich den Hinweistext im Property *Hint* eines beliebigen Steuerelements an, schalten Sie die Anzeige der Hinweise an (*ShowHint*-Property des Formulars=*True*), und die CLX zeigt zur Laufzeit automatisch einen Hinweistext für das gewählte Element an. Die Verwendung des *Hint*-Properties zur Angabe zweier Hinweisversionen und die Anzeige von Hinweisen in der Statuszeile wurden bereits in Kapitel 1.8.5 erläutert, ebenso wie das Aktivieren und Deaktivieren der Hinweisfunktion für nur einige Komponenten des Formulars.

Weitere Optionen für Hinweistexte sind die Properties *HintPause*, *HintShortPause*, *HintHidePause*, *HintFont*, *HintColor* und *HintShortCuts* von *TApplication* (siehe Kapitel 3.2.3).

3.3.4 Anzeigesteuerung

Neben den schon in Kapitel 3.3.1 genannten wichtigen Properties befasst sich eine Vielzahl von weiteren *TControl*-Properties mit dem äußeren Erscheinungsbild der Komponenten:

- *Color* bestimmt die Farbe des Steuerelements, falls *ParentColor=False*. Im Objektinspektor können Sie eine der vordefinierten Farbkonstanten aus einer Liste auswählen. Zu *TColor*, dem Typ dieser Werte, finden Sie in Kapitel 4.4.2 mehr Informationen.
- *Font* gibt die Schriftart an, die für den *Text* und/oder die *Caption*-Beschriftung verwendet werden soll. *Font* ist ein Objekt des Typs *TFont*. Im Objektinspektor erhalten Sie leicht einen Überblick über die zur Verfügung stehenden Attribute wie z.B. Schriftgröße, Farbe, Fett-, Kursivschrift und Unterstreichung.
- Wenn Sie *ParentColor* auf *True* setzen, ignoriert das Steuerelement den Wert seines eigenen *Color*-Properties und übernimmt stattdessen die Farbe des Elternelements (bei dem ebenfalls *ParentColor* auf *True* gesetzt sein könnte, mit der entsprechenden logischen Wirkung).
- *ParentFont* entspricht dem Property *ParentColor*, bezieht sich jedoch auf die Schriftart des Elements.
- Das Property *Brush* legt üblicherweise fest, mit welchem Pinsel die Hintergrundfarbe des Steuerelements gezeichnet werden soll. Es ist erst in *TWidgetControl* deklariert und wird von dort an die Nachkommen weitervererbt. Nachkommen der Klasse *TControl* müssen bei Bedarf ihr eigenes *Brush*-Property deklarieren. Zu den Untereigenschaften eines Pinsels siehe Kapitel 4.4.2.

Neuzeichnen von Elementen *R98*

Wenn Sie das Aussehen der Komponente modifizieren, beispielsweise indem Sie eines der gerade genannten Properties ändern, muss die Komponente neu gezeichnet werden. Üblicherweise sorgt die CLX automatisch dafür, indem sie die *Invalidate*-Methode aufruft. *Invalidate* ist eine von den *TControl*-Methoden, die Sie in manchen Fällen vielleicht auch direkt aufrufen wollen, beispielsweise wenn Ihnen die automatische Neuausgabe durch *Invalidate* zu lange dauert:

- *Invalidate* erklärt die Komponente für ungültig, was dazu führt, dass sie neu gezeichnet wird, wenn die in der Nachrichtenschlange der Anwendung wartenden Nachrichten abgearbeitet sind.
- *Repaint* zeichnet den Fensterinhalt sofort neu.
- *Update*: wie *Refresh*, allerdings werden nur ungültige Bereiche neu gezeichnet (ähnlich wie *Invalidate* den gesamten Bereich für ungültig erklären kann, können Fensterbereiche auch nur teilweise für ungültig erklärt werden).

Da die meisten Anwendungen die eintreffenden Eingaben und Nachrichten sehr schnell bearbeiten, entsteht bei der Aktualisierung des Fensterinhalts normalerweise keine störende Verzögerung, so dass Sie wahrscheinlich selten in den Standardablauf (Aufruf von *Invalidate*) eingreifen müssen. Mehr Informationen zum Ungültig-Erklären von Fensterbereichen finden Sie in den Kapiteln 6.5.1 und 6.5.3.

3.3.5 Kontrolle über Größe und Position

Die als Nächstes untersuchten Properties von *TControl* definieren auf verschiedene Weise Regeln zur Größeneinstellung und -veränderung von Steuerelementen oder Formularen.

Beschränkungen für die Größenänderung

Das *Constraints*-Property ist seinerseits wieder ein Objekt, was sich im Objektinspektor darin zeigt, dass es aus vier Unter-Properties besteht, die die minimale und die maximale Breite und Höhe des Steuerelements angeben: *MaxHeight*, *MaxWidth*, *MinHeight* und *MinWidth*. Solange Sie diese bei ihrer Voreinstellung belassen, darf das Steuerelement unbegrenzt verkleinert oder vergrößert werden. Wenn Sie aber einen dieser Werte auf eine Zahl größer 0 setzen, wird diese Vorgabe unerbittlich eingehalten: So ist es beispielsweise schon zur Entwurfszeit nicht mehr möglich, ein Formular breiter zu machen, als in *Constraints.MaxWidth* festgelegt.

Bei einigen Komponenten haben Sie auch die Möglichkeit, Größenbeschränkungen dynamisch festzulegen, indem Sie das Ereignis *OnConstrainedResize* mit einer Methode bearbeiten, die neue Minimal- und Maximalmaße zurückgibt. Beim Ereignis *OnCanRe-*

size können Sie eine unmittelbar bevorstehende Größenveränderung durch direkte Größenangaben beeinflussen oder ganz unterbinden. Beide Ereignisse gibt es beispielsweise für *TForm* und *TPanel*.

Regeln für automatische Größen- und Positionsanpassung

Bevor man mit Hilfe des *Constraints*-Property die Veränderlichkeit der Formulargröße einschränkt, sollte man überlegen, ob das Formular nicht auch so gestaltet werden kann, dass die Größenänderung Sinn macht – z. B. dadurch, dass sich auch die im Formular enthaltenen Komponenten dieser Größe anpassen. Hierbei kann das *Anchors*-Property helfen (als eine spezielle Alternative zum bereits in Kapitel 3.3.1 besprochenen und wahrscheinlich häufiger verwendeten *Align*-Property).

Anchors ist ein Mengen-Property mit vier möglichen Elementen*: akLeft, akRight, akTop* und *akBottom*. Jedes davon bewirkt, dass eine Seite des Steuerelements an einer Seite des Elternelements verankert wird. »Verankerung« bedeutet, dass der Abstand zwischen zwei bestimmten Begrenzungen von Eltern- und Kindelement gleich bleibt. Die Voreinstellung von *akLeft* und *akTop* besagt beispielsweise, dass der Abstand zwischen der linken oberen Ecke der Elternkomponente und der linken oberen Ecke der Kindkomponenten konstant gehalten wird. *AkRight* und *akBottom* können in gleicher Weise dazu verwendet werden, einen konstanten rechten oder unteren Rand zur Elternkomponente zu bestimmen.

Mit Hilfe von *Anchors*-Einstellungen ist es möglich, ohne die Verwendung des *Align*-Properties die Fläche eines Fensters in jeder Größeneinstellung voll auszunutzen, jedoch ziehen die Beispielprogramme dieses Buchs dank seiner intuitiveren Funktionsweise das *Align*-Property vor.

Automatische Größeneinstellung

Das neue *AutoSize*-Property von *TControl* ist nur bei ausgewählten Komponenten veröffentlicht. Seine Wirkung können Sie testen, indem Sie eine *TPanel*-Komponente auf ein leeres Formular platzieren, *AutoSize* auf *True* setzen und eine neue Komponente in das Panel setzen: Sofort schrumpft das Panel auf die Größe dieser Komponente zusammen. Wenn Sie mehrere Komponenten im Panel unterbringen wollen, sollten Sie *AutoSize* daher erst setzen, wenn alle Komponenten eingefügt sind.

Ein sinnvoller Anwendungszweck für *AutoSize* sind alle Platzhalterkomponenten, die dazu dienen, zur Laufzeit andere Komponenten aufzunehmen, so zum Beispiel Panels, die zum Andocken von Fenstern oder Toolbars gedacht sind. Jedes Mal, wenn sich die Größe der Kindelemente ändert oder wenn Kindelemente hinzugefügt oder entfernt werden, berechnet eine Komponente, bei der *AutoSize* eingeschaltet ist, ihre Größe neu. Im TreeDesigner wird *AutoSize* für das Andock-Panel auf der rechten Seite des Hauptfensters verwendet (siehe Kapitel 6.8.2).

3.3.6 TWidgetControl

Das meiste, was *TWidgetControl* der Klasse *TControl* hinzufügt, folgt unmittelbar aus den schon dargelegten Unterschieden zwischen *TWidgetControl* und *TControl*. So können *TWidgetControl*-Elemente den Tastaturfokus erhalten ...

astaturfokus *R8 I*

Normalerweise entscheidet der Benutzer, welches Steuerelement den Tastaturfokus erhält, an welches Element also die Eingaben der Tastatur geleitet werden. Verwendet er Tabulatortaste und Pfeiltasten, entscheidet das Property *TabOrder*, in welcher Reihenfolge die einzelnen Elemente in einem Fenster angesteuert werden. Der Wert 3 im *abOrder*-Property besagt beispielsweise, dass das Steuerelement innerhalb der übergeordneten Komponente als Drittes den Tastaturfokus erhält. Dies wurde, zusammen mit dem Property *TabStop*, bereits in Kapitel 1.4.6 unter *Rücksichtnahme auf die Tastaturbedienung* erläutert.

Um den Tastaturfokus programmgesteuert zu bewegen, bedienen Sie sich der Methoden von *TWidgetControl*:

- Die wichtigste Methode ist sicher *SetFocus*, die den Fokus auf das Steuerelement selbst setzt. So erreichen Sie zum Beispiel durch den Aufruf *NextButton.SetFocus*, dass ein Schalter namens *NextButton* fokussiert wird, so dass der Benutzer nur noch die Eingabetaste drücken muss, um diesen Schalter zu betätigen.
- Die Funktion *Focused* gibt mit ihrem booleschen Ergebnis an, ob das Steuerelement den Tastaturfokus besitzt.
- Außerdem gibt es noch die Funktion *CanFocus*, die besagt, ob das Element überhaupt fokussiert werden kann, was nur der Fall ist, wenn es sichtbar *und* aktiviert ist.

Unterelemente

Die wichtigsten Eigenschaften und Methoden für die Unterelemente eines *TWidgetControl* – *ControlCount* und das Array *Controls* sowie die Methoden *InsertControl* und *RemoveControl* – wurden bereits in Kapitel 3.2.2 erläutert. Auch die folgenden Methoden befassen sich in irgendeiner Weise mit den Kindelementen im Array *Controls*:

function ContainsControl(Control): Boolean;	überprüft, ob das als Parameter angegebene *Control* ein Kindelement des aktuellen *TWidgetControl*-Elements ist.
function ControlAtPos(Pos, AllowDisabled): TControl;	gibt das Steuerelement zurück, dass sich an der Position des *TPoint*-Parameters *Pos* befindet. Falls *AllowDisabled=False*, ignoriert die Methode deaktivierte Elemente. Befindet sich kein oder nur ein inaktiviertes Element an der Position, liefert die Funktion *nil*.

procedure FindNextControl(...)	erlaubt die »Vorhersage«, wie sich Tabulator- und Pfeiltasten von einem bestimmten Startpunkt auf den Tastaturfokus auswirken, siehe Online-Hilfe.
procedure ScaleBy(a, b);	skaliert das Steuerelement um den Faktor a/b.
procedure ScrollBy(DeltaX, DeltaY);	führt programmgesteuerten Bildlauf durch.

Fenster-Handle

Das Fenster-Handle stellt die Verbindung zwischen einer *TWidgetControl*-Komponente und dem zugehörigen Qt-Widget dar. Obwohl sie es selbst bereits perfekt verwaltet und Sie sich normalerweise nie darum zu kümmern brauchen, gewährt Ihnen die CLX über das Property *Handle* freien Lesezugriff auf dieses Handle.

Damit können Sie beispielsweise wie folgt einen direkten Aufruf von Qt durchführen:

```
QTextBrowser_setSource(TextBrowser1.Handle, @WideCharURL);
```

Hierbei muss *TextBrowser1* eine CLX-Komponente der Klasse *TTextBrowser* sein. Deren *Handle* weist auf ein *QTextBrowser*-Widget von Qt, das als erster Parameter von *QTextBrowser_setSource* erforderlich ist.

So wie in diesem Beispiel *TTextBrowser* mit einem *QTextBrowser* steht jede *TWidget Control*-Klasse mit einer bestimmten Klasse von Qt-Widgets in Beziehung. Den Zusammenhang, welche Qt-Widget-Klasse zu welcher CLX-Komponente gehört, könen Sie der Tabelle aus der Einleitung von Kapitel 3 entnehmen oder auch der Online-Hilfe zum *Handle*-Property einer speziellen Klasse.

3.4 Formulare, TScrollingWidget und TScrollBox

In diesem Kapitel erreichen wir das Ziel auf dem Weg durch die Vererbungshierarchie der CLX: die Klasse *TForm*, die die Elemente von sieben Vorfahrklassen erbt und über den gemeinsamen Vorfahr *TWidgetControl* eng mit anderen visuellen Komponenten verwandt ist. Auf dem Weg zu *TForm* machen wir noch bei einer weiteren Klasse Halt, der Klasse *TScrollingWidget*. Diese Klasse sorgt bei *TForm* dafür, dass Sie mehr Komponenten in einem Formular unterbringen können, als in das Fenster passen, und dass Sie mit Hilfe von Bildlaufleisten doch alle Elemente erreichen können – sowohl zur Laufzeit als auch zur Entwurfszeit.

3.4.1 TScrollingWidget und TScrollBox

In der CLX befinden sich drei von *TScrollingWidget* abgeleitete Klassen: *TScrollBox*, *TCustomForm* und *TCustomFrame*. Wie der Name schon sagt, ist das wichtigste gemeinsame Merkmal der drei Klassen, das sie vom üblichen *TWidgetControl* unterscheidet,

die besondere Unterstützung des Scrollings. Diese besteht in der automatischen Erzeugung von Scrollbars, die fest in die Fenster dieser Klassen eingebunden sind, also nicht als einzelne (*TScrollBar*-)Komponenten vorliegen, und in der automatischen Durchführung des Scrollens.

TScrollingWidget deklariert nur drei neue Properties, von den Methoden ist besonders die *ScrollInView*-Methode interessant:

Element	Beschreibung
operty AutoScroll; Boolean;	wenn *True*, scrollt die Komponente automatisch alle Elemente in Sichtweite, die den Tastaturfokus erhalten.
operty HorzScrollBar;	weist auf ein *TControlScrollBar*-Objekt für die horizontale Bildlaufleiste.
operty VertScrollBar;	weist auf eine *TControlScrollBar*-Instanz für die vertikale Bildlaufleiste.
ocedure ScrollInView(TControl);	verändert die Bildlaufleisten so, dass das als Parameter angegebene Element sichtbar wird.

ür die beiden *TControlScrollBar*-Objekte *HorzScrollBar* und *VertScrollBar* können Sie sowohl zur Laufzeit als auch zur Entwurfszeit die folgenden Properties verändern:

- Ob eine Bildlaufleiste überhaupt angezeigt werden soll, legen Sie in *Visible* fest. Solange der gesamte Bereich sichtbar ist, wird die Bildlaufleiste unabhängig von *Visible* versteckt.
- *Range* enthält den horizontalen bzw. vertikalen Bereich, über den gescrollt wird. Falls Sie ihn zur Entwurfszeit manuell ändern, wird er nicht mehr automatisch angepasst, wenn Sie Komponenten einfügen oder entfernen.
- Die Schrittweite der Bewegung mit den Pfeilschaltern können Sie im Property *Increment* angeben.
- In *Margin* können Sie einen zusätzlichen Abstand angeben, der zum Bereich *Range* hinzugezählt werden soll. Wenn Sie *Margin* bei der Voreinstellung von 0 Pixeln lassen, wird nur so lange gescrollt, bis die am weitesten rechts bzw. unten befindlichen Elemente gerade noch ganz sichtbar sind.
- *Position* gibt die aktuelle Position der Bildlaufleiste an, sie liegt innerhalb des Bereichs *Range*. Der maximale Wert von *Position* ergibt sich, wenn Sie von *Range+Margin* die Breite des Fensters abziehen.
- Das Flag *Tracking* bezieht sich auf das Ziehen des Reglers mit der Maus. Ist es eingeschaltet, wird der Bereich auch während des Ziehens sichtbar gescrollt, sonst findet das Scrolling erst nach dem Loslassen der Maustaste statt.

Die eigenständige Scrollbar-Komponente *TScrollBar* weist übrigens eine andere Arbeitsweise als die Klasse *TControlScrollBar* auf. *TScrollBar*-Komponenten erlauben Ihnen darüber hinaus eine weitaus genauere Kontrolle des Scrollings über das Ereignis *OnScroll*.

TScrollBox

Die Klasse *TScrollBox* hat den Eigenschaften von *TScrollingWidget* kaum etwas hinzuzufügen. Ein Beispiel zur Anwendung von *TScrollBox* zum Scrollen über einen großen virtuellen Zeichenbereich geben die Kapitel 6.5.4 und 6.6.2.

3.4.2 Die verschiedenen Arten von Formularen

Formulare können nach den folgenden grundlegenden Arten unterschieden werden:

- nach dem Ablauf, in dem sie angezeigt werden: als normales oder als modales Fenster;
- nach dem grundlegenden Rahmentyp, der auch Auswirkungen auf die Funktion des Fensters hat;
- in MDI-Anwendungen wird schließlich zwischen den MDI-Kindfenstern und dem MDI-Hauptfenster unterschieden.

TCustomForm

Wenn Sie sich die Klasse *TForm* im Quelltext ansehen (Unit *QForms.pas*), werden Sie feststellen, dass diese neben den Prozeduren *Cascade*, *Next*, *Previous* und *Tile* (diese sind für MDI-Formulare gedacht) keine neuen Elemente besitzt, sondern lediglich geerbte Properties von *TCustomForm* öffentlich macht. In der Tat ist die gesamte Funktionalität, die in diesem Kapitel besprochen wird, bereits von *TCustomForm* implementiert.

TCustomForm wurde einst unter Windows in Delphi 3 eingeführt, um die Gemeinsamkeiten verschiedener Formulararten zu kapseln: normale Formulare, ActiveX-Formulare und Eigenschaftsseiten von ActiveX-Controls. Letztere beiden stehen zwar unter Linux nicht zur Verfügung, so dass in Kylix bisher *TForm* der einzige *TCustomForm*-Nachfahre ist. Doch wer weiß, was zukünftige Versionen von Kylix noch für neue Klassen bringen werden ...

Der Unterschied zwischen *TForm* und *TCustomForm* spielt jedenfalls in diesem Kapitel keine Rolle, so dass im Folgenden nur noch von *TForm* die Rede sein wird.

Modal und nicht modal

Wie in Kapitel 1.8.1 am Beispiel gezeigt wurde, können Sie ein Fenster modal machen, indem Sie seine *ShowModal*-Methode aufrufen. Eine Voraussetzung ist allerdings, dass

das Formular unsichtbar ist, bevor Sie *ShowModal* aufrufen. Also benötigen Sie theoretisch zwei Zeilen, um ein beliebiges Fenster modal anzuzeigen:

```
FormX.Visible := False;
if FormX.ShowModal = mrOK then
  ...
```

Das Ergebnis von *ShowModal* gibt den Ergebniscode des Formulars an, wie z. B. *mrOK* oder *mrCancel*, wenn es mit dem *OK*- oder *Cancel*-Schalter beendet wurde. Nach der Ausführung von *ShowModal* ist das Fenster wieder unsichtbar, existiert aber weiterhin mit all seinen Daten (und verbraucht auch weiter Systemressourcen).

Ein modales Formular wird »geschlossen« (bzw. unsichtbar), nachdem das Formular-Property *ModalResult* einen Wert ungleich *mrNone* erhält. Dazu gibt es drei Möglichkeiten:

- Wenn der Benutzer das Formular über das Systemmenü schließt, setzt die CLX *ModalResult* selbst auf *mrCancel*.
- Sie können den Wert von *ModalResult* zur Laufzeit setzen, wodurch Sie bewirken, dass das Formular bei der nächsten Gelegenheit geschlossen wird (wenn die aktuelle Nachricht fertig bearbeitet ist).
- Schließlich können Sie im *ModalResult*-Property eines Schalters einen Wert eintragen. Der Schalter kopiert diesen, sobald er gedrückt wird, in das *ModalResult*-Property des Formulars (siehe Kapitel 1.8.1).

BorderStyle

Der Rahmen eines Fensters wird durch die Properties *BorderStyle* und *BorderIcons* festgelegt. In *BorderStyle* haben Sie die Möglichkeit, zwischen sechs verschiedenen Rahmenarten zu wählen. Nicht immer wirkt sich diese Einstellung nur auf das äußere Erscheinungsbild des Formulars aus. Abgesehen davon, dass Sie Fenster ohne Rahmen nicht mit der Maus vergrößern und desgleichen Fenster ohne Titelzeile nicht verschieben können, verweigert die CLX den Fenstern, deren Rahmen im Dialogstil *fbsDialog* gezeichnet wird, eine Menüzeile. Menüs, die Sie im Property *Menu* angeben, erscheinen einfach nicht, wenn Sie den *BorderStyle* auf *fbsDialog* eingestellt haben.

Hinweis: In der ersten Kylix-Version haben die unterschiedlichen Stilkonstanten noch wenig Wirkung (der Quelltext von *QForms.pas* beweist, dass sie weitgehend ignoriert werden) und es ist zu vermuten, dass sich das in späteren Versionen der CLX noch ändern wird. Die folgende Beschreibung der einzelnen Stile geht von der vollen Funktion aller Stile aus, wie sie unter Delphi für Windows zu beobachten ist. Im Test von Kylix 1 jedenfalls waren alle Fenster in der Größe veränderbar, auch die mit den Stilen *fbsSingle*, *fbsDialog* und *fbsToolWin*.

Zur Entwurfszeit werden Formulare immer mit dem Stil *fbsSizeable* angezeigt. Um das Fenster also in seiner wahren Gestalt sehen zu können, müssen Sie das Programm zuerst starten. Die Stile sind im Einzelnen:

- *fbsNone*: Mit diesem Stil erhalten Sie ein Fenster ohne Rahmen und Titelzeile. Falls Sie auch kein Hauptmenü angeben, besteht das gesamte Fenster nur aus seinem Arbeitsbereich. Derartige Fenster werden häufig für Bildschirmschoner oder für den Hintergrund von Installationsprogrammen oder bei anderen speziellen Aufgaben wie Tutorials eingesetzt. Da ein solches Fenster auch kein Systemmenü hat, müssen Sie es per Tastatur (etwa mit der Tastenkombination Alt+F4) schließen, falls Sie kein Steuerelement zum Schließen in ihm unterbringen.
- *fbsSingle*: Fenster mit diesem Stil haben eine Titelzeile, aber einen (theoretisch, siehe Hinweis oben) in der Größe unveränderlichen Rahmen.
- *fbsSizeable* ist die Voreinstellung für ein leeres Formular mit frei manipulierbarem Rahmen. Fenster dieses Typs können als Standard bezeichnet werden.
- *fbsDialog* gibt Ihnen ein (theoretisch) größenunveränderliches Dialogfenster, das außerdem auch keine Hauptmenüzeile haben darf. (Das fehlende Hauptmenü ist so ziemlich der einzige Unterschied, der in der ersten CLX-Version für diesen Stil in *QForms.pas* programmiert ist. Vielleicht kommt später einmal eine spezielle optische Rahmengestaltung hinzu).
- *fbsToolWindow* ist eine Variation des Stils *fbsSingle*, bei der es vorgesehen ist, die Titelzeile des Fensters wie die Titelzeile des Objektinspektors kleiner darzustellen (in Kylix 1 noch nicht implementiert).
- *fbsSizeToolWin* ist ebenfalls als Fenster mit kleiner Titelzeile gedacht, das jedoch größenveränderlich ist.

Falls Sie ein Fenster haben wollen, dessen Größe sich wirklich nicht ändern lässt, sei Ihnen nachfolgendes Rezept empfohlen:

Formulare mit automatisch skalierender Aufteilung *R18*

Wenn Sie besondere Vorstellungen über die dynamische Aufteilung Ihres Formulars haben, die sich nicht durch die in Kapitel 3.3.1 besprochenen *Align*- und *Anchor-Properties* sowie die *TSplitter*-Komponente erfüllen lassen, können Sie die Aufteilung des Formulars bei jedem *OnResize*-Ereignis selbst festlegen. Das folgende Beispiel ist dem Projekt *ThreadDemo1* aus Kapitel 4.8.1 entnommen. Es besitzt zunächst ein mit *AlBottom* ausgerichtetes Panel am unteren Teil des Formulars. Dieses Panel (*Panel5*) dient als Kontrollbereich und wird automatisch von der CLX an sich ändernde Fenstergrößen angepasst. Den restlichen freien Bereich (Höhe *CH* und Breite *CW*) sollen sich *Panel1* bis *Panel4* zu je einem Viertel aufteilen. Die Methode setzt alle Koordinaten der Panels über deren *SetBounds*-Methode:

```
procedure TThreadDemo1Form.FormResize(Sender: TObject);
var
  CW, CH, CW2, CH2: integer;
begin
  CH:=ClientHeight-Panel5.Height;
  CW:=ClientWidth;
  CH2:=CH div 2;
  CW2:=CW div 2;
  Panel1.SetBounds(0, 0, CW2, CH2);
  Panel2.SetBounds(CW2, 0, CW2, CH2);
  Panel3.SetBounds(0, CH2, CW2, CH2);
  Panel4.SetBounds(CW2, CH2, CW2, CH2);
end;
```

Hinweis: Während die VCL ein *OnResize*-Ereignis auch vor dem ersten Anzeigen des Formulars erzeugt, fällt das erste *OnResize*-Ereignis in der CLX (Version 1.0) erst an, wenn die Größe danach erstmals geändert wird. Zu einer richtigen Initialisierung des Formulars muss daher obige Methode unter Linux manuell aus der *OnCreate*-Methode des Formulars aufgerufen werden.

ormulare mit fester Größe *R19*

Als relativ schwieriges Unterfangen erwies sich mit der ersten CLX-Version das Erstellen eines Formulars mit fester Größe. Während dazu in der CLX einfach der Formularrahmenstil *(f)bsSingle* oder *(f)bsDialog* gewählt werden kann, wirken sich diese Stile in der CLX (noch) nicht auf diese Weise aus. Als Alternative bietet sich der Weg über das *Constraints*-Property an. Wenn in diesem sowohl Minimal- als auch Maximalhöhe, Minimal- und Maximalbreite jeweils auf dieselben Werte eingestellt sind, kann das Formular nicht mehr in der Größe geändert werden (die CLX teilt dies dem Fenstermanager auch mit und dieser zeigt dem Benutzer dann auch keine Doppelpfeil-Mauszeiger mehr, wenn die Maus sich über einer Fensterecke befindet, allerdings wird evtl. noch ein Symbol im Rahmen des Fensters angezeigt, welches dem Benutzer doch eine Größenveränderlichkeit suggeriert).

Umständlich ist es jedoch, *Constraints* direkt zur Entwurfszeit auf diese Art einzustellen, denn dann können auch Sie selbst die Formulargröße selbst zur Entwurfszeit nicht mehr mit der Maus ändern, sondern müssen alle vier *Constraints*-Properties manuell anpassen, wenn Sie die Größe einmal verändern wollen. Hier kann es nützlich sein, die Formulargröße zur Entwurfszeit flexibel zu lassen und erst zur Laufzeit auf die tatsächliche Größe fest einzustellen, indem Sie etwa folgenden Code mit *OnConstrainedResize* verknüpfen (auf der CD im Projekt *FixedSizeFrm* zu finden):

```
procedure TForm1.FormConstrainedResize(Sender: TObject; var MinWidth,
  MinHeight, MaxWidth, MaxHeight: Integer);
begin
  Caption := Format('%d / %d', [Width, Height]);
```

```
    with Constraints do begin
      MinHeight := Height;
      MaxHeight := Height;
      MinWidth := Width;
      MaxWidth := Width;
    end;
  end;
```

BorderIcons

Das zweite Property, das bei der Rahmengestaltung eine Rolle spielt, ist *BorderIcons*. Es enthält eine Auswahl der Flags *biSystemMenu*, *biMinimize*, *biMaximize* und *biHelp*, von denen die ersten drei angeben, ob das Systemmenü und die Schalter zum Verkleinern des Fensters auf Symbolgröße und zum Maximieren des Fensters angezeigt werden sollen.

Die Abwesenheit eines Rahmenicons bedeutet jedoch an sich noch nicht die Abwesenheit der mit dem Icon verbundenen Funktionen. So können Sie, wie schon erwähnt, Fenster im Stil *bsNone* über die Tastatur schließen.

Die Funktion des *BorderStyle*-Properties ist unter Linux allerdings stark vom verwendeten Fenster-Manager abhängig, der normalerweise alle Icons anzeigt. Das Ausblenden eines Icons dürfte daher in vielen Systemkonfigurationen nicht möglich sein. In den Tests des Autors mit mindestens zwei Fenstermanagern war jedenfalls nur das *BiHelp*-Icon zum Erscheinen und Verschwinden zu bewegen.

FormStyle

FormStyle enthält verschiedene weitere Variationsmöglichkeiten. Abgesehen vom voreingestellten Wert *fsNormal*, der keine Besonderheiten aufweist, haben Sie die drei folgenden Werte zur Auswahl:

- *fsMDIForm* und *fsMDIChild*: *fsMDIForm* macht ein Formular zum MDI-Hauptfenster, von dem es nur eines in jeder Anwendung geben darf; zu ihm gehören ein oder mehrere Formulare mit dem Stil *fsMDIChild* (siehe Kapitel 6.7).
- Mit *fsStayOnTop* erreichen Sie, dass ein Fenster immer vor allen Fenstern derselben Anwendung zu sehen ist, die nicht diesen Stil aufweisen. Da Sie diesen Stil zur Laufzeit beliebig zwischen *fsStayOnTop* und *fsNormal* umschalten können, bietet es sich an, dafür in der Anwendung einen Popup-Menüpunkt abzubieten, wie Sie es z.B. auch in der Kylix-IDE für den Objektinspektor finden (Menüpunkt IMMER IM VORDERGRUND).

  Wenn Sie diesen Stil dem Hauptfenster Ihrer Anwendung zuweisen, bleibt dieses sogar vor allen anderen Anwendungen sichtbar, die nicht ebenfalls diesen Stil besitzen.

3.4.3 Eigenschaften und Ereignisse der Formulare

Die folgende Tabelle zeigt eine Zusammenfassung der wichtigsten Properties, die ein Formular seinen Vorfahrklassen voraus hat, ausgenommen die bereits bei der automatischen Größeneinstellung/-beschränkung in Kapitel 3.3.5 erläuterten.

Property	**Typ**	**Bedeutung**
Active	Boolean	gibt an, ob das Fenster das gerade aktive Fenster der Anwendung ist.
ActiveControl	TControl	weist auf das Steuerelement, das den Tastaturfokus hat.
BorderIcons	TBorderIcons	siehe Kapitel 3.4.2.
BorderStyle	TFormBorderStyle	siehe Kapitel 3.4.2.
Canvas	TCanvas	Zeichenfläche, die den gesamten Client-Bereich umfasst (von dem z.B. Mauspaletten und Statuszeilen abgezogen sind).
ormStyle	TFormStyle	siehe Kapitel 3.4.2.
Icon	TIcon	Icon für die Symboldarstellung des Formulars, standardmäßig wird das Icon der Anwendung verwendet.
eyPreview	Boolean	Wenn *True*, ist die Vorschaufunktion auf die Tastatureingaben der Steuerelemente eingeschaltet, siehe Kapitel 6.8.1.
Menu	TMainMenu	Komponente, die das Hauptmenü für das Fenster enthält.
ModalResult	TModalResult (Integer)	Rückgabewert für ein modales Formular (siehe Kapitel 3.4.2), vordefinierte Werte sind *mrNone*, *mrOK*, *mrCancel*, *mrAbort*, *mrRetry*, *mrIgnore*, *mrYes*, *mrNo*, *mrAll*.
PixelsPerInch	Integer	Vom System definierte Auflösung des Bildschirms in Punkten pro Zoll (unter Windows normalerweise 96, unter Linux 75).
osition	TPosition	gibt mit folgenden Werten an, ob der Fenstermanager zur Laufzeit die Fensterposition und/oder die Größe des Fensters selbst festlegen soll (*Default*) oder ob die zur Entwurfszeit angegebene Position bzw. Größe verwendet werden soll (*Designed*): *poDesigned*, *poDefault*, *poDefaultPosOnly*, *poDefaultSizeOnly*. Darüber hinaus gibt es noch *poScreenCenter* für bildschirmzentrierte Anzeige, *poDesktopCenter* und *poMainFormCenter* und *poOwnerFormCenter*. Zwischen der Zentrierung auf den Bildschirm und auf den Desktop gibt es natürlich nur dann einen Unterschied, wenn Bildschirm und Desktop verschiedene Bereiche abdecken (bei virtuellen Bildschirmen oder Desktops der Fall).
WindowState	TWindowState	*wsNormalized*, *wsMaximized* und *wsMinimized* geben an, ob das Fenster normal, im Vollbild oder in der minimierten Darstellung angezeigt wird.

Dazu kommen natürlich alle von *TScrollingWidget*, *TWidgetControl* und den anderen Klassen geerbten Properties.

Die folgenden Properties sind ebenfalls Bestandteil von *TForm*, gelten aber nur für MDI-Hauptfenster und sind bis auf *WindowMenu* nicht zur Entwurfszeit ansprechbar (mehr zu MDI-Anwendungen in Kapitel 6.7):

MDI-Property	Typ	Bedeutung
ActiveMDIChild	TForm	gibt bei einem MDI-Hauptfenster das aktive MDI-Kindfenster an.
MDIChildCount	Integer	Zahl der MDI-Kindfenster
MDIChildren	array of TForm	Liste aller MDI-Kindfenster

Wichtige Ereignisse

Ein Formular hat als Elternfenster und Besitzer zahlreicher Komponenten viele Verwaltungsaufgaben, von denen die unbedingt notwendigen natürlich in der CLX bereits implementiert sind. Um Ihnen die Möglichkeit zu geben, am Management des Formulars mitzuwirken, stellt *TForm* Ihnen eine Reihe von Ereignissen zur Verfügung, die über die von den meisten Komponenten verwendeten Ereignisse hinausgehen:

Ereignis	Bedeutung
OnActivate	Benachrichtigung, dass das Formular aktiviert wurde (beim Wechsel zwischen Anwendungen wird stattdessen das *OnActivate*-Ereignis des *Application*-Objekts erzeugt, siehe Kapitel 3.2.3).
OnClose	Hierin entscheiden Sie, wie das Formular geschlossen werden soll, siehe Kapitel 3.4.4.
OnCloseQuery	Voranfrage, ob Formular geschlossen werden darf.
OnConstrainedResize	Dient zum dynamischen Überschreiben der im *Constraints*-Property festgelegten Größeneinschränkungen.
OnCreate	Initialisierungs-Ereignis: Das Formular wurde gerade erzeugt, ist aber noch nicht sichtbar.
OnDeactivate	Ein anderes Formular der Anwendung wird aktiviert.
OnDestroy	Gegenstück zu *OnCreate*: Das Formular wird gerade freigegeben.
OnPaint	Auftrag an das Formular, sich neu zu zeichnen.
OnResize	Die Größe des Formulars (*Width*/*Height* bzw. *ClientWidth*/*ClientHeight*) hat sich geändert.
OnShortCut	Hier können Sie Tastenkürzel definieren, die nicht schon aufgrund eines *ShortCut*-Properties eines Menüs, Schalters oder einer Aktionsliste automatisch von der CLX abgefragt werden. *OnShortCut* wird für jede Tastatureingabe aufgerufen, noch bevor die Standardverarbeitung mit *OnKeyDown* usw. einsetzt.
OnShow	Das Formular wird sichtbar, sobald die *OnShow*-Bearbeitungsmethode ihre Arbeit beendet hat.

3.4.4 Arbeiten mit mehreren Formularen

Obwohl die Verwendung mehrerer Formulare grundsätzlich ganz einfach ist, gibt es einige beachtenswerte Dinge. Zunächst eine kurze Zusammenfassung der Erkenntnisse aus der Projektverwaltung (siehe Kapitel 1.6.3):

- In den Projektoptionen geben Sie ein *Hauptformular* an, das in der Kylix-Anwendung eine besondere Rolle spielt und das als Erstes erstellt wird.
- Alle anderen Formulare, die Sie in den Projektoptionen dafür bestimmt haben, werden zur Laufzeit automatisch erzeugt.
- Die Formulare, deren *Visible*-Property Sie zur Entwurfszeit auf *True* gesetzt haben, werden auch automatisch angezeigt.

Zur Laufzeit können Sie das Hauptformular zwar nicht mehr auswechseln, aber doch dynamisch neue Formulare erzeugen und das Property *Visible* verändern.

Schließen des Hauptformulars und anderer Formulare *R68*

Zwar funktioniert das Schließen eines Fensters zur Laufzeit wie gewohnt bzw. wie es der gerade verwendete Window-Manager verlangt – hinter den Kulissen gibt es jedoch etwas zu bedenken:

Wenn Sie das Hauptformular schließen, wird die Anwendung beendet und alle Formulare werden freigegeben. Wenn Sie ein anderes Formular schließen, so wird die Anwendung nicht geschlossen. Wie kann der Benutzer nun ein aus Versehen geschlossenes »Sekundärformular« wieder öffnen, ohne die gesamte Anwendung neu zu starten?

Die Lösung der CLX für Fenster, die nicht das Hauptformular sind, ist einfach: Standardmäßig gibt sie ein solches Fenster nicht frei, wenn der Benutzer es in der üblichen Art schließt, sondern versteckt es nur. Sie können daher im Hauptfenster z.B. in einem Ansicht-Menü einen Menüpunkt zur Verfügung stellen, mit dem der Benutzer das Formular wieder sichtbar machen kann. Rufen Sie in der Methode für diesen Menüpunkt einfach die Methode *Show* des versteckten Formulars auf. Eine andere Lösung wäre, dem Benutzer nicht zu erlauben, ein solches Fenster zu schließen. Sie müssten dazu das Ereignis *OnQueryClose* entsprechend bearbeiten.

Dynamische Formularerstellung *R33*

In der Voreinstellung (siehe Projektoptionen, Seite FORMULARE) werden alle Formulare zur Laufzeit automatisch erzeugt, auch wenn sie nicht alle gleich sichtbar sind (die Sichtbarkeit steuern Sie ja durch das *Visible*-Property). Nun kann es sein, dass das Erzeugen eines Formulars aus irgendeinem Grund besonders ressourcenaufwändig ist, z.B. weil dabei ein großes Bild geladen oder eine komplexe *TreeView*-Struktur auf-

gebaut wird. Und wenn Sie Ihre Anwendung auch unter Windows zum Laufen bringen wollen, müssen Sie dort bedenken, dass die Windows 9x-Produktreihe nur einen ziemlich beschränkten Vorrat an Ressourcen für Fenster und Steuerelemente besitzt.

Hier bietet es sich an, ein Formular dynamisch zu erzeugen. Wenn Sie zur Laufzeit sogar mehrere Exemplare eines Formulars öffnen wollen, müssen Sie sogar zu dieser dynamischen Erzeugung greifen. Dazu stellen Sie zuerst die automatische Erzeugung des Formulars in den Projektoptionen ab (dies ist zwar nicht unbedingt erforderlich, aber doch sinnvoll, wenn Sie das automatisch erzeugte Exemplar nicht verwenden) und schreiben dann eine Ereignismethode wie im folgenden Beispiel demonstriert.

Diese Vorgehensweise ist sehr gut für Formulare geeignet, die modal angezeigt werden, denn diese können Sie noch in der gleichen Methode, in der sie konstruiert werden, wieder freigeben. Angenommen, Sie haben ein Dialogformular der Klasse *TDialogForm* entworfen, dann können Sie dieses wie folgt dynamisch erzeugen:

```
procedure TForm1.MenuePunktClick(Sender: TObject);
var
  DynamischerDialog: TDialogForm;
begin
  DynamischerDialog := TDialogForm.Create(self);
  try
    DynamischerDialog.ShowModal;
  finally
    DynamischerDialog.Free;
  end;
end;
```

Hinweis: Bei dieser Gelegenheit sei ein kleiner Vergleich zwischen Linux und Windows erlaubt. Auf der CD finden Sie unter dem Namen *ResTestMax* ein Projekt, das unter Windows und Linux läuft und eine unendliche Folge von Fenstern mit jeweils 10 Radioschaltergruppen à 10 Schaltern erzeugt (bzw. versucht, dies zu tun). Am Ablauf des Programms zeigt sich, dass sich unter der ähnlichen Oberfläche sehr gegensätzliche technische Realisierungen im Fenstersystem von X Window und Windows befinden: Während die Fenster unter Windows sehr schnell entstehen (auf dem langsamen Testsystem des Autors noch mit 3 Fenstern pro Sekunde), sind die internen Speicherstrukturen von Windows begrenzt. Unter Windows 95 treten schon vor dem 20. Fenster erste Irritationen auf, über 30 kommt das System nicht hinaus. Selbst unter Windows 2000 tritt ungefähr beim 68. Fenster ein *Win32Error* auf. Ganz anders unter Linux: Hier kann man scheinbar unendlich lange Fenster öffnen lassen. Der Grund liegt aber nicht nur darin, dass Linux die 68-Fenster-Marke von Windows 2000 zu übertreffen in der Lage ist, sondern auch darin, dass sich die Zeit bis ins Unendliche zu strecken scheint: Schon am Anfang des Tests öffnet X Window die Fenster um Faktoren langsamer als Windows, nach 100 Fenstern hat sich diese Geschwindigkeit nochmals um den Faktor 10 reduziert (ein praktisches Arbeiten wäre auf einem mit so vielen Fenstern belegten Desktop kaum möglich, da die Fensterwechsel *noch* viel mehr Zeit in Anspruch nehmen).

Eingriff in das Schließverhalten *R39*

Formulare, die nicht modal angezeigt werden, können Sie weniger gut von außen schließen, da es hier keinen Aufruf von *ShowModal* gibt, dessen Beendigung Sie einfach abwarten könnten. Stattdessen können Sie jedoch das oben erwähnte Standard-Schließverhalten des Formulars verändern.

Normalerweise wird ein Formular, das nicht das Hauptformular ist, nur unsichtbar gemacht, wenn der Benutzer das Systemmenü doppelklickt oder eine gleichwertige Aktion durchführt, die normalerweise zum Schließen des Fensters führt. Um das Formular bei einem solchen Ereignis freizugeben, können Sie das Ereignis *OnClose* bearbeiten. Sie haben dabei die Wahl, ob das Fenster versteckt (*caHide*) oder freigegeben werden soll (*caFree*) oder ob überhaupt nichts geschehen soll (*caNone*). Weisen Sie den Wert Ihrer Wahl dem Variablenparameter *Action* zu, wie im folgenden Beispiel geschehen:

```
procedure TSekundaerFormular2.FormClose
   (Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Natürlich dürfen Sie keine Methoden (wie z.B. *Show*) eines freigegebenen Formulars aufrufen oder Properties und Variablen lesen oder verändern. Da Sie nicht vorhersehen können, wann der Benutzer ein solches nicht-modales Formular schließt, empfiehlt sich eine Bearbeitung von *OnClose* am ehesten für MDI-Kindfenster (siehe Kapitel 6.7.4), deren Vorhandensein für das Hauptfenster keine große Rolle spielt.

Wenn Sie ein Formular, das durch den Benutzer freigegeben werden kann, in anderen Formularen programmgesteuert ansprechen, so brauchen Sie eine Möglichkeit, zu erkennen, ob das Formular existiert oder nicht. Besonders einfach ist dies zu realisieren, wenn Sie eine feste Variable für das dynamisch konstruierte Formular verwenden und diese auf *nil* setzen, sobald das Formular geschlossen wird:

```
var
  Form2: TForm2;

procedure TForm2.FormClose
   (Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  Form2 := nil;
end;
```

Hinweis: Das Setzen von *Form2 := nil* hat keinerlei Auswirkungen auf das Formular selbst, da *Form2* intern nur ein Zeiger auf das tatsächliche Formular ist. Das Setzen von *nil* dient also ausschließlich der Information anderer Programmteile.

3.5 Aufbau von Formularen und Verwendung von Dialogen

Kylix stellt Ihnen einige leistungsfähige Funktionen zur Verfügung, mit denen Sie auch komplexe Formulare übersichtlich gestalten und einfach programmieren können. Dieses Kapitel beschreibt sowohl den Entwurf von Steuerelement-Gruppen und mehrseitigen Formularen als auch deren Handhabung im Programmcode. Diese Techniken werden zwar meistens in Dialogen angewendet, Sie können sie aber auch bei jedem normalen Formular benutzen. Der letzte Abschnitt dieses Kapitels (3.5.6) behandelt schließlich ein reines Dialog-Thema: die Standarddialog-Komponenten, die Ihnen in vielen Fällen den Entwurf eigener Dialog-Formulare ganz ersparen.

3.5.1 Steuerelemente in Gruppen

Zwar können Sie Steuerelemente auf einfache Weise optisch gruppieren, indem Sie sie beispielsweise mit einer *TBevel*-Komponente umranden; eine erheblich leistungsfähigere Methode der Gruppierung besteht jedoch im Aufbau einer Eltern-Kind-Hierarchie der Komponenten. Bei dieser werden die Kindkomponenten einer Elternkomponente als Steuerelement-*Gruppe* angesehen. Diese Art der Gruppierung nutzt nicht nur dem Anwender, der ein wahrscheinlich übersichtlicheres Formular präsentiert bekommt, sondern hilft auch bei der Handhabung des Formulars und oft auch bei dessen Programmierung (siehe beispielsweise Steuerelement-Arrays in Kapitel 3.5.2).

Gruppieren von Steuerelementen zur Entwurfszeit *R1*

Um eine Gruppe von Steuerelementen zu erzeugen, benötigen Sie zunächst eine *Container-Komponente*, die andere Steuerelemente enthalten kann. Als Elternkomponenten kommen etwa *TPanel*, *TScrollBox*, *TGroupBox* und *TPageControl* in Frage.

Um einer dieser Komponenten Steuerelemente hinzuzufügen, haben Sie die folgenden Möglichkeiten:

- Wenn Sie ein Steuerelement direkt mit der Maus auf der Fläche der Elternkomponente einzeichnen, erkennt Kylix, dass das Steuerelement in diese Komponente eingefügt werden soll.
- Um Steuerelemente aus der Zwischenablage in die Gruppe einzufügen, markieren Sie zuerst die Elternkomponente und wählen dann BEARBEITEN | EINFÜGEN.
- Auch wenn Sie eine Komponente mit einem Doppelklick einfügen möchten, müssen Sie vorher die Elternkomponente auswählen, in die die neue Komponente eingefügt werden soll.

Wenn Sie Komponenten nachträglich verschieben, bleiben diese immer innerhalb ihrer bisherigen Elternkomponente und werden an deren Rand abgeschnitten. Insbesondere ist es nicht möglich, ein Steuerelement, das sich auf der freien Formularfläche befindet,

in eine Gruppe aufzunehmen, indem Sie es einfach auf die Fläche der Gruppe schieben. Dadurch liegt das Steuerelement zwar auf (oder unter) der Gruppe, gehört aber nicht dazu. Um Steuerelemente in eine andere Gruppe zu verschieben, müssen Sie sie ausschneiden (BEARBEITEN | AUSSCHNEIDEN) und dann wie oben erwähnt in die gewünschte, vorher markierte Gruppe einfügen.

Oft ist es auch sinnvoll, Gruppen zu schachteln. So können Sie ein Formular beispielsweise durch *TPanel*-Komponenten, die mit *alClient* und *alTop* ausgerichtet sind, in einen größenveränderlichen und einen festen Bereich unterteilen. Innerhalb dieser beiden Fensterhälften, die bereits ihrerseits Gruppen sind, können Sie dann weitere Untergruppen von Schaltern und anderen Steuerelementen unterbringen.

Gruppeneigenschaften

Steuerelement-Gruppen weisen die folgenden Eigenschaften auf, die Sie allesamt auch schon zur Entwurfszeit überprüfen können:

- Die Gruppenkomponente ist das Elternfenster der enthaltenen Elemente (das Formular bleibt weiterhin Besitzer und Ereignisempfänger). Die Kindelemente werden also am Rand des Gruppenfensters abgeschnitten, falls sie über diesen hinausragen.
- Wenn Sie das Gruppenfenster beim Entwurf oder zur Laufzeit verschieben, bewegen sich alle enthaltenen Elemente mit.
- Dementsprechend löschen Sie durch das Entfernen des Gruppenfensters auch alle darin enthaltenen Komponenten.
- Jede Gruppe hat eine eigene Tabulatorreihenfolge für die tastaturgesteuerte Navigation durch das Formular. Über das lokale Menü der Gruppenelemente und der Gruppenkomponente selbst gelangen Sie mit TABULATORREIHENFOLGE... in die Dialogbox, in der Sie die Reihenfolge der Gruppenelemente ändern können. Eine Gruppenebene höher (also in der Elternkomponente der Gruppe, normalerweise also im Formular) taucht die gesamte Gruppe nur als ein Punkt der Tabulatorreihenfolge auf. Die Tabulatorreihenfolge des Formulars wird also durch Gruppen erheblich übersichtlicher.
- Falls die Gruppe Radioschalter enthält, ist zu beachten: Innerhalb jeder Gruppe kann immer nur ein Radioschalter markiert sein.

Um zur Entwurfszeit einen genauen Überblick über die Gruppen bzw. über die gesamte Eltern-Kind-Hierarchie Ihres Formulars zu erhalten, wählen Sie ANSICHT ALS TEXT aus dem lokalen Menü des Formulars. Sie erhalten so eine durch Einrückung hierarchisch gegliederte Textdarstellung des Formulars mit all seinen Komponenten und Properties (siehe auch Kapitel 3.5.5, dort werden wir diese Darstellung editieren).

Gruppen im Programmcode

Wie sich die Eltern-Kind-Hierarchie im Programmcode ausdrückt und wie Sie sie zur Laufzeit verändern können, beschreibt Kapitel 3.2.2. Auch die Themen der folgenden beiden Abschnitte sind eng mit Steuerelement-Gruppen verknüpft: Wenn eine Ereignisbearbeitungsmethode mit mehreren Komponenten verknüpft werden soll (Kapitel 3.5.2), ist es oft auch sinnvoll, diese Komponenten in einer Gruppe zusammenzufassen. Darüber hinaus lassen sich die Komponenten einer Gruppe besonders einfach über Arrays ansprechen (Kapitel 3.5.3).

3.5.2 Gleichzeitige Behandlung mehrerer Komponenten

In Kapitel 1.5.5 wurde bereits gezeigt, wie sich mehrere Ereignisse einer oder mehrerer Komponenten in einer Methode bearbeiten lassen; allerdings wurde dort der Parameter *Sender* noch nicht benötigt. Dies ändert sich mit dem Beispielprogramm *Application AndBorderStyles* (Abbildung 3.5), in dessen Formular Sie das Property *BorderStyle* eines Widgets und das Property *BorderIcons* des Formulars zur Programmlaufzeit interaktiv verändern können, beides über jeweils eine Schaltergruppe. Die linke Gruppe besteht aus einzelnen *TRadioButton*-Komponenten. Mit *TRadioGroup* hätten diese Schalter natürlich erheblich einfacher erstellt werden können, jedoch geht es hier ja um die Demonstration der Behandlung mehrerer gleichartiger einzelner Komponenten. Die rechte Schaltergruppe ist einfach eine *GroupBox* mit mehreren *TCheckBox*-Komponenten.

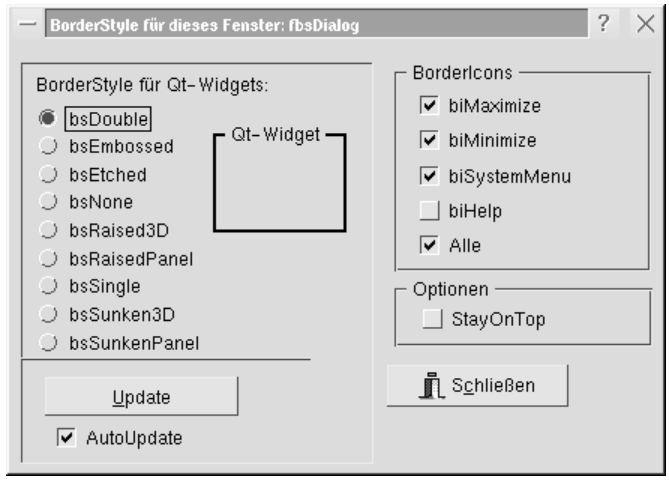


Abbildung 3.6: Die beiden Schaltergruppen dieses Fensters demonstrieren das Tag-Property und die Verwendung von Steuerelement-Arrays

An dieser Stelle geht es um die erste dieser Gruppen: Unter der Überschrift *BorderStyle ür Qt-Widgets* finden Sie in einem *TPanel* neun Radioschalter, die die möglichen Stile ür die Umrandung der meisten Komponenten, die von Qt-Widgets abstammen, demonstrieren. Außerdem gibt es einen Markierungsschalter *AutoUpdate* und den Befehlsschalter *Update*, wobei es für die Funktion der letzteren beiden Elemente keine Rolle spielt, dass sie ebenfalls zur Gruppe gehören.

Solange der Schalter *AutoUpdate* markiert ist, soll der Rahmen des Widgets immer sofort angepasst werden, sobald der Benutzer einen anderen *BorderStyle*-Schalter anwählt. Dafür müssen wir das *OnClick*-Ereignis dieser Schalter bearbeiten. So würde es für den Schalter *bsDouble* beispielsweise genügen, die folgende Methode zu schreiben:

```
procedure TBorderStyleForm.RadioButton1Click(Sender: TObject);
begin
  QtDemoWidget.BorderStyle := bsDouble;
end;
```

Entsprechend würden dann die Methoden für die anderen Schalter aussehen. Wartungsfreundlicher, übersichtlicher und kleiner wird das Programm jedoch, wenn Sie in solchen Fällen alle Schalter mit einer einzigen Methode behandeln. Um zwischen den verschiedenen Schaltern unterscheiden zu können, erhält die Methode im Parameter *Sender* einen Hinweis (um nicht zu sagen einen Zeiger) auf den Schalter, auf den sich das Ereignis bezieht.

Um für alle Schalter am schnellsten eine gemeinsame Methode zu erstellen, wählen Sie im Entwurfsmodus alle Schalter gleichzeitig aus (⇧+Mausklicks). Der Objektinspektor zeigt daraufhin deren gemeinsame Properties an und erlaubt, dass Sie einzelne Properties für alle gewählten Schalter gleichzeitig verändern. Doppelklicken Sie also neben das *OnClick*-Ereignis, um den Rumpf der gemeinsamen Methode zu erhalten. Wenn der erste Schalter etwa den Namen *RadioButton1* hat, gibt der Objektinspektor den automatisch erzeugten Methoden für alle Schalter den Namen *RadioButton1Click*:

```
procedure TBorderStyleForm.RadioButton1Click(Sender: TObject);
begin
end;
```

Unterscheidung der Schalter in einer Methode *R13*

Die Schalter sind zwar schon durch verschiedene Beschriftungen, verschiedene Koordinaten und verschiedene Namen eindeutig unterscheidbar, wesentlich effektiver ist es aber, jedem Schalter darüber hinaus eine eindeutige Kennziffer zu geben. Hierfür ist das Allzweck-Property *Tag* bestens geeignet. Die Schalter der *BorderStyle*-Gruppe erhalten die *Tag*-Werte 0 bis 8, die genau den ordinalen Zahlenwerten der Stilkonstanten entsprechen, die mit den Schaltern eingeschaltet werden sollen. So ist z. B. der ordinale Wert von *bsDouble* gleich zwei, umgekehrt ergibt die Typenumwandlung *TBorderStyle(2)* den Wert *bsDouble* (siehe ordinale Typen in Kapitel 2.4.1).

Hinweis: Aufgrund der willkürlich durchgeführten alphabetischen Sortierung der Radioschalter im Fenster entspricht ihre Reihenfolge nicht der aufsteigenden Reihenfolge der *Tag*-Werte.

Wir können nun also innerhalb der Methode für alle Schalter wie folgt vom *Tag*-Property des gedrückten Schalters auf den zu setzenden *BorderStyle* schließen:

```
BorderStyle := TBorderStyle(RadioButton.Tag);
```

Offen ist allerdings noch, um welchen *RadioButton* es sich handelt. Hier kommt nun der Parameter *Sender* ins Spiel, den diese Methode (wie jede andere Ereignisbearbeitungsmethode auch) erhält. Er gibt an, bei welcher Komponente das Ereignis aufgetreten ist. Bevor wir jedoch auf das *Tag*-Property von *Sender* zugreifen können, müssen wir dem Compiler mitteilen, dass der Parameter *Sender* nicht nur ein *TObject* ist, sondern eine Komponente, die ein *Tag*-Property enthält, also ein *TRadioButton* oder mindestens ein Objekt des Typs *TComponent*. Dies erreichen wir mit dem Ausdruck *Sender as TRadioButton*:

```
procedure TBorderStyleForm.rbDoubleClick(Sender: TObject);
begin
  if AutoUpdate.Checked then // Wenn automatische Aktualisierung gewählt
    QtDemoWidget.BorderStyle :=
      TBorderStyle((Sender as TRadioButton).Tag);
end;
```

Klicken Sie zur Laufzeit beispielsweise auf den Schalter *bsRaised3D*, erhält die obige Methode im Parameter *Sender* die Komponente *rbRaised3D*, deren Property *Tag* auf 5 gesetzt wurde. Die Typenumwandlung *TBorderStyle(Tag)* führt zum Rahmenstil *bsRaised3D*, der dem Property *QtDemoWidget.BorderStyle* zugewiesen wird. Die Komponente ändert zur Laufzeit sofort ihr Aussehen.

Die Umwandlung von *Sender* in *TRadioButton* setzt natürlich voraus, dass Sie diese Methode nur mit Ereignissen verknüpfen, die auch von einem *TRadioButton* stammen.

Das Kapitel 3.5.3 beschreibt die Methode, die die manuelle Aktualisierung des Rahmenstils durchführt (falls der Schalter *AutoUpdate* nicht markiert ist).

Eine Variation

Die ersten vier Schalter der Gruppe *BorderIcons* können auf eine ähnliche Weise behandelt werden. Auch in dieser Schaltergruppe soll jeder Klick auf einen Schalter sofort zu einem Ergebnis führen. Die vier ersten Markierungsschalter entsprechen je einem Flag, das in der Menge *TBorderIcons* enthalten sein kann (zu diesen *TForm*-Properties siehe Kapitel 3.4.2).

Die Methode für die Markierungsschalter geht in zweierlei Hinsicht einen anderen übersichtlicheren, aber nicht unbedingt besseren) Weg als die zuletzt gezeigte Methode für die Radioschalter:

- Sie speichert den in eine *TCheckBox* umgewandelten Parameter *Sender* in der lokalen Variable *Button*, über die sie auf alle Felder des Markierungsschalters explizit zugreifen kann.
- Sie verwendet das *Tag*-Property der Schalter nur zur Unterscheidung und nicht auch noch zur Typenumwandlung der Art *Flag := TBorderIcon(Button.Tag)* (für eine solche Typenumwandlung müssten die *Tag*-Properties auch zuerst in eine Reihenfolge gebracht werden, die der von *TBorderIcon* entspricht).

Schließlich bietet sich bei dieser Methode eine gute Gelegenheit, den »grauen Zustand« der Markierungsschalter auch einmal in diesem Buch zu testen:

```
procedure TBorderStyleForm.CheckBox1Click(Sender: TObject);
var
  Flag: TBorderIcons;
  Button: TCheckBox;
  NewSet: TBorderIcons;
begin
  Button := Sender as TCheckBox;
  case Button.Tag of
    1: Flag := [biMaximize];
    2: Flag := [biMinimize];
    3: Flag := [biSystemMenu];
    4: Flag := [biHelp];
  end;
  if Button.Checked
    then BorderIcons := BorderIcons+Flag
    else BorderIcons := BorderIcons-Flag;
  Alle.State := cbGrayed; (* nach Wahl eines einzelnen Icons sind
      wahrscheinlich weder Alle gewählt, noch Alle nicht gewählt.
      daher wird der Schalter Alle grau dargestellt. *)
end;
```

Ist die Checkbox *Sender* markiert, fügt die Methode dem Mengen-Property *BorderIcons* die neue einelementige Menge *Flag* hinzu, ansonsten zieht sie sie davon ab. Da die Funktionen *Exclude* und *Include* (siehe *Mengen*, Kapitel 2.4.5) nicht mit Mengen-Properties arbeiten können (sie erwarten einen Variablenparameter), muss das einzelne Flag auf die oben gezeigte Weise in die Properties aufgenommen werden.

Der letzte Schalter der Gruppe mit der Aufschrift *Alle* benötigt eine Sonderbehandlung, die im folgenden Kapitel vorgestellt wird.

3.5.3 Steuerelemente in Arrays

Wenn mehrere Komponenten nicht nur einen ähnlichen Zweck, sondern auch den gleichen Typ haben, bietet es sich an, sie nicht nur in einer Gruppe zusammenzufassen, sondern auch in einem Array unterzubringen. Dies hat beispielsweise den Vorteil, dass Sie alle Elemente in einer einfachen Schleife ansprechen können.

Wir beschäftigen uns an dieser Stelle weiter mit dem Beispielprogramm *Application AndBorderStyles*, das bereits in Abbildung 3.5 gezeigt wurde.

Selbst gemachte Arrays *R20*

Durch die *Tag*-Properties erhielten die Schalter des Programms bereits in Kapitel 3.5.2 eine numerische Reihenfolge; es ist dadurch aber noch nicht möglich, sie in einer Schleife anzusprechen. Wenn Sie eine Schleife schreiben wollen (zwei Beispiele folgen), können Sie in der Formulardeklaration ein Array für die Schalter reservieren:

```
type
  TBorderStyleForm = class(TForm)
  ...
  private
    StyleButtons: array[TBorderStyle] of TRadioButton;
  ...
  end;
```

Durch diese Array-Deklaration erhalten Sie ein neunelementiges Array mit den Indizes des Typs *TBorderStyle* (*bsNone* bis *bsEmbossed*). Jeder Schalter soll an der Array-Position gespeichert werden, die seinem *Tag* und dem entsprechenden Rahmenstil entspricht. Selbst definierte Arrays müssen natürlich auch selbst gefüllt werden, eine günstige Gelegenheit dazu bietet das *OnCreate*-Event des Formulars:

```
procedure TBorderStyleForm.FormCreate(Sender: TObject);
begin
  StyleButtons[bsNone]:= rbNone;
  StyleButtons[bsDouble]:= rbDouble;
  StyleButtons[bsEmbossed]:= rbEmbossed;
  StyleButtons[bsEtched]:= rbEtched;
  ...
```

Auch hier nutzen wir wieder die Tatsache, dass die Variablen *rb...* intern Zeiger sind, denn durch das Kopieren dieser Variablen in ein Array erzeugen wir keine Doppelgänger für diese Schalter, sondern setzen lediglich ein Array von Verweisen auf die Schalterobjekte.

Nun zu den Anwendungen der Arrays: Als Erstes soll die bisherige automatische Anpassung des Rahmenstils bei jeder Veränderung der Radioschalter durch den Befehlsschalter *Update* ersetzt werden, den der Benutzer drücken muss, um die Änderungen zu bestätigen und sichtbar werden zu lassen. Die Methode *ManualUpdateClick*

(verknüpft mit dem *OnClick*-Ereignis dieses Befehlsschalters) muss also herausfinden, welcher Schalter gedrückt ist, und dementsprechend den Rahmenstil ändern. Hierzu könnte man in den *OnClick*-Ereignissen der verschiedenen Schalter jeweils das *Tag*-Property des neu gesetzten Schalters in einer Variablen zwischenspeichern. Die Methode *ManualUpdateClick* soll jedoch völlig eigenständig sein und nur das Schalterarray verwenden:

```
procedure TBorderStyleForm.ManualUpdateClick(Sender: TObject);
var
  i: TBorderStyle;
begin
  { mit Low und High werden unabhängig von der Delphi-Version
    alle TBorderStyle-Werte durchlaufen: }
  for i := Low(TBorderStyle) to High(TBorderStyle) do
    if StyleButtons[i].Checked then begin
      QtDemoWidget.BorderStyle := i;
      exit;
    end;
end;
```

Diese Methode durchläuft alle Schalter in einer Schleife und ändert den Stil *BorderStyle*, wenn sie den einzigen gesetzten Schalter (dessen *Checked*-Property gesetzt ist) gefunden hat. Danach überspringt sie die Überprüfung der restlichen Schalter durch die Anweisung *exit*.

Hinweis: Dieses Beispiel soll die Verwendung eines Komponenten-Arrays demonstrieren. Für Radioschalter sollten Sie jedoch die Komponente *TRadioGroup* verwenden, die noch einfacher verwendbar ist als ein Array von Radioschaltern. Statt der fünf Zeilen wäre in der obigen Methode nur eine Zeile notwendig, wenn die einzelnen Optionen in einer *TRadioGroup* untergebracht worden wären:

```
BorderStyle := TBorderStyle(RadioGroup.ItemIndex);
```

ertig vorgegebene Arrays *R2 I*

ür ein zweites Array-Beispiel wenden wir uns nun der zweiten Gruppe des Programms ApplicationAndBorderStyles zu: Es handelt sich dieses Mal um eine *TGroupBox*-Komponente, die fünf Markierungsschalter enthält. Wir werden nun eine weitere ereinfachung durchführen, die allerdings auch mit zusätzlichen (kalkulierbaren) Gefahren verbunden ist. Die vier ersten Markierungsschalter sollen über den Schalter *Alle* gleichzeitig umgestellt werden können. Für das *OnClick*-Ereignis von *Alle* ist die folgende Methode zuständig:

```
procedure TBorderStyleForm.AlleClick(Sender: TObject);
var
  i: Byte;
begin
```

```
    for i := 0 to 4 do
      with GroupBox1.Controls[i] as TCheckBox do
        Checked := Alle.Checked;
end;
```

Diese Methode nutzt die Tatsache, dass alle Markierungsschalter als Kindfenster der *GroupBox1* in deren Array-Property *Controls* liegen (zum *Controls*-Array siehe Kapitel 3.2.2). Wenn die Gruppe nur diese fünf Schalter enthält, ist es auch nicht problematisch, die Indizes derselben herauszufinden – es müssen die Indizes 0 bis 4 sein. Die oben gezeigte Schleife setzt *alle* Schalter auf den Markierungsstatus von *Alle*, also auch *Alle* selbst. Wenn Sie jedoch nicht alle Elemente einer Gruppe auf einmal behandeln wollen, so müssen Sie den Index der einzelnen Elemente kennen. In diesem Beispiel lässt sich nicht ohne weiteres sagen, dass der Schalter *Alle* den Index 4 hat, nur weil er optisch an vierter Stelle untergebracht ist.

Die Indizes des Controls-Arrays *R8*

Es ist nicht möglich, die Indizes im Formular-Designer gezielt zu überprüfen: In der ERSTELLUNGSREIHENFOLGE tauchen die visuellen Komponenten nicht auf und die TABULATORREIHENFOLGE hat nichts mit den Indizes zu tun. Tatsächlich entsprechen die Indizes genau der Reihenfolge, in der die Steuerelemente in die Gruppe eingefügt wurden. Schneiden Sie beispielsweise das zweite von vier Elementen in die Zwischenablage aus und fügen es wieder ein, so wird es am Ende des *Controls*-Arrays wieder eingefügt, wo es bei vier Elementen den Index 3 erhält. Gleiches bewirkt übrigens der lokale Formular-Menüpunkt NACH VORNE SETZEN, während NACH HINTEN SETZEN dem Element den Index 0 zuweist.

Um die Indizes aller Elemente festzustellen, können Sie das Formular in der Textdarstellung untersuchen (ANSICHT ALS TEXT im lokalen Menü des Formulars wählen). Die Reihenfolge der Objektdeklarationen in der Textdarstellung entspricht auch der Reihenfolge der Steuerelemente im *Controls*-Array. Durch Verschieben einzelner Komponenten können Sie in der Textform sogar die Indizes der Komponenten verändern (*Vorsicht:* Eine Verschiebung der falschen Textteile kann zur Beschädigung der `xfm`-Datei führen!).

Falls Sie das *Controls*-Array an einem Index lesen wollen, an dem sich gar kein Kindfenster/Steuerelement befindet, oder wenn Sie ein Element des Arrays in einen inkompatiblen Typen umwandeln, löst die CLX eine Exception aus, die das gerade bearbeitete Ereignis abbricht, die die Anwendung aber ansonsten sicher weiterlaufen lässt.

3.5.4 Mehrseitige Dialoge

Zu den Standardkomponenten gehören auch solche, mit denen Sie Ihr Formular in mehrere Seiten unterteilen können. Ein mehrseitiger Dialog kann eine ganze Palette von einseitigen Dialogen ersetzen. Der große Vorteil liegt darin, dass zum Wechseln zwischen zwei Seiten nur ein (Maus-)Tastendruck notwendig ist, während zum Wechseln zwischen verschiedenen (modalen) Dialogen zuerst der eine geschlossen und dann der nächste wieder geöffnet werden muss.

Im Gegensatz zur VCL unter Windows gibt es in der CLX zunächst nur eine Komponente, die die Grundlage für mehrseitige Dialoge zu liefern in der Lage ist: *TPageControl*. Grundsätzlich ist diese kompatibel mit der *TPageControl*-Komponente unter Windows, jedoch ist sie etwas flexibler und kann auch die *TNotebook*-Komponente von Delphi für Windows ersetzen (eine Komponente, die lediglich aus mehreren Seiten besteht, ohne diese Seiten über ein Register ansteuerbar zu machen).

Das Beispielprogramm *PageControlNotebook* (Abbildung 3.6) wird eine Reihe von Verfahren demonstrieren, mit denen sich mehrere Seiten eines Fensters umschalten lassen.



Abbildung 3.7: Ansichten des Beispielprogramms

Verwendung der TPageControl-Komponente R27

In der Tat ist *TPageControl* als Komponente so einfach handhabbar, dass wir uns hier nur kurz mit ihr zu beschäftigen brauchen. Um einen mehrseitigen Dialog mit Hilfe dieser Komponente zu entwerfen, können Sie die folgenden Schritte ausführen:

- Im einfachsten Fall fertigen Sie eine Kopie des *Registerdialogs* der Objektablage an (DATEI | NEU, Seite *Formulare*, Auswählen des Radioschalters *Kopieren* und des Icons *Dialogfeld mit Registerseiten*). Sie erhalten dann nicht nur drei Demonstrationsseiten und die drei üblichen Schalter *OK*, *Abbrechen* und *Hilfe* in ihrer üblichen Position, sondern auch zwei unsichtbare *TPanel*-Komponenten, die das *PageControl* auch bei Größenänderung des Dialogs immer anpassen – und das mit einem stabilen Rand zu den Grenzen des Formulars. Alternativ dazu können Sie natürlich auch selbst eine *TPageControl*-Komponente in Ihr Formular einfügen, aus deren lokalem Menü Sie mit NEUE SEITE neue Seiten erstellen können.
- Die einzelnen Seiten des *PageControls* stehen Ihnen nun in Form von *TTabSheet*-Komponenten zur Verfügung. Durch einen Klick ins Innere des *PageControls* wählen Sie jeweils das *TabSheet* für die aktuelle Seite aus. Über deren *Caption*-Property bestimmen Sie die Seitenbeschriftung des *PageControls*, über den *PageIndex* die Reihenfolge der Seiten, über das Property *Enabled* können Sie alle Elemente dieser Seite gleichzeitig aktivieren und deaktivieren. Löschen Sie ein *TabSheet* mit Hilfe von Entf, so entfernen Sie damit eine Seite des *PageControls*.
- Um die einzelnen Seiten mit Komponenten zu füllen, schlagen Sie sie einfach auf und zeichnen die Komponenten ein.

Erwähnenswert sind auch die Ereignisse von *TPageControl*: Es gibt nicht nur das übliche Ereignis *OnChange*, das Sie nach einer Seitenanwahl benachrichtigt, sondern auch ein Ereignis *OnChanging*, das Ihnen *vor* dem Umschlagen einer Seite erlaubt, weitere Maßnahmen zu treffen oder das Umschlagen gar zu verhindern (durch Setzen des Methodenparameters *AllowChange* auf *False*).

Das Beispielprogramm

Das Beispielprogramm, das Sie unter dem Namen *PageControlNotebook* auf der CD-ROM finden, verwendet zu Demonstrationszwecken zwei verschiedene Möglichkeiten der Dialoggliederung in einem einzigen Formular (Abbildung 3.6): Das gesamte Formular wird zunächst von einem äußeren *TPageControl* ausgefüllt, dessen Seiten über ein am oberen Rand untergebrachtes Register umgeschaltet werden können. Auf diesen Seiten befindet sich ein weiteres PageControl mit den fünf phantasievollen Seitennamen *Allgemeines*, *Details*, *Details-Editor*, *Details-Grafik* und *Drucker*. Diese Seiten können Sie jedoch nicht nur mit dem standardmäßigen Register, sondern auch über

eine Liste, einen TreeView und eine Menge von Aktionsschaltern umschalten. Welche dieser Varianten zum Einsatz kommt, wird gerade über das Register des äußeren PageControls ausgewählt.

Hinweis: Das innere PageControl ist immer sichtbar, egal, welche Seite des äußeren Controls gerade aufgeschlagen ist. (Damit es immer sichtbar ist, befindet sich die innere *PageControl*-Komponente *auf* dem *PageControl*, ist also keiner von dessen Seiten als Kindkomponente untergeordnet.)

Programmierung und Properties

An der Programmierung der Steuerelemente auf den verschiedenen Seiten ändert sich nichts: Alle Komponenten sind im Programm immer ansprechbar. Allerdings ist etwas mehr Vorsicht geboten als bei einseitigen Dialogen: Wenn Sie zwei zueinander gehörende Kontrollelemente, zum Beispiel einen Schalter und eine Liste, die durch den Schalter verändert wird, auf verschiedenen Seiten unterbringen, beeinträchtigt das zwar nicht die Funktion des Programms, dafür aber den Gemütszustand des Anwenders.

Da eine *TPageControl*-Komponente die Seitenanwahl völlig automatisch steuert, verursacht sie normalerweise überhaupt keinen Mehraufwand an Programmierung. Nur wenn die Seitensteuerung wie in diesem Beispielprogramm von separaten Steuerelementen übernommen werden soll, sind pro Dialog ca. zwei zusätzliche Zeilen Code erforderlich, denen wir uns in einem der folgenden Abschnitte zuwenden. An dieser Stelle seien die wichtigsten Properties von *TPageControl* zusammengefasst, die Ihnen im Code Informationen über die vorhandenen Seiten und die gewählte Seite geben können:

- *Pages* ist ein Array der *TTabSheet*-Komponenten, von denen jede jeweils eine Seite des PageControls repräsentiert.
- *ActivePage* weist direkt auf die *TTabSheet*-Komponente, die für die gewählte Seite steht.
- *ActivePageIndex* gibt den Index von *ActivePage* im Array *Pages* an.
- Die Eigenschaften der einzelnen Seiten sind als Property der *TTabSheet*-Komponenten zu finden, z.B. *Pages[0].Caption* für den Titel der ersten Seite.

Seitenübergreifende Elemente

Komponenten, die immer sichtbar sein sollen, unabhängig davon, welche Seite gerade aufgeschlagen ist, werden in vielen Anwendungen außerhalb der Seiten untergebracht, so z.B. die Schalter *Ok*, *Abbruch* und *Hilfe* in den Dialogen der Kylix-IDE. Sie können jedoch auch innerhalb der Seiten Komponenten unterbringen, die immer angezeigt werden.

Dafür müssen Sie sich eines einfachen Tricks bedienen: Fügen Sie diese Komponenten so ein, dass Kylix sie nicht der *PageControl* oder *Notebook*-Komponente unterordnet. Positionieren Sie sie also zuerst außerhalb jeglicher Komponenten, die Kylix als *Parent* missverstehen könnte, so dass das Formular zum *Parent* dieser Komponenten wird, und schieben Sie die Komponenten erst danach auf die Seitenfläche. (Wenn das gesamte Formular durch die Seiten-Komponente belegt wird, ist das so einfach nicht möglich. Wählen Sie dann das Formular aus der aufklappbaren Liste direkt unter der Objektinspektor-Titelzeile aus und fügen Sie die Komponente mit einem Doppelklick in die Komponentenpalette in das Formular ein.)

Klassen, die wie z.B. *TBevel* von *TGraphicControl* abstammen, können Sie leider nicht auf verschiedenen Seiten gleichzeitig anzeigen, da diese immer unterhalb aller *TWidgetControl*-Steuerelemente und somit auch unter den PageControls liegen und daher unsichtbar werden, wenn sie keine Kindelemente dieser Komponenten sind.

Alternative Seitenauswahlverfahren *R28*

Wir kommen nun zu der Implementierung der drei alternativen Seitenauswahlverfahren für das innere *PageControl* des Beispielprogramms – der Liste, dem TreeView und den Aktionsschaltern. Dazu müssen wir zunächst das hauseigene Register des PageControls abschalten. Wie Sie aus Abbildung 3.6 entnehmen können, soll das Register nur angezeigt werden, wenn auf dem äußeren PageControl die Seite *PageControl-Steuerung* aufgeschlagen ist. In diesem Fall befindet sich rechts vom inneren PageControl noch eine Radioschalter-Gruppe, mit der Sie den Stil der eingebauten PageControl-Steuerung auswählen können. Neben den standardmäßigen *tsTabs* können Sie nämlich auch über Schalter (*tsButtons* oder *tsFlatButtons*) zwischen den Seiten umschalten.

Das folgende Listing zeigt die *OnChange*-Methode für das äußere PageControl – sie dient dazu, das Register des inneren PageControls abzuschalten, falls nicht die erste Seite des äußeren PageControls (namentlich die *TTabSheet*-Komponente *TS1*) aufgeschlagen ist – und die *OnClick*-Methode für die Radioschaltergruppe zur Wahl des Registerstils für das innere PageControl (der Name des inneren PageControls ist *Nootbook*):

```
procedure TForm1.OuterPageControlChange(Sender: TObject);
begin
  if OuterPageControl.ActivePage <> TS1 then
    Notebook.Style := tsNoTabs
  else
    RadioGroup1Click(nil); // Stil des Registers setzen
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
```

```
      0: Notebook.Style := tsButtons;
      1: Notebook.Style := tsFlatButtons;
      2: Notebook.Style := tsTabs;
    end;
  end;
```

Nun soll das innere PageControl über eine ListBox gesteuert werden. Die Einträge der ListBox sollen die Titel der verschiedenen Seiten des PageControls angeben. Hier können wir es uns ersparen, diese Titel zur Entwurfszeit in das *Items*-Property der ListBox einzutragen. Es genügt, bei der Erzeugung des Formulars (*OnCreate*) in einer kleinen Schleife die Titel direkt auszulesen und in die ListBox zu schreiben:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  ListBox1.Items.Clear;
  for i:=0 to Notebook.PageCount - 1 do
    ListBox1.Items.Add(Notebook.Pages[i].Caption);
end;
```

Da die Indizes der ListBox-Einträge nun mit den Indizes der Seiten in *Notebook* übereinstimmen, muss, wenn ein anderer Eintrag der ListBox gewählt wird, nur dessen Index an das *ActivePageIndex*-Property des PageControls weitergegeben werden, um auf die passende Seite umzuschalten:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  NoteBook.ActivePageIndex := Listbox1.ItemIndex;
end;
```

Zum Abschluss sollen die PageControl-Seiten noch über ein TreeView und über eine Menge von Aktionsschaltern ausgewählt werden. In beiden Fällen wurden die Seitentitel bereits zur Entwurfszeit in das TreeView respektive die Buttons geschrieben, so dass hier keine Initialisierung in *OnCreate* erforderlich ist wie für die Einträge der ListBox. (Im Falle des TreeViews gibt es auch kaum eine Alternative zu einer Initialisierung schon zur Entwurfszeit, denn das TreeView hat ja eine hierarchische Struktur – in diesem Fall enthält es einen Knoten »Details« mit zwei Untereinträgen – und diese hierarchische Struktur kann nicht aus der reinen PageControl-Beschriftung gewonnen werden.)

Die *OnClick*-Methoden für das TreeView und die Buttons enthalten nun nicht mehr viel Neues. In beiden Fällen muss wieder *ActivePageIndex* gesetzt werden. Beim den Buttons wurde die passende Seite wieder im *Tag*-Property gespeichert, das bereits in Kapitel 3.5.2 vorgestellt wurde. Im Falle des TreeViews gilt es, den absoluten Index des gewählten Eintrages (*AbsoluteIndex*) festzustellen. Dieser entspricht – im Gegensatz zum Property *Index*, welches den Index des Eintrages in der Liste seiner in derselben

Hierarchieebene liegenden Geschwisterelemente angibt – genau dem Index der aufzuschlagenden Seite:

```
procedure TForm1.AllgemeinesClick(Sender: TObject);
begin
  NoteBook.ActivePageIndex:=(Sender as TButton).Tag;
end;

procedure TForm1.TreeView1Click(Sender: TObject);
begin
  NoteBook.ActivePageIndex:=TreeView1.Selected.AbsoluteIndex;
end;
```

3.5.5 Formulardateien im Textmodus editieren

So komfortabel der Entwurf von Formularen in Kylix ist – es gibt doch einige Arbeitsabläufe, die visuell nicht oder nur sehr schwierig durchzuführen sind, etwa durch teilweisen Neuaufbau des Formulars oder umfangreiche Zwischenablageoperationen. Zu diesen Arbeitsabläufen gehören z. B.:

- Auswechseln einer Komponente, die andere Komponenten enthält, wobei diese enthaltenen Komponenten weiter bestehen sollen. Dies wird visuell besonders hart, wenn etwa bei der Portierung einer Windows-Anwendung nach Linux eine ganze *TNotebook* durch eine *TPageControl* ersetzt werden soll und alle Komponenten von den Notebook-Seiten auf Seiten des PageControls wechseln sollen.
- Austausch der Basisklasse des Formulars für die Formularvererbung,
- Einfügen und Löschen von Komponenten aus der Eltern-Kind-Hierarchie (z. B. Einfügen einer neuen Panel-Komponente als *Parent* für eine Ansammlung anderer Komponenten oder Entfernen eines nicht benötigten Panels),
- Suchen und Ersetzen von Property-Werten.

Für solche Aufgaben gibt es in Kylix die Textdarstellung eines Formulars, die ja in den letzten Abschnitten bereits angesprochen wurde. Zum Austausch der Basisklasse bei der Formularvererbung finden Sie in Kapitel 3.7.1 eine detaillierte Anleitung. An dieser Stelle geht es um den erstgenannten Punkt, also quasi um das Ändern der Klasse einer Komponente.

Austausch der Komponentenklassen *R7*

Als Beispiel soll ein mehrseitiger Dialog im *TTabbedNotebook*-Stil in einen mit *TPageControl* arbeitenden Dialog umgewandelt werden. Der visuelle Weg kommt hier kaum in Frage, denn versuchen Sie einmal, alle im Notizbuch befindlichen Komponenten im Entwurfsformular auszutauschen, ohne alle Elemente auf den einzelnen Seiten mühsam vom alten in das neue Mehrseiten-Steuerelement zu verschieben.

Um das Formular als Textdatei editieren zu können, müssen Sie in Delphi für Windows die Option zur Speicherung im Textformat gewählt haben[2]. Um dieses Beispiel auch ohne Windows-Delphi nachvollziehen zu können, finden Sie auf der CD das Formular `Kapitel3/DelphiNotebook.dfm`, das bereits als Textdatei abgespeichert wurde. Es enthält die besagte *TTabbedNotebook*-Komponente, auf deren Seiten diverse andere Komponenten enthalten sind.

Sie können die Datei `DelphiNotebook.dfm` nun unter Linux in einem Texteditor editieren. Der Übersicht halber wurde das folgende Listing auf wenig mehr als das Wesentliche gekürzt, denn das vollständige Listing aller in dem Beispielformular vorkommenden Properties und Komponenten hätte mehrere Seiten in Anspruch genommen:

```
object Form1: TForm1
  Left = 202
  Top = 118
  BorderStyle = bsDialog
  Caption = 'Notebook-Demo: Seitenauswahl über...'
  ...
  object TabbedNotebook1: TTabbedNotebook
    Left = 0
    Top = 0
    Width = 419
    Height = 269
    Align = alClient
    TabFont.Charset = DEFAULT_CHARSET
    ...
    object TTabPage
      ...
      Caption = '... Tabs'
      object Bevel2: TBevel
        ...
```

Es genügen zwei kleine Änderungen, um das TabbedNotebook zu einem PageControl-Notizbuch zu machen:

- Ändern Sie die Typenangabe *TTabbedNotebook* in *TPageControl*.
- Ersetzen Sie mit der Suchen&Ersetzen-Funktion des Editors alle Vorkommnisse von *TTabPage* durch *TTabSheet*. Die Komponentenklasse *TTabPage* stellt die Seiten der *TTabbedNotebook*-Komponente dar und ist zur Entwurfszeit nicht ansprechbar.

Die Änderungen sind im folgenden Listing kursiv hervorgehoben, wobei die Zeile *object TTabSheet* in der vollständigen Datei natürlich mehrere Male vorkommt – einmal pro Seite.

Lokales Menü des Formulars: TEXT-DFM bzw. NEUE FORMULARE ALS TEXT in den Umgebungsoptionen unter PRÄFERENZEN.

```
object Form1: TForm1
  ...
  object TabbedNotebook1: TPageControl
    Left = 0
    Top = 0
    ...
    object TTabSheet
      ...
      Caption = '... Tabs'
      object Bevel2: TBevel
        ...
```

Sie können natürlich nur dann den Typ einer Komponente einfach so ändern, wenn der neue Typ alle Properties besitzt, die schon im alten vorhanden waren. Beispielsweise verfügen sowohl *TTabPage* als auch *TTabSheet* über das im obigen Listing erwähnte *Caption*-Property. Allerdings besitzt *TTabbedNotebook* einige Properties, die *TPageControl* nicht kennt, so etwa das im ersten Listing zu sehende Property *TabFont*. Wenn also nur wie vorgeschlagen die beiden Klassen *TTabbedNotebook* und *TTabSheet* ersetzt werden, erhalten wir zunächst eine unkorrekte Formulardefinition.

Trotzdem können wir in diesem Beispiel sofort ANSICHT ALS FORMULAR aus dem lokalen Menü des Editors wählen, um das Formular, dessen »Fundament« gerade ausgetauscht wurde, anzusehen. Kylix stellt nun fest, dass die Formulardatei einige nicht vorhandene Properties enthält (Abbildung 3.7), weigert sich jedoch nicht, das Formular zu laden, sondern bietet Ihnen an, die nicht vorhandenen Properties zu ignorieren. Konkret sind das die *TTabbedNotebook*-Properties *PageIndex*, *TabsPerRow, TabFont.Color*, *TabFont.Height*, *TabFont.Name* und *TabFont.Style*. *Ignorieren* bedeutet, dass die Werte dieser Properties einfach verloren gehen.

Manchmal verfügen Komponenten zwar über die gleichen Einstellmöglichkeiten, bieten diese aber in verschiedenen Properties an: So wird beispielsweise die Registerbeschriftung des PageControls nicht in einem *TabFont*-Property, sondern über das Property *Font* eingestellt. (Der Verlust des *TabFont*-Properties hätte in diesem Fall vermieden werden können, indem wir schon in der Textdarstellung *TabFont* durch *Font* ersetzt hätten).

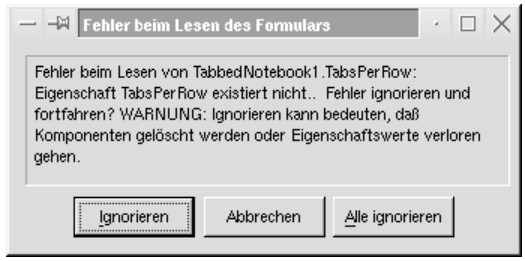


Abbildung 3.8: Eine segensreiche Automatik von Kylix hat sich hier als Fehlermeldung getarnt.

Wenn Sie mehrmals den Schalter IGNORIEREN oder einmal den Schalter ALLE IGNORIEREN drücken, nimmt Kylix alle noch notwendigen Änderungen automatisch vor. Falls dadurch wichtige Properties verloren gehen, können Sie dies später im Objektinspektor korrigieren. So wollen Sie vielleicht auch den *TTabSheet*-Komponenten, deren *TTabPage*-Vorläufer noch unbenannt gewesen sind, Namen geben, um sie im Programm irekt ansprechen zu können.

Hinweis: Auf ähnliche Weise können Sie auch ein *TNotebook* von Delphi für Windows durch ein *TPageControl* ersetzen. Statt *TTabPage* müssen Sie hier nur *TPage* durch *TTabSheet* ersetzen.

3.5.6 Verwenden der Standarddialoge

Da viele Routineaufgaben wie das Auswählen von Dateien, Schriften und Farben in sehr vielen Anwendungen vorkommen, werden auch diese Dialoge häufig in einer allgemeinen Bibliothek definiert, so dass sie dann von mehreren Anwendungen gleichzeitig benutzt werden können. Besonders deutlich ist dies auf der Plattform, von der Kylix einst nach Linux portiert wurde: Unter Windows stellen natürlich die Microsoft-Bibliotheken die Standard-Dialoge zur Verfügung, und da diese so zu einem Teil des Betriebssystems werden, sind diese Dialoge unter Windows-Anwendungen auch extrem weit verbreitet.

Nicht ganz so verbreitet sind die Dialoge der Qt-Bibliothek unter Linux, da dort Qt nur eine von vielen alternativen Bibliotheken darstellt. Allerdings sind die Qt-Dialoge den Dialogen von Windows sehr ähnlich, so dass sie für die Portierung von Windows-Anwendung gerade recht kommen. In Kylix treten diese Dialoge in Form von leicht handhabbaren Komponenten in Erscheinung. Im Gegensatz zu den Steuerelementen sehen Sie diese Dialoge zur Entwurfszeit jedoch nicht so, wie sie zur Laufzeit auftreten. Lediglich ein kleines Icon weist zur Entwurfszeit auf das Vorhandensein eines Standarddialogs hin.

Standarddialog-Übersicht

Abbildung 3.8 zeigt die in Kylix zur Wahl stehenden Dialoge. In der CLX stammen die Klassen der Dialogkomponenten allesamt von der Klasse *TDialog* ab, einem Nachfahren von *TComponent*. Die Dialoge, die sich auf die Qt berufen, haben eine noch speziellere gemeinsame Basisklasse als *TDialog*, nämlich *TQtDialog*. Im Einzelnen sind dies die Klassen *TColorDialog*, *TFontDialog* und *TOpenDialog/TSaveDialog*. Bei den beiden zuletzt genannten ist *TOpenDialog* wiederum die Basisklasse für die *TSaveDialog*, die sich aber kaum von *TOpenDialog* unterscheidet. Ohne Hilfe der Qt-Bibliothek hat Borland die Klassen *TFindDialog/TReplaceDialog* (wieder mit Vererbungsbeziehung) für die Suchen&Ersetzen-Dialoge geschrieben. Die von der VCL unter Windows bekannten

Drucker-Dialoge existieren zwar in Kylix nicht als Komponenten, jedoch können Sie auch hier leicht an einen praktischen Druckdialog gelangen (siehe Kapitel 6.6.3). Für die von der VCL bekannten *TOpenPictureDialog* und *TSavePictureDialog* ist dagegen kein schneller Ersatz möglich.

Unterschiede zwischen Windows- und Qt-Dialogen gibt es auch im Detail. So haben sowohl CLX als auch VCL für die Dateiwahldialoge einige exklusive Optionen im *Options*-Property, in den wichtigsten stimmen sie jedoch überein. Die Windows-Dateiwahl-Dialoge können mit einer Historyliste (*HistoryList*-Property) versehen werden, während die Qt-Dialoge ihre Größe flexibel anpassen lassen (in Kylix über die Properties *Width* und *Height*).

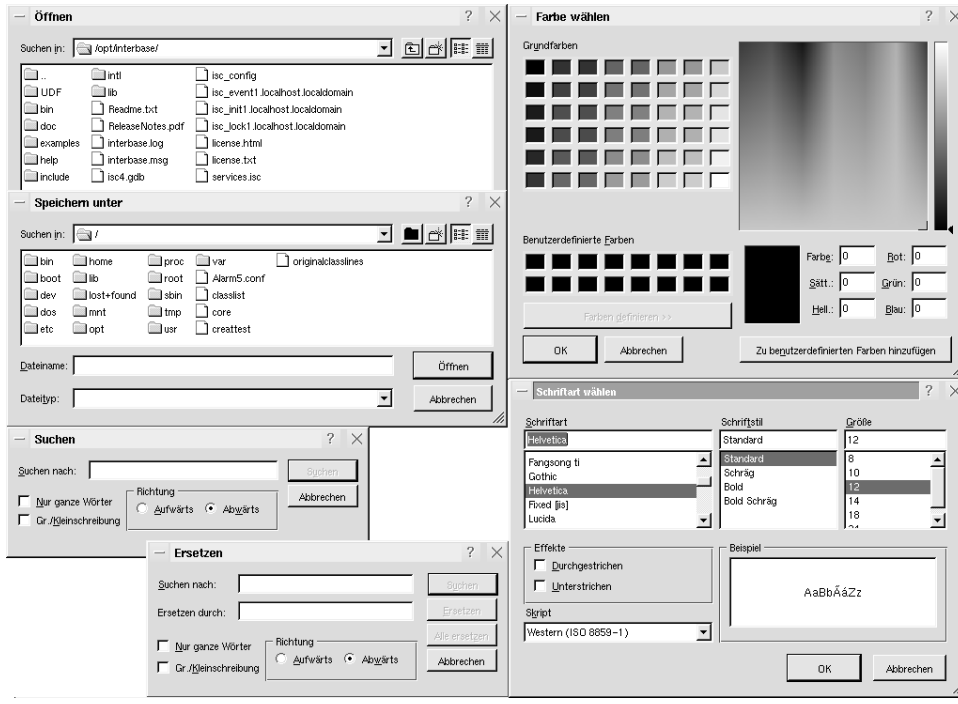


Abbildung 3.9: Die Standarddialoge von Kylix

Grundsätzliche Verwendung

Da die Dialog-Komponente, die zur Entwurfszeit immerhin noch als Icon dargestellt wird, zur Laufzeit überhaupt nicht mehr sichtbar ist, müssen Sie also ein weiteres Steuerelement wie z.B. einen Schalter oder einen Menüpunkt vorsehen, über den der Benutzer den Dialog aufrufen kann. In der Methode, die das von diesem Element aus-

gelöste Befehlsereignis (meistens *OnClick*) bearbeitet, können Sie den Code schreiben, der den Dialog aufruft. Das Minimum besteht hier in der reinen Ausführung des Dialogs, für die Sie wieder einen der typischen Einzeiler benötigen:

```
procedure TForm1.Dateispeichernunter1Click(Sender: TObject);
begin
  SaveDialog1.Execute;
end;
```

Die *Execute*-Methode initialisiert den Standarddialog nach den Werten der Properties, die Sie im Objektinspektor angegeben haben, und zeigt den Dialog an. Wenn der Benutzer ihn mit einem bestätigenden Schalter schließt, speichert *Execute* die Eingaben des Benutzers wieder in den Properties, bevor sie zu Ihrem Programm zurückkehrt.

Meistens gesellen sich zum Aufruf von *Execute* weitere Anweisungen: Vor dem Aufruf von *Execute* können Sie noch weitere Properties der Dialog-Komponente setzen, und nach der Ausführung werden Sie womöglich daran interessiert sein, was der Benutzer eingegeben hat. Außerdem müssen Sie unterscheiden, ob der Benutzer den Dialog per *OK*-Schalter oder mit einem Abbruch beendet hat (das Schließen über das Systemmenü zählt als Abbruch). Dies erfahren Sie vom booleschen Ergebnis, das *Execute* zurückliefert. Der erweiterte Ablauf eines Standarddialog-Aufrufs sieht damit wie folgt aus:

```
procedure OnDialogAufruf(...);
begin
  { optionale Initialisierung von Dialog-Properties }
  ...
  if StandardDialog.Execute = True then begin
     { Ergebnisverwertung, z.B. }
     LoadFile(StandardDialog.FileName);
  end;
  { else: Benutzer hat den Dialog abgebrochen }
end;
```

Eine wichtige Frage muss noch beantwortet werden: Angenommen, der Benutzer hat nun eine Einstellung vorgenommen, die sich katastrophal auf sein Computersystem auswirken würde, und bricht den Dialog mit *Abbruch* ab. Ihr Programm fragt aber nicht das Ergebnis von *Execute* ab und tut so, als hätte der Benutzer *OK* gedrückt. Was passiert dann?

Auch hier sorgen die Komponenten von Kylix für mehr Sicherheit, denn die Properties der Standarddialoge sind völlig von den Werten des tatsächlichen Dialogs getrennt: Wenn der Benutzer *Abbruch* drückt, liest die Dialogkomponente die Eingaben gar nicht aus dem Dialog aus, so dass sich die Werte der Properties durch den *Execute*-Aufruf nicht verändern. Waren also vor dem Dialogaufruf sinnvolle Werte in den Properties, so sind diese auch danach noch vorhanden; es passiert also nichts Schlimmes, um die eben gestellte Frage zu beantworten.

Dateiauswahl-Dialoge

Bei den Dialogen zur Dateiauswahl gibt es unterschiedliche Dialoge zum Speichern und zum Laden von Dateien. Neben minimalen Unterschieden im Äußeren liegen die Differenzen hauptsächlich in den Überprüfungen, die die Dialoge automatisch durchführen können. Wenn Sie z. B. eine nicht existierende Datei eingeben, kann der *Datei-öffnen*-Dialog fragen, ob eine neue Datei erstellt werden soll. Der *Datei-speichern-unter*-Dialog kann Sie zur Sicherheit fragen, ob die Datei überschrieben werden soll, wenn Sie eine schon existierende Datei angeben.

Trotz dieser Unterschiede sind jedoch die Properties der Komponentenklassen *TOpenDialog* und *TSaveDialog* absolut identisch, je nach Typ haben allerdings einige Optionen im Property *Options* keine Wirkung.

Property	Typ	Bedeutung
DefaultExt	String	Endung, die nach Abschluss des Dialogs automatisch an *FileName* angehängt wird, wenn der Benutzer keine Endung angegeben hat.
FileName	String	Pfad und Name der Datei, wird zur Initialisierung des Dialogs und zur Rückgabe des Werts verwendet.
Filter	String	Dieser String enthält die vorgegebenen Filter des Dialogs als eine Folge von sich abwechselnden Filternamen und Dateimasken. Drücken Sie zur Entwurfszeit auf den Schalter neben dem Property-Wert, um einen komfortablen Editor für diese Liste zu erhalten.
FilterIndex	Integer	Index in das *Filter*-Property, der den beim Öffnen des Dialogs verwendeten Filter angibt
InitialDir	String	Hier können Sie ein Verzeichnis angeben, das die Dialogbox als Erstes anzeigt bzw. in das sie wechselt (siehe auch Option *ofNoChangeDir*).
Options	TOpenOptions	siehe folgende Tabelle
Title	String	Titel des Dialogfensters, der den Standardtitel ersetzen soll

Hinzu kommen die Properties *Width* und *Height*, über die Sie die Größe des Dialogs einstellen können. Diese Properties sind bereits in der allgemeinen Klasse *TQtDialog* definiert und daher auch in allen anderen Standarddialog-Komponenten enthalten – sie werden aber bei der weiteren Beschreibung der anderen Dialoge nicht jedes Mal extra erwähnt.

Ausgewählte Optionen

Die vielleicht beliebtesten Optionen, die Sie als Menge im Property *Options* angeben können, sind:

- *ofPathMustExist* ist in den meisten Fällen empfehlenswert: Falls der Benutzer einen neuen Dateinamen angibt, so muss dieser wenigstens einen bestehenden Pfad beinhalten.
- *ofFileMustExist* empfiehlt sich beim Öffnen von Dateien, denn es erlaubt nur die Auswahl bestehender Dateien.
- *ofOverwritePrompt* ist normalerweise für den Speichern-Dialog unbedingt zu empfehlen: Falls die ausgewählte Datei existiert, macht der Dialog eine Sicherheitsabfrage. Wenn Sie diese Option verwenden und das Ergebnis der *Execute*-Methode *True* ist, können Sie davon ausgehen, dass der Anwender sich entweder zustimmend zu dieser Sicherheitsabfrage geäußert oder eine neue Datei angegeben hat (diese Option ist nur beim Speichern-Dialog gültig).

Die zahlreichen weiteren Optionen betreffen unter anderem die Mehrfachauswahl, die Größenveränderlichkeit der Dialogbox, die Anzeige versteckter Dateien, den Wechsel des Verzeichnisses und die Anzeige von Details zu den Dateien. Die CLX-Online-Hilfe gibt darüber weitere Auskunft.

Beispiel für Dateidialog-Einstellungen

Bei den Dateiauswahl-Dialogen des TreeDesigners (Kapitel 6) wurden nur jeweils drei Properties geändert. So sieht beispielsweise der *Datei-öffnen*-Dialog in der Formulardatei wie folgt aus:

```
object OpenDialog1: TOpenDialog
  DefaultExt = 'TVF'
  Filter = 'Tree Viewer Files|*.TVF'
  Options = [ofPathMustExist, ofFileMustExist]
  Left = 152
  Top = 141
end
```

Die *Filter*-Einstellung sorgt (in Zusammenhang mit dem voreingestellten *FilterIndex* von »1«) dafür, dass der Dialog gleich die Dateien *.TVF anzeigt. *DefaultExt* stellt sicher, dass der Dialog die Endung automatisch anhängt, wenn der Benutzer vergessen sollte, sie einzugeben. Auch die beiden in den Optionen angegebenen Flags ersparen dem TreeDesigner einige Überprüfungen. Im *Speichern-unter*-Dialog wurden diese beiden Flags durch die Option *ofOverwritePrompt* ersetzt, ansonsten stimmen die wesentlichen Properties beider Dialoge überein.

Der Farbauswahl-Dialog

Aufgabe des *TColorDialogs* ist es, einen Wert des Typs *TColor* zum Editieren bereitzustellen. Sie können die anfangs angezeigte Farbe im Property *Color* bestimmen, aus dem Sie auch die Ergebnisfarbe auslesen können. Eine Liste von vordefinierten Farben,

die der Farbdialog im Feld *Selbst definierte Farben* anzeigt, können Sie als String im Property *CustomColors* angeben. Ein Beispiel zum Format dieses Strings finden Sie im Zusammenhang mit der Farbpaletten-Komponente in Kapitel 8.6.3.

Die von Delphi für Windows bekannten Optionen entfallen in der CLX, was jedoch nicht als besonderer Verlust zu gelten braucht: Der Qt-Farbdialog erspart dem Benutzer die beim Windows-Farbdialog erforderliche Klickerei, um den Dialog zur Definition eigener Farben aufzuklappen, und präsentiert sich immer gleich in voller Farbenpracht. Und damit erübrigen sich bereits fast alle vom *TColorDialog* der VCL bekannten Optionen.

Der Dialog zur Schriftauswahl

Im Schriftauswahl-Dialog dreht sich alles um das Property *Font*, das den Typ *TFont* besitzt. Sie bedienen dieses Property wie die Properties *Color* und *FileName* der bisher gesehenen Dialoge. In den Beispielprogrammen von Kapitel 1 wurde dieser Dialog bereits zur Einstellung der Schrift eines Memo-Feldes und einer Uhrzeit-Anzeige herangezogen. Die besonderen Optionen des *TFontDialog* der VCL entfallen bei der CLX – auch dies dürfte in den meisten Fällen nicht auffallen, leistet der Dialog doch auch so gute Dienste.

Dialoge für die Textsuche

Aus technischer Sicht sind die beiden Dialoge zum Suchen bzw. zum Suchen und Ersetzen interessant, denn *TReplaceDialog* ist von *TFindDialog* abgeleitet. Der *Ersetzen*-Dialog ist eine Erweiterung des *Suchen*-Dialogs, so dass die Klasse *TReplaceDialog* einiges von *TFindDialog* erben kann.

Ein weiter Aspekt, der bei keinem anderen Standarddialog zu finden ist, ist die Ereignissteuerung: Beide Dialoge können Ereignisse erzeugen, zu denen Sie Bearbeitungsmethoden schreiben müssen, wenn Sie das übliche Verhalten der Dialoge sicherstellen wollen: Drückt der Benutzer den Schalter *Suchen* oder *Ersetzen*, so soll jeweils das nächste Vorkommnis des Suchtextes gesucht oder ersetzt werden. Den Code zum Suchen müssen Sie selbst zur Verfügung stellen, und zwar müssen Sie ihn mit den *OnFind-* bzw. *OnReplace*-Ereignissen verknüpfen. Diese Ereignisse werden aufgerufen, während der Dialog läuft, also noch bevor der Aufruf der *Execute*-Methode beendet ist.

Welchen Text der Benutzer suchen und ersetzen will, erfahren Sie aus den Dialog-Properties *FindText* und *ReplaceText*; die Optionen, die er eingestellt hat (Groß- und Kleinschreibung, ganze Wörter etc.), befinden sich wieder im Property *Options*, das noch viele weitere Elemente enthält, die alle von Ihrer Suchmethode berücksichtigt werden könnten und die hier aufgrund ihrer selbst erklärenden Namen nur kurz zur Übersicht aufgelistet werden sollen:

```
TFindOption = (frDown, frFindNext, frHideMatchCase, frHideWholeWord,
    frHideUpDown, frMatchCase, frDisableMatchCase, frDisableUpDown,
    frDisableWholeWord, frReplace, frReplaceAll, frWholeWord,
    frShowHelp);
```

3.6 Komplexere Steuerelemente

Dieses Kapitel behandelt in Ergänzung zu den Kapiteln 1.8 (Überblick und Vorstellung einfacher Komponenten) und 3.3 (Gemeinsamkeiten aller Komponenten) die folgenden Komponenten:

- die Editierkomponenten *TEdit* und *TMemo* (Kapitel 3.6.1)
- die beiden sich sehr gut ergänzenden Komponenten *TListView* und *TIconView* (Kapitel 3.6.2)
- die Komponente *TTreeView* zur Anzeige hierarchischer Strukturen (Kapitel 3.6.3).

3.6.1 Editierfelder und Memos

Auf der *Standard*-Seite der Komponentenpalette finden Sie zwei verschiedene Arten von Editierfeldern: einzeilige *Edit*-Komponenten und mehrzeilige *Memos*. Beide haben viele übereinstimmende Eigenschaften, sind aber dennoch nicht wie in der VCL von einer gemeinsamen Basisklasse *TCustomEdit* abgeleitet. Für die Verwendung der Komponenten spielt dieses interne Detail jedoch keine Rolle. Im Folgenden werden zunächst die Funktionen von *TEdit* erläutert, darauf aufbauend danach die Funktionen von *TMemo*.

extinhalt

Das Wichtigste an einem Editierfeld jeglicher Art ist sicher der Text selbst. Über das Property *Text* erhalten Sie Zugriff auf diesen Text; Memofelder geben Ihnen über das Property *Lines* außerdem Zugriff auf einzelne Zeilen.

Markierter Text

Die vielleicht zweitwichtigste Information, die Sie einer jeden Editierkomponente entnehmen können, ist, welcher Text selektiert ist. Dabei kann grundsätzlich nur ein durchgängiger Bereich des Textes markiert sein, der ab einem bestimmten Zeichen beginnt und eine bestimmte Länge hat. Der Index des ersten markierten Zeichens steht dabei in dem Property *SelStart*, die Zahl der markierten Zeichen finden Sie im Property *SelLength*. Das letzte markierte Zeichen hat also den Index *SelStart+SelLength-1*.

Wenn Sie jedoch den selektierten Text auslesen wollen, brauchen Sie nicht mit diesen Indizes im Textpuffer der Komponente herumzusuchen, denn das Property *SelText* gibt direkten Zugriff auf den ausgewählten Text. Die wichtigen *TEdit*-Elemente für den selektierten Text zusammengefasst sind:

Elementart	Name	Typ bzw. Parameter	Bedeutung
property	SelStart	Integer	Index des ersten markierten Zeichens; falls nichts markiert ist: Index der Cursorposition
property	SelLength	Integer	Zahl der markierten Zeichen.
property	SelText	String	enthält den markierten Text.
procedure	SelectAll	-	selektiert den gesamten Text.

Eine Besonderheit von *SelStart* ist, dass dieses Property auch dann sinnvolle Werte enthält, wenn kein Text markiert ist, wenn also *SelLength* Null ist. *SelStart* enthält in diesem Fall den Index des Zeichens, das sich gerade an der Position der Eingabemarkierung befindet.

Editierfeld-Properties

Weitere von *TEdit* deklarierte Properties sind:

Property	Typ	Bedeutung
AutoSelect	Boolean	Wenn *True*, wird der gesamte Text selektiert, sobald der Benutzer mit der Tabulatortaste zu diesem Editierfeld springt (so, dass er den Text sofort überschreiben kann).
AutoSize	Boolean	automatische Größenanpassung des Editierfelds an die Schriftgröße (und nicht an den Text selbst, wie bei *TLabel*)
CharCase	(ecNormal, ecUpperCase, ecLowerCase)	bestimmt, ob die Zeichen automatisch in Groß- oder Kleinbuchstaben umgewandelt werden.
HideSelection	Boolean	Wenn *True*, wird die Markierung des Textes nur angezeigt, wenn das Editierfeld den Tastaturfokus hat.
MaxLength	Integer	gibt die maximale Länge des Textes an; bei Null gilt eine systembedingte Längenbeschränkung.
Modified	Boolean	*True*, wenn Text geändert wurde, nachdem die Methode *ClearModify* das letzte Mal aufgerufen wurde.
ReadOnly	Boolean	verhindert Veränderungen am Text, wenn *True*

Bei einzeiligen Editierfeldern der Klasse *TEdit* benötigen Sie häufig nur das Property *Text*, das den Feldinhalt speichert, und manchmal das Ereignis *OnChange*, um sofort auf jede Änderung reagieren zu können.

Bearbeitungs- und Zwischenablage-Methoden

Ein weiterer Funktionsbereich von *TEdit* beschäftigt sich mit der Zwischenablage und anderen Funktionen, die häufig in einem BEARBEITEN-Menü zur Verfügung stehen. Hierfür sind ausschließlich Methoden zuständig:

Methode	Bedeutung
Clear	löscht den gesamten Text.
ClearSelection	löscht den ausgewählten Text.
CopyToClipboard	kopiert den Text in die Zwischenablage und wird normalerweise in einer Methode aufgerufen, die das *OnClick*-Ereignis eines BEARBEITEN | KOPIEREN-Menüpunkts bearbeitet.
CutToClipboard	entspricht dem Menüpunkt AUSSCHNEIDEN (=*CopyToClipBoard+ClearSelection*).
PasteFromClipboard	entspricht dem Menüpunkt EINFÜGEN – fügt den Inhalt der Zwischenablage ein und markiert den eingefügten Text (bisher markierter Text wird überschrieben).

TMemo

TMemo stimmt in Teilbereichen mit *TEdit* überein. Die oben bereits aufgelisteten Elemente für selektierten Text und für Zwischenablage-Operationen finden Sie auch bei *TMemo* wieder, sogar noch ein klein wenig ergänzt: Mit *Unselect* können Sie die Auswahl des Textes auch zurücknehmen und in *Selection* erhalten Sie die Angaben über den gewählten Text in Form von Zeilen- und Spaltenangaben (statt einfach als absoluten Index in den gesamten Textinhalt). Die folgende Tabelle fasst weitere Properties von *TMemo* zusammen, die gegenüber *TEdit* neu sind:

Property	Typ/Werte	Bedeutung
Lines	TStrings	Alternative zum Property *Text*, die Zugriff auf einzelne Zeilen des Textes bietet.
Alignment	taLeftJustify, taRightJustify, taCenter	gibt die Ausrichtung der Zeilen innerhalb des Memos an.
ScrollBars	ssHorizontal, ssVertical, ssBoth, ssNone	gibt der Komponente eine horizontale oder vertikale Bildlaufleiste oder es schaltet beide Bildlaufleisten ab (*ssNone*).
WantReturns	Boolean	Solange dieses Property bei seiner Voreinstellung *True* bleibt, bearbeitet und »verschluckt« das Editierfeld die Eingabetaste. Bei *False* geht die Eingabetaste an den Dialog, der dann einen Klick auf den Standard-Befehlsschalter simuliert. Anstelle der Eingabetaste können Sie dann im Editierfeld die Tastenkombination Strg+↵ verwenden.

Property	Typ/Werte	Bedeutung
WantTabs	Boolean	wie *WantReturns*, bezieht sich jedoch auf die Tabulatortaste. Außerdem gibt es innerhalb des Editierfelds keinen Ersatz für [⇆], wenn *WantTabs* auf *False* gesetzt ist.
WordWrap	Boolean	Bei *True* nimmt das Editierfeld automatisch einen Zeilenumbruch am Ende der Zeile vor.

3.6.2 ListViews, IconViews und Bilderlisten

In Kapitel 1.8.5 wurde bereits anhand von *AlarmClock5* der praktische Einsatz eines ListViews und eines IconViews demonstriert. Dieses Kapitel befasst sich etwas tiefer gehend mit diesen Komponenten, geht dabei auch auf die von beiden verwendete Klasse *TImageList* ein und zeigt, wie Sie ein ListView und ein IconView kombinieren können, um damit die vier von der VCL-Klasse *TListView* unter Delphi für Windows gewohnten Ansichten zu erhalten. Hierbei dienen die Programme *ListViewIconView* (auf der CD im Verzeichnis *Kapitel3*) und *AlarmClock5* (CD-Verzeichnis: *Kapitel1*) als Beispiel.

Sowohl *TListView* als auch *TIconView* zeigen einer Menge von Elementen, die jeweils im Property *Items* gespeichert werden und erwartungsgemäß über Methoden wie *Items.Add* und *Items.Delete* manipuliert werden können. Die wichtigsten Eigenschaften eines Elements in *Items* sind seine Beschriftung und sein Bildsymbol. Beides können Sie bereits zur Entwurfszeit festlegen, wenn Sie im Objektinspektor den Property-Editor für *Items* aufrufen. Die Festlegung dieser Daten während der Laufzeit soll später am Beispielprogramm demonstriert werden.

Alle Bildsymbole innerhalb von ListView und IconView werden dort über einen Index in eine getrennte Bilderliste der Klasse *TImageList* angegeben. Die Verknüpfung zwischen ListView/IconView und Bilderliste wird über Property *TListView/TIconView.Images* hergestellt.

Hinweis für VCL-Kenner: Im Vergleich zur *TListView*-Klasse der VCL unter Windows ist die Situation in der CLX recht einfach, da jeweils nur eine Bilderliste mit dem ListView bzw. IconView verknüpft sein kann. Getrennte Bilderlisten für Status- und Overlay-Bilder entfallen, da diese von den CLX-Komponenten (bzw. den zugrunde liegenden Qt-Widgets) gar nicht unterstützt werden. Selbst eine Unterscheidung zwischen kleinen und großen Bilder (*SmallImages* und *LargeImages* in der VCL) entfällt, doch können Sie zwischen kleinen und großen Bildern jederzeit wechseln, indem Sie einfach eine andere Bilderliste mit Ihrem ListView bzw. IconView verknüpfen.

Hinweis: Über eine »visuelle Formularverknüpfung« genannte Funktion können Sie von mehreren Formularen auf dieselbe Bilderliste zugreifen. Das Formular, das die Bilderliste enthält, muss dazu in der *uses*-Anweisung der Unit aufgenommen werden, die auf die Bilderliste zugreifen will. Der Objektinspektor zeigt dann für die *ImageList*-Properties in dieser Unit auch die Bilderlisten des anderen Formulars an. Auch die in Kapitel 7.2.1 besprochenen Datenmodule eignen sich zur Aufnahme von Bilderlisten, die in mehreren Formularen benötigt werden.

TImageList zur Entwurfszeit

Wenn Sie ein Exemplar der Klasse *TImageList* in das Formular einfügen, müssen Sie die Größe der Bilder in den Properties *Height* und *Width* einstellen, bevor Sie die Bilder hinzufügen, denn ein nachträgliches Ändern von *Height* und *Width* führt zur Löschung aller bisher eingefügten Bilder. Eine Verwendung von verschieden großen Bildern ist nicht möglich, da die Bilderlisten zur effektiven Verwaltung großer Bilderzahlen auf Gleichförmigkeit aufbauen.

Nach der Größeneinstellung können Sie aus dem Kontextmenü der *TImageList*-Komponente den *Bildlisten-Editor* aufrufen (der Objektinspektor bietet keinen Zugang zu diesem Editor) und, wenn auch auf eine etwas umständliche Weise, einzelne Dateien in die Bilderliste laden. Zu den von der CLX akzeptierten Dateiformaten siehe Kapitel 1.4.9.

TListView zur Entwurfszeit

Sie können eine *TListView*-Komponente schon vor dem Programmstart dialoggesteuert mit Inhalt füllen, allerdings sind die hierzu verwendeten Dialoge sehr umständlich in der Handhabung und in der Praxis werden die ListView-Inhalte sowieso meistens erst zur Laufzeit aufgebaut. Um einen Überblick über die *TListView*-Strukturen zu gewinnen, eignet sich diese Vorgehensweise aber allemal: Rufen Sie im Objektinspektor den Editor zum Property *Items* auf. Sie erhalten ein Dialogfenster, in dem Sie zu jedem Eintrag neben dem obligatorischen Text auch schon zwei der erwähnten Bilder-Indizes eintragen können.

Der Schalter NEUER UNTEREINTRAG dient dazu, den Text festzulegen, der im Ansichtsmodus *vsReport* in den weiteren Spalten dargestellt wird. Damit die Komponente in diesem Modus überhaupt über mehrere Spalten verfügt, müssen Sie jedoch zuerst im Property *Columns* diese Spalten definieren. Der Property-Editor zu *TListView.Columns* wurde schon in Kapitel 1.8.5 verwendet, um die Spalten für die Alarmereignisse des dortigen Beispielprogramms zu definieren.

TListItems und TListItem

TListView richtet sich wie alle anderen komplexeren Komponenten der CLX nach dem Prinzip, dass die Daten, die in der Komponente dargestellt werden, in einer eigenen Klasse zusammengefasst werden. Was also die Klasse *TStrings* für das *Items*-Property der Komponente *TListBox* ist, ist die Klasse *TListItems* für *TListView.Items*.

Da *TListItems* große Ähnlichkeit zu *TStringList* aufweist, ist das *Items*-Property von *TListView* im Quelltext fast noch einfacher zu verwenden als im Property-Editor. Im Wesentlichen besteht es aus einem Array von *TListItem*-Objekten, von denen jedes einzelne einen Listeneintrag repräsentiert und unter anderem den Beschriftungstext sowie die Nummer des Bildes in den Bilderlisten enthält. Die einzelnen *TListItem*-Objekte sind über das indizierte Property *TListItems.Item* ansprechbar, das als Standard-Array-Property definiert ist, so dass Sie ein Listenelement auf zwei verschiedene Arten ansprechen können. Zum Auswählen des Elements an der Position *Index* können Sie beispielsweise schreiben:

```
  ListView.Items.Item[Index].Selected := True;
// und einfacher:
  ListView.Items[Index].Selected := True;
```

TIconView

Die bisher für *TListView* erläuterten Dinge treffen auf eine vereinfachte Weise auch auf *TIconView* zu: Schon zur Entwurfszeit können Sie mit einem speziellen Property-Editor Einträge hinzufügen und mit Text und Bild versehen. Die Möglichkeit, jeden Eintrag mit mehreren Feldern von Information zu versehen (»Untereinträge«), entfällt.

Auch bei *TIconView* heißt das Property, in dem der eigentliche Inhalt gespeichert wird, *Items*. Die Ansteuerung dieses Properties zur Laufzeit entspricht dem von *TListView* gewohnten, das Beispielprogramm wird hierzu auch ein konkretes Code-Beipiel beisteuern.

Beispielprogramm ListViewIconView

Anhand des Beispielprogramms *ListViewIconView* (Abbildung 3.9) wenden wir uns nun der Funktion des ListViews zur Laufzeit zu.

Das ListView des Programms wird zur Laufzeit dynamisch mit Inhalt gefüllt, erlaubt die Auswahl mehrerer Einträge gleichzeitig (*MultiSelect = True*) und reagiert auf die Löschtaste, indem es die gewählten Einträge entfernt. Alternativ können Sie Einträge auch per Drag&Drop löschen, indem Sie sie auf den »Papierkorb« auf der linken Seite des Programmfensters ziehen.

Hier auf der linken Seite finden Sie auch noch einige Markierungsfelder, mit denen Sie die Properties *RowSelect*, *CheckBoxes*, *AutoArrange*, *Arrangement* und *TextPosition* beeinflussen können. Für Details zu diesen Properties sei auf die Online-Hilfe verwiesen.

Eine wichtige Funktion von *ListViews* ist das Editieren der Einträge. Sie ist zwar auch im Beispielprogramm vorhanden, erfüllt dort aber keinen besonderen Zweck. Für eine sinnvolle Verwendung der Editierfunktion sei daher auf das Programm *AlarmClock5* in Kapitel 1.8.5 verwiesen.

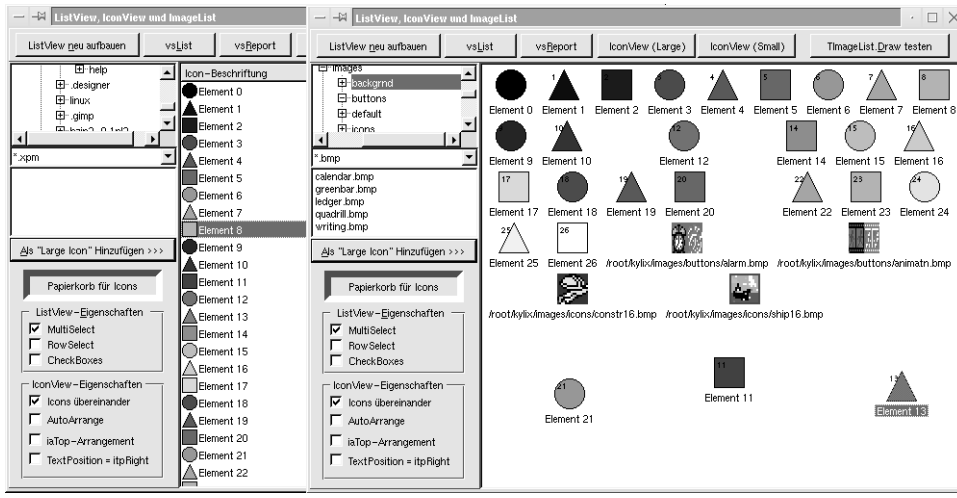


Abbildung 3.10: Dieses Programm implementiert zahlreiche ListView-Funktionen und baut den ListView-Inhalt dynamisch auf. Im Fenster rechts übernimmt anstelle des ListViews ein IconView die Darstellung.

Behandeln der gewählten Einträge am Beispiel des Löschens

Im Gegensatz zum Editieren der Beschriftung können Sie sich für das Löschen von Einträgen nicht auf eine Automatik des ListViews verlassen. Das Beispielprogramm demonstriert anhand des Löschens, wie Sie alle im ListView markierten Einträge abfragen und bearbeiten. Dabei kommt das schon erwähnte *Items*-Array zum Einsatz:

```
procedure TMainForm.DeleteSelection;
var
  i: Integer;
begin
  i := 0;
  while i < ListView.Items.Count do begin // alle Einträge durchlaufen
    if ListView.Items[i].Selected then  // wenn markiert
      ListView.Items.Delete(i) // löschen
    else inc(i); // (Index nicht erhöhen, wenn Eintrag gelöscht wurde)
  end;
end;
```

Die gezeigte Methode wird im Beispielprogramm bei zwei Gelegenheiten aufgerufen: Wenn der Benutzer Einträge des ListViews per Drag&Drop in den als »Papierkorb« bezeichneten Fensterbereich zieht (dies findet im Rahmen der Drag&Drop-Funktionalität statt) und wenn er die Löschtaste drückt. Letzteres wird beim Ereignis *OnKeyDown* des ListViews abgefragt:

```
procedure TMainForm.ListViewKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = Key_Delete then
    DeleteListViewSelection;
end;
```

Der fokussierte Eintrag

Wenn Sie in einem ListView die Mehrfachauswahl zulassen, gilt es zwischen einem selektierten und einem *fokussierten* Eintrag zu unterscheiden. Während mehrere Einträge selektiert sein können, ist immer nur ein Eintrag fokussiert, und zwar derjenige, der zuletzt angeklickt wurde. Zwar hat jeder Eintrag der *Items*-Liste auch ein *Focused*-Property; statt wie oben in einer Schleife können Sie den fokussierten Eintrag jedoch ganz schnell über das *FocusItem*-Property des ListViews herausfinden. Die folgende Anweisung beispielsweise löscht nur den fokussierten Eintrag. Auch sie benötigt einen Index als Parameter für *Delete*, den sie vom *Index*-Property des fokussierten Eintrages erfährt:

```
ListView.Items.Delete(ListView.ItemFocused.Index);
```

Einträge dynamisch hinzufügen

Um zur Laufzeit einen Eintrag zu einem ListView hinzuzufügen, verwenden Sie die Methode *Items.Add*, die ein neues *TListItem*-Objekt erzeugt, dieses an die Liste anhängt und einen Verweis auf dieses Objekt als Funktionsergebnis zurückliefert. Wie schon erwähnt, hat dieses Objekt die Klasse *TListItem*. Im Anschluss an den Aufruf von *Add* legen Sie die verschiedenen Daten des Eintrags fest, indem Sie die Properties des *TListItems* setzen. Das Beispielprogramm erzeugt beispielsweise alle Einträge in einer Schleife, wobei es für jeden Eintrag ein anderes Bild verwendet (anders ausgedrückt erzeugt es für jedes Bild der Bilderliste einen Eintrag):

```
const
  ElementCount = 27;
var
  NewItem: TListItem;
  iItem: Integer;
begin
  ListView.Items.Clear;
  for iItem := 0 to ElementCount-1 do begin
    NewItem := ListView.Items.Add;
```

```
    // Eintrag beschriften mit dem Text "Eintrag [Nummer]":
    NewItem.Caption := 'Element '+(IntToStr(iItem));
    NewItem.ImageIndex := iItem; // mit Bild verknüpfen
    // die weiteren Spalten erzeugen (das Beispielprogramm
    // verwendet einen anderen Text):
    NewItem.SubItems.Add('Text für erste Spalte');
    NewItem.SubItems.Add('Text für zweite Spalte');
    ...
  end;
```

Nachdem sie wie oben gezeigt mit *Add* hinzugefügt wurden, lassen sich die Textspalten des *SubItems*-Properties übrigens ebenso leicht ansprechen wie die Haupteinträge des ListViews, denn bei *SubItems* handelt es sich einfach um ein Stringlistenobjekt der weit verbreiteten *TStrings*-Klasse:

```
ListView.Items[ElementIndex].SubItems[SpaltenIndex] := 'Text';
```

Programmierung des IconViews *R54*

Die Programmierung des IconViews ist der des ListViews sehr ähnlich. Im Beispielprogramm wird das IconView ebenso wie das ListView mit 27 Einträgen gefüllt, von denen jeder ein anderes Bild aus der Bilderliste zeigt:

```
with IconView.Items do begin
  Clear;
  for iItem:=0 to ElementCount-1 do begin
    NewIconViewItem := Add;
    NewIconViewItem.Caption := 'Element '+(IntToStr(iItem));
    NewIconViewItem.ImageIndex := iItem;
  end;
end;
```

Der größte Unterschied zum entsprechenden Listing von *TListView* dürfte hier darin bestehen, dass eine *with*-Anweisung verwendet wurde, um mit weniger Schreibarbeit auf *IconView.Items* zugreifen zu können.

Des Weiteren bearbeitet das Beispielprogramm auch das *OnKeyDown*-Ereignis des IconViews, um die Löschtaste abzufragen und die gewählten Elemente des IconViews zu löschen. Dies geht genau wie das Löschen der Einträge des ListViews, was oben schon anhand eines Listings gezeigt wurde.

Eine besondere Möglichkeit, die es im ListView nicht gibt, ist das freie Verschieben von Einträgen mit der Maus. Das Beispielprogramm zeigt, wie Sie dieses durch Bearbeiten dreier Ereignisse realisieren können: Zunächst stellen Sie in *OnMouseDown* fest, ob ein Eintrag angeklickt wurde. Dies können Sie mit der Methode *TIconView.FindItemByPoint* feststellen, der Sie die Mauskoordinaten übergeben. Das Beispielprogramm speichert einen eventuell angeklickten Eintrag für die spätere Verwendung in der Variablen

DraggedItem. Die Position der Maus relativ zur linken oberen Ecke des angeklickten Eintrags wird in den Variablen *XOfs/YOfs* gespeichert. Dies ist wichtig, um den Eintrag beim Loslassen der Maustaste exakt positionieren zu können:

```
procedure TMainForm.IconViewMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button=mbLeft then begin
    DraggedItem := IconView.FindItemByPoint(Point(X, Y));
    if Assigned(DraggedItem) then begin
      XOfs:=x-DraggedItem.Left;
      YOfs:=y-DraggedItem.Top;
    end;
  end;
end;
```

Alles weitere ist reines Drag&Drop. Details zu diesem Thema finden Sie in Kapitel 6.8.3. Drag&Drop innerhalb des ListViews wird aktiviert, indem Sie sein *DragMode*-Property auf *dmAutomatic* stellen. Dann gilt es noch, die Ereignisse *OnDragOver* und *OnDragDrop* zu bearbeiten.

Die Besonderheit des Beispielprogramms liegt darin, dass es, wenn der entsprechende Schalter am linken Rand nicht markiert ist, keine überlappenden Icons zulässt. Dies geht nur, indem beim *OnDragOver*-Ereignis geprüft wird, ob sich an der Mausposition bereits ein Icon befindet. Ist das der Fall, setzt die Methode den *Accept*-Parameter auf *False*, so dass der Benutzer das Icon an dieser Stelle nicht ablegen kann:

```
procedure TMainForm.IconViewDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source = IconView then begin
    Accept := IconOverlap.Checked  or
      (IconView.FindItemByPoint(Point(x, y))=nil);
  end;
end;
```

Das eigentliche Verschieben des Eintrags findet dann in der Methode für *OnDragDrop* statt. Die Methode *Move* des verschobenen Icons, welches oben in der Variablen *DraggedItem* zwischengespeichert wurde, sorgt für die neue Positionierung:

```
procedure TMainForm.IconViewDragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  if Source=IconView then begin
    if Assigned(DraggedItem) then begin
      DraggedItem.Move(Point(x-XOfs, y-YOfs));
    end;
  end;
end;
```

Datenobjekte im IconView speichern *R57*

Ein wesentlicher Unterschied zwischen ListView und IconView ist in den obigen Listings noch nicht sichtbar geworden: *TIconView* verwendet eine grundlegend andere Art und Weise, um benutzerdefinierte Datenobjekte mit den einzelnen Einträgen zu verknüpfen. Während Sie in *TListView* an *ListView.Items[index].Data* ein beliebiges Objekt mit dem *Item[index]* verbinden können, verfügt die Klasse *TIconViewItem* gar nicht über ein *Data*-Property, *IconView.Items[index].Data* existiert also nicht.

Um einen *IconViewItem* mit selbst definierten Daten zu versehen, müssen Sie eine eigene Klasse von *TIconViewItem* ableiten, darin die gewünschten Daten definieren und dem IconView mitteilen, dass es statt der vordefinierten Klasse *TIconViewItem* Ihre erweiterte Klasse verwenden soll.

Ein Beispiel dafür gibt das Programm *AlarmClock5* aus Kapitel 1.8.5. In diesem wurde bereits jeder ListView-Eintrag mit einem *TAlarm*-Objekt verknüpft (über das bekannte *Data*-Property). Dieses *TAlarm*-Objekt enthält alle Daten, die auch mit dem IconView-Einträgen verknüpft werden sollen. Also wird *TIconViewItem* wie folgt erweitert:

```
type
  TAlarmIcon = class(TIconViewItem)
    AlarmData: TAlarm;
  end;
```

Wenn es in *AlarmClock5* nicht notwendig wäre, die bestehende *TAlarm*-Klasse weiter zu verwenden, könnten deren Datenelemente natürlich auch direkt in *TAlarmIcon* gespeichert werden, die Deklaration sähe dann ungefähr so aus:

```
type
  TAlarmIcon = class(TIconViewItem)
    Alarmzeit: TDateTime;
    Kommentar: String[255];
    ...
  end;
```

Entscheidend ist jedoch hier nicht die Struktur der von *TIconViewItem* abgeleiteten Klasse, diese können Sie beliebig selbst bestimmen. Worauf es ankommt ist, dass die bereits bekannte Methode *TIconView.Items.Add* ein Objekt der selbst definierten Klasse rzeugt. Um dies zu bewirken, rufen Sie vor der *Add*-Methode die Methode *Iconiew.Items.SetItemClass* auf und übergeben ihr die selbst definierte Klasse. Nach dem Aufruf von *SetItemClass* funktioniert alles wie gehabt, nur die von *Add* erzeugten Objekte lassen sich nun mit dem *as*-Operator in ein Objekt der selbst definierten Klasse, im Beispiel also in ein *TAlarmIcon*-Objekt, umwandeln.

In *AlarmClock5* finden diese Vorgänge in einer Methode der Klasse *TAlarm* statt, die ein neues IconView-Icon erzeugt. Da die selbst definierte Klasse *TAlarmIcon* ja von *TIconiewItem* erbt, sind darin auch die bereits bekannten Properties *Caption* und *ImageIndex*

enthalten. *Caption* wird in *AlarmClock5* auf die vom Benutzer eingestellte Alarmzeit gesetzt. Wie der zu den Alarmdaten passende *ImageIndex* ermittelt wird, ist ein spezielles Thema, das an dieser Stelle nicht weiter interessieren soll:

```
procedure TAlarm.AddToIconView(iv: TIconView);
var
  NewItem: TAlarmIcon;
begin
  iv.Items.SetItemClass(TAlarmIcon);
  NewItem := iv.Items.Add as TAlarmIcon;
  // Speichern der selbst definierten Daten im IconView:
  NewItem.AlarmData := self;
  NewItem.Caption := TimeToStr(Alarmzeit);
  NewItem.ImageIndex := GetImageIndex;
end;
```

Bug-Fix: Die erste CLX-Version enthält hier einen Fehler, der den korrekten Ablauf des obigen Programms verhindert. Und zwar ignoriert die Methode *TListItems.Add* die von *TListItems.SetItemClass* gespeicherte Klasse *FIconViewItemClass* und erzeugt immer ein neues Objekt der Klasse *TIconViewItem*. Dieser Fehler lässt sich durch Änderung einer Zeile beheben, Genaueres dazu sagt Ihnen die Readme-Datei auf der CD.

Hinweis zur Speicherverwaltung**: *TIconView*** sorgt automatisch für die Freigabe der Items, auch wenn diese eine selbst definierte Klasse haben. *TIconViewItem* definiert einen virtuellen Destruktor, so dass Sie mit einem überschriebenen Destruktor eventuelle eigene dynamische Speicherreservierungen rückgängig machen können. Im Fall von *AlarmClock5* ist das nicht nötig, da der einzige zusätzliche Inhalt von *TAlarmIcon* eine Referenz auf ein *TAlarm*-Objekt ist, welches an anderer Stelle freigegeben wird. Es sei in diesem Zusammenhang noch einmal daran erinnert, dass die Variable *NewItem.AlarmData* wie alle Objekte in Object Pascal intern lediglich ein Zeiger auf ein dynamisch im Speicher angelegtes Objekt ist. Die Zuweisung *NewItem.AlarmData := self* erzeugt also kein Duplikat des *self*-Objekts, sondern setzt intern nur einen Zeiger auf das schon bestehende Objekt.

IconView und ListView kombinieren *R32*

Wie schon das Programm *AlarmClock5* aus Kapitel 1.8.5 bietet auch das Beispielprogramm dieses Kapitels vier verschiedene Ansichten, indem es die beiden Ansichten des ListViews mit zwei IconView-Ansichten kombiniert. Die in Kapitel 1.8.5 noch aufgeschobene Beschreibung, wie dies im Detail funktioniert, soll an dieser Stelle nachgeholt werden. Es ist auf diese Weise auch möglich, alle vier Ansichtsarten eines Windows-ListViews nach Linux zu portieren.

Dafür muss zunächst eine Möglichkeit geschaffen werden, zwischen ListView und IconView umzuschalten. Immer wenn eine Ansicht der einen Komponente aktiv ist, soll die andere Komponente versteckt werden. Da immer nur eine von beiden sichtbar ist, können beide Komponenten im Prinzip dieselbe Fläche einnehmen. Dies ist jedoch zur Entwurfszeit eine Schwierigkeit, da auch beide Komponenten mit *alClient* ausgerichtet werden müssen, um den verfügbaren Platz optimal auszunutzen. Im Beispielprogramm wurde wird die Ausrichtung *alClient* daher erst zur Laufzeit, wenn das Formular erzeugt wird, vorgenommen. Abbildung 3.10 zeigt das Formular zur Entwurfszeit und zur Laufzeit. Nachfolgend die Methode für das *OnCreate*-Ereignis, die dafür sorgt, dass nur noch das *ListView* sichtbar ist:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  IconView.Align := alClient;
  IconView.Visible := False;
  ListView.Align := alClient;
end;
```

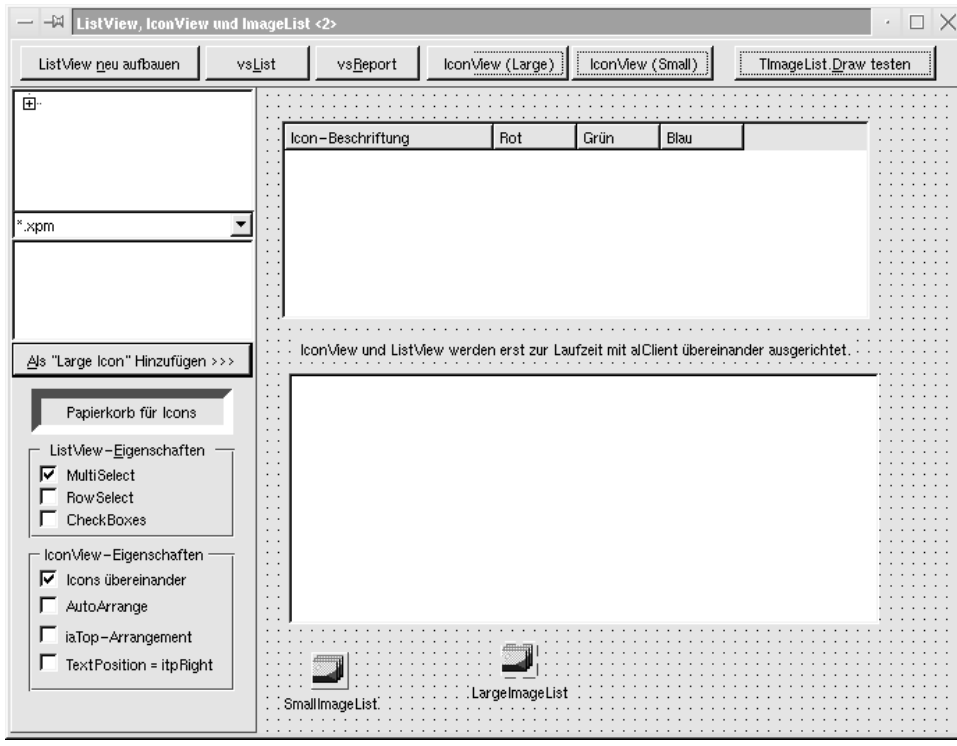


Abbildung 3.11: Steuerelemente, die sich zur Laufzeit überlappen (siehe Bild 3.9), können zur Entwurfszeit noch nebeneinander angezeigt werden.

Das Umschalten der Ansichten erfolgt im Beispielprogramm über die vier Schalter *SetIconMode*, *SetSmallIconMode*, *SetListMode* und *SetReportMode*. Die Bearbeitung ihrer *OnClick*-Ereignisse wird in einer gemeinsamen Methode zusammengefasst, in der anhand von *Sender* zwischen den Schaltern unterschieden wird. Im Falle von *SetIconMode* und *SetSmallIconMode* wird das *IconView* sichtbar gemacht, das *ListView* versteckt und außerdem die Bilderliste des *IconView* entsprechend der Ansicht (große oder kleine Icons) gewählt:

```
procedure TMainForm.SetIconModeClick(Sender: TObject);
begin
  If Sender = SetIconMode then begin
     IconView.Visible := True;
     IconView.Images := LargeImageList;
     ListView.Visible := False;
  end else if Sender = SetSmallIconMode then begin
     IconView.Visible := True;
     IconView.Images := SmallImageList;
     ListView.Visible := False;
  end
```

In den anderen beiden Fällen wird das *ListView* aktiviert und sein *ViewStyle* passend eingestellt:

```
  else if Sender = SetListMode then begin
     IconView.Visible := False;
     ListView.Visible := True;
     ListView.ViewStyle := vsList;
  end else if Sender = SetReportMode then begin
     IconView.Visible := False;
     ListView.Visible := True;
     ListView.ViewStyle := vsReport;
  end;
end;
```

Hinweis: Da insgesamt nur vier verschiedene Properties gesetzt werden, ließe sich die obige Methode natürlich auf vier Zuweisungen verkürzen, die dann allerdings etwas komplizierter wären. Zur Verdeutlichung der Vorgänge wurde daher oben die ausführliche Schreibweise gewählt.

Gemeinsame Datenobjekte für ListView und IconView

Mit dem einfachen Umschalten der Ansichten ist es jedoch noch nicht getan. Häufig werden ja auch Datenobjekte mit den Einträgen von ListView und IconView verknüpft. Wenn nun je ein ListView und ein IconView verschiedene Ansichten auf *dieselben* Daten liefern sollen, dann müssen auch beide mit denselben Datenobjekten verknüpft werden.

Da das Beispielprogramm *ListViewIconView* keine Datenobjekte mit den Einträgen verknüpft, kommen wir noch einmal auf das bereits im Abschnitt *Datenobjekte im IconView speichern* verwendete Programm *AlarmClock5* zurück. Dieses speichert die zu einem Alarmeintrag gehörenden Daten in den *Data*-Properties der ListView-Einträge. Unter *Datenobjekte im IconView speichern* wurde auch bereits beschrieben, wie diese Daten auch mit den IconView-Einträgen verknüpft werden. Insgesamt gibt es in *AlarmClock5* also zu jedem vom Benutzer eingegebenen Alarm drei Objekte:

- das *TAlarm*-Objekt mit den eigentlichen Alarmdaten,
- das *TListItem*-Objekt für das ListView-Element, in seinem *Data*-Property befindet sich ein Verweis auf das *TAlarm*-Objekt,
- das *TAlarmIcon*-Objekt für das IconView-Element. Sein *AlarmData*-Property enthält einen Verweis auf das *TAlarm*-Objekt.

Zusätzlich speichert das *TAlarm*-Objekt von *AlarmClock5* auch noch Rückverweise auf die mit ihm verknüpften *TListItem*- und *TAlarmIcon*-Objekte. Das folgende Listing fasst die entscheidenden Programmteile zusammen, die für die Verknüpfung derselben Daten mit ListView und IconView sorgen:

```
type
  TAlarm = class
    ... alle zu einem Alarm gehörenden Daten ...
    // und zusätzlich:
    IconViewItem: TIconViewItem;
    ListViewItem: TListItem;
  end;
  TAlarmIcon = class(TIconViewItem)
    AlarmData: TAlarm;
  end;
// Erzeugung eines neuen ListView-Eintrags:
  NewItem := lv.Items.Add;
  NewItem.Data := AlarmData;
  AlarmData.ListViewItem := NewItem;
// Erzeugung eines neuen IconView-Eintrags:
  iv.Items.SetItemClass(TAlarmIcon);
  NewItem := iv.Items.Add as TAlarmIcon;
  NewItem.AlarmData := AlarmData;
  AlarmData.IconViewItem := NewItem;
```

Was noch zu erläutern bleibt, ist die Begründung für die Rückverweise von *TAlarm* zu den Einträgen in ListView und IconView. Sie werden erforderlich, da IconView und ListView im weiteren Programmverlauf übereinstimmend verändert werden müssen. Im Fall von *AlarmClock5* gibt es drei Ereignisse, die die Liste der Alarme verändern. In allen drei Fällen müssen sowohl IconView als auch ListView aktualisiert werden:

- wenn der Benutzer einen neuen Alarm definiert,
- wenn er einen Alarm löscht,
- wenn er einen Alarm-Eintrag im ListView »umbenennt« (das heißt, wenn er seine Alarmzeit ändert) oder wenn er die Alarmzeit über den Alarm-Editieren-Dialog ändert.

Übereinstimmende Ereignisbearbeitung

Alle drei Ereignisse treten aber entweder im ListView auf *oder* im IconView. So kann der Benutzer z.B. im IconView Entf drücken, um den aktuellen Eintrag zu löschen. Der aktuelle Eintrag des IconViews lässt sich leicht finden, ebenso wie das zugehörige *TAlarm*-Objekt. Nun muss aber auch der zu diesem Alarm gehörende Eintrag des ListViews gelöscht werden. Um nun nicht alle Einträge des ListViews absuchen zu müssen, ist der Rückverweis *TAlarm.ListViewItem* sehr hilfreich, denn mit der Methode *TListItems.IndexOf* lässt sich der Index dieses Eintrags ermitteln und dieser ist erforderlich, um das Ziel zu erreichen – das Löschen des Eintrages im ListView:

```
procedure TAlarmForm.DelButtonClick(Sender: TObject);
var
  i: Integer;
begin
  if Sender = IconView1 then begin
    with IconView1 do
      for i:=Items.Count-1 downto 0 do
        if Items[i].Selected then begin
          Items.Delete(i);
          ListView1.Items.Delete(
            ListView1.Items.IndexOf(
              TAlarm(Items[i].Data).ListViewItem));
          TAlarm(Items[i].Data).Free;
        end;
  end else if Sender = ListView1 then begin
    ... analog zum ersten Fall ...
  end;
end;
```

Ebensolche Verwendung finden die Rückverweise in den Methoden zum Löschen der Auswahl des ListViews, zum Editieren eines Alarm-Eintrags (hier muss das *Caption*-Property beider Einträge aktualisiert werden) und zum Umbenennen des Eintrags im ListView.

Die Bilderliste des Beispielprogramms

Wie Abbildung 3.9 schon vermuten lässt, verwendet das Beispielprogramm ListView IconView keine zur Entwurfszeit festgelegte Bilderliste mit mühsam selbst gezeichneten Icons, sondern es zeichnet sich diese Icons einfach selbst. Es gestattet Ihnen zur

Laufzeit außerdem, aus einer Dateiliste Icon-Dateien auszuwählen, die Sie mit dem Schalter HINZUFÜGEN in das IconView aufnehmen können. Dabei macht es sich die Dienste der Klasse *TImageList* zunutze, der wir uns als Nächstes zuwenden werden.

Die so hinzugefügten Icons erscheinen übrigens nicht im ListView, sondern nur im IconView und auch da nur, wenn die großen Icons gewählt sind. Auf eine Übereinstimmung mit dem ListView wurde verzichtet, da es hier um die Demonstration der *TImageList*-Fähigkeiten geht und nicht um die Kombination ListView/IconView.

Bilderlisten zur Laufzeit aufbauen *R48*

Die Klasse *TImageList* verfügt trotz ihres recht einfachen Zwecks – der effizienten Verwaltung großer Mengen von Bildern – über eine umfangreiche Schnittstelle, von der hier nur die Methoden und Properties herausgegriffen werden sollen, die für den dynamischen Aufbau der Liste zur Laufzeit erforderlich sind.

TImageList besitzt drei mal zwei Methoden, um neue Bilder zur Liste hinzuzufügen: Sie beginnen mit einem der Worte *Add...*, *Insert...* und *Replace...* Während die *Add*-Methoden das Bild immer ans Ende der Liste anhängen und als Ergebnis den Index zurückliefern, den das angehängte Bild erhalten hat, geben Sie bei den beiden anderen Arten von Methoden in einem zusätzlichen Parameter an, an welcher Position das Bild in die Liste eingefügt (*Insert...*) bzw. welches alte Bild durch das neue Bild ersetzt werden soll (*Replace...*).

Die weitere Unterscheidung der Einfügemethoden verläuft nach der Art, wie die *Maske* des Bildes angegeben wird. Diese Maske gibt an, welche Teile des Bildes transparent sind, den Hintergrund also nicht abdecken. Ein transparentes Pixel nimmt immer die Hintergrundfarbe des ListViews an, unabhängig davon, welche das ist. (Liegt unter diesem transparenten Pixel beispielsweise das Icon eines anderen Eintrags, so wird dieser mit der Hintergrundfarbe übermalt. Sie können das beim Verschieben von Icons im Beispielprogramm ausprobieren.)

Die folgende Aufzählung nennt der Einfachheit halber jeweils nur die mit »*Add*« beginnende Methode:

- *Add(Image, Mask: TBitmap)* fügt ein Bild ein, das in Form eines *TBitmap*-Objekts vorliegt (siehe Kapitel 4.5.2). Die Maske des Bildes wird in Form eines zweiten *TBitmap*-Parameters festgelegt. Masken-Bitmaps verwenden pro Pixel nur ein Bit, wobei gesetzte Bits ein transparentes Pixel ausweisen.
- *AddMasked(Image: TBitmap; MaskColor: TColor)* bewirkt, dass die Bilderliste die Maske selbst berechnet. Sie markiert in der Maske jedes Pixel als transparent, das in *Image* die in *MaskColor* angegebene Farbe aufweist.

Zu den weiteren Methoden von *TImageList* gehören auch *Delete* und *Move*, mit denen Sie Bilder auch wieder löschen und verschieben können. Das Beispielprogramm macht hiervon jedoch keinen Gebrauch.

Der folgende Code-Auszug zeigt, wie *ListViewIconView* die Liste der großen Bilder mit Hilfe eines einzigen *TBitmap*-Objekts erzeugt, das wiederholt an die Methode *AddMasked* übergeben wird. (Die Anweisungen, die die Grafik zeichnen, wurden dabei weggelassen, ebenso die äußere Schleife, die den gezeigten Programmabschnitt noch einmal für die Liste der kleinen Bilder wiederholt.)

```
BM := TBitmap.Create;
try
  LargeImageList.Clear;
  BM.Height := 32;    // Erzeugen der großen Bitmaps
  BM.Width := BM.Height;
  for iBitmap := 0 to ElementCount-1 do begin
    with BM.Canvas do begin
      ... Zeichnen der Bitmap, siehe CD-ROM ...
    end;
    LargeImageList.AddMasked(BM, clWhite)
  end;
finally
  BM.Free;
end;
```

Während dieser Code bei Programmstart ausgeführt wird (*OnCreate*-Ereignis des Formulars), um einen festen Satz an Bildern zu erhalten, können auch noch nachträglich – z.B. zusammen mit einem neuen Eintrag für das ListView – Bilder zu einer Bilderliste hinzugefügt werden.

Wenn Sie in der Dateiliste des Beispielprogramms eine Bilddatei markieren und den Schalter HINZUFÜGEN drücken, lädt das Programm die gewählte Datei, verkleinert sie auf die Größe eines Icons, fügt sie so in die Bilderliste mit den großen Bildern ein und zeigt im IconView ein neues Element mit diesem Bild an (wie erwähnt unterbleibt eine entsprechende Ergänzung des ListViews).

Zum Verkleinern verwendet das Programm eine temporäre Bitmap. Während die Bitmap *TempImage* zum Laden der eigentlichen Datei verwendet wird, dient *Image* als Zeichenoberfläche, auf der mit *StretchDraw* eine verkleinerte (oder auch vergrößerte, falls das Bild in der Datei noch kleiner ist als ein IconView-Icon) Version des Bildes ausgegeben wird. Die Größe von *Image* wird bei der Initialisierung exakt auf die von der ImageList erwartete Größe gesetzt. Beim *AddMasked* wird einfach die Farbe Weiß als Maskenfarbe angegeben, was natürlich blind geraten ist und nicht bei allen Bildern eine Farbe treffen wird, die sinnvollerweise transparent dargestellt werden sollte:

```
procedure TMainForm.InsertFile;
var
  Index: Integer;
  ListItem : TListItem;
  IconViewItem: TIconViewItem;
  TempImage, Image: TBitmap;
begin
  FileListBox.FileName;
  TempImage:=TBitmap.Create;
  try
    TempImage.LoadFromFile(FileListBox.FileName);
    Image := TBitmap.Create;
    try
      Image.Width := LargeImageList.Width;
      Image.Height := LargeImageList.Height;
      Image.Canvas.StretchDraw(
         Rect(0, 0, Image.Width, Image.Height),
         TempImage);
      Index:=LargeImageList.AddMasked(Image, clWhite);
    finally
      Image.Free;
    end;
  finally
    TempImage.Free;
  end;
  IconViewItem:=IconView.Items.Add;
  IconViewItem.Caption:=FileListBox.FileName;
  IconViewItem.ImageIndex:=Index;
end;
```

Da *TImageList* eine Kopie aller mit *Add...* hinzugefügten Bilder behält, wird auch die *Image*-Bitmap mit *Free* freigegeben.

3.6.3 Baumanzeige-Komponenten (TreeViews)

Erfreulicherweise bestehen zwischen *TListView* und *TTreeView* große Ähnlichkeiten, so dass Sie, wenn Sie eine dieser Komponenten kennen, die andere nicht von Grund auf neu kennen lernen müssen. So laufen beispielsweise das Verknüpfen der Einträge mit Datenobjekten und das Editieren der Eintrags-Beschriftungen nach demselben Prinzip ab.

Auch die Anbindung an eine Bilderliste über das Property *Image* entspricht den Verältnissen bei *TListView*. Die Klasse der einzelnen Einträge heißt *TTreeNode* und wie *TListItem* verfügt sie über ein Property *ImageIndex*, das den Index des Bilderlisten-Bildes angibt, welches neben dem Text des Eintrages angezeigt werden soll.

TTreeNode besitzt jedoch noch einen zweiten Bilderindex: In *SelectedIndex* geben Sie ein Bild an, das dargestellt werden soll, wenn der Eintrag selektiert ist. Dieses Bild tritt dann an die Stelle des mit *ImageIndex* angegebenen Bildes. Ein typisches Beispiel sind

Dateisystem-Browser, in denen das ausgewählte Verzeichnis als geöffneter Ordner, alle anderen Verzeichnisse als geschlossene Ordner dargestellt werden. Um den gewählten Eintrag zur Laufzeit abzufragen, verwenden Sie das Property *TTreeView.Selected*.

TTreeView zur Entwurfszeit

Wie schon bei *TListView* geben Sie auch bei einem *TTreeView* den Text der einzelnen Elemente und die Indizes für die Bilder im Editor für das Property *Items* an. Sogar der Schalter NEUER UNTEREINTRAG stimmt mit dem entsprechenden Editor der ListViews überein – mit dem Unterschied, dass Sie bei TreeView einen echten Untereintrag erhalten, bei dem Sie auch die Felder *Bild-Index* und *Ausgewählt-Index* editieren können.

TTreeNodes

Die Klasse *TTreeNodes* und damit das *Items*-Property von *TTreeView* ist um einiges komplizierter strukturiert als die Eintragsliste eines ListViews, da jedes einzelne Element der Klasse *TTreeNode* als Unterelemente nicht einfach eine Liste von Strings (wie *TListItem*), sondern wieder eine Liste von einzelnen *TTreeNode*-Objekten zugeordnet bekommt. So können Sie mit *TreeView.Items[0][4][0].Text* den Text des ersten Untereintrags des fünften Untereintrags der Wurzel ansprechen.

Um einem Eintrag einen neuen Untereintrag zu spendieren, dürfen Sie sich jedoch nicht an diesen Eintrag wenden (ein Aufruf wie *Items[0].Add* ist also nicht möglich), sondern Sie müssen wieder eine *Add*-Methode der Gesamtliste aufrufen (*Items.Add(...)*). Dafür haben Sie dort gleich acht *Add*-Methoden zur Auswahl. Diese Methoden unterscheiden sich in drei Punkten:

- Bei den vier *AddChild...*-Methoden geben Sie im ersten Parameter einen Eintrag an, der einen neuen Untereintrag erhalten soll. Bei den Methoden ohne das »Child« im Namen geben Sie stattdessen einen Eintrag der Baumebene an, in der der neue Eintrag eingefügt werden soll (also einen Geschwistereintrag).
- Bei den vier *Add[Child]Object...*-Methoden geben Sie als Parameter eine Beschriftung für den neuen Eintrag und einen Zeiger auf zu verknüpfende Daten an. Die Methoden ohne das »Object« erwarten nur einen Beschriftungs-String.
- Die vier *Add[Child][Object]First*-Methoden fügen den Eintrag vor alle benachbarten Einträge an; ohne das »First« wird er hinter seinen Geschwistereinträgen eingeordnet.

Das folgende Beispiel entstammt der Komponente *TDirectoryTreeView* aus der Komponentenbibliothek des Buchs (siehe auch Kapitel 8.8). *TDirectoryTreeView* zeigt die Verzeichnisstruktur der Festplatte bzw. des gesamten Verzeichnisbaums von Linux an, dabei wird jeder Knoten mit dem Namen eines Verzeichnisses versehen (*Caption*, wird

von *AddChild* gesetzt). Als Bilderindizes werden die Indizes 0 und 1 vergeben, es wird angenommen, dass eine enstprechende Bilderliste zur Verfügung steht.

Neue TTreeNode-Klassen definieren

Zusätzlich sollen mit jedem Knoten weitere Daten verknüpft werden. Hierzu stellt *TTreeView* zwei Ansatzpunkte bereit, die bereits von anderen Komponenten bekannt sind:

- Mit *TTreeNode.Data* können Sie eine selbst definierte Datenstruktur verknüpfen. Diese Vorgehensweise ist auch von *TListView* und *TListBox* bekannt.
- Über *Items.SetNodeClass* können Sie wie bei *TIconView* mit *Items.SetItemClass* eine eigene Klasse für die Elemente der Komponente definieren. Im Fall von *TTreeView* muss diese Klasse von *TTreeNode* abgeleitet werden.

TDirectoryTreeView wählt den zweiten Weg, denn bei diesem werden die eigenen Daten automatisch über den virtuellen Destruktor von *TTreeNode* freigegeben, der ja an die abgeleitete Klasse vererbt wird, während die mit *TTreeNode.Data* verknüpften Daten selbst freigegeben werden müssten (etwa beim *OnDeletion*-Ereignis). Die abgeleitete Klasse sieht wie folgt aus:

```
type
  TDirectoryNode = class(TTreeNode)
  public
    Path, DirName: String;
    Expanded: Boolean;
  end;
```

Diese Klasse befolgt zwar nicht die reine Lehre des OOP, da sie öffentliche Variablen enthält, jedoch ist ein Zugriff über Properties bei dieser kleinen und nur in dieser Unit intern verwendeten Klasse nicht notwendig. Wichtig ist, dass von der Klasse keine dynamischen Objekte reserviert werden, so dass auch kein Destruktor notwendig ist und der geerbte Destruktor die gesamten Daten freigeben kann.

Die neuen Daten von *TDirectoryNode* sind der komplette Pfad des durch den Knoten repräsentierten Verzeichnisses (*Path*), der Name des Verzeichnisses, der auch schon in *Caption* steht (*DirName*), und ein boolescher Wert, der angibt, ob die Unterverzeichnisse des aktuellen Verzeichnisses bereits in den Baum eingefügt wurden (*Expanded*).

Die folgende Methode erzeugt einen Knoten dieser abgeleiteten Klasse. Da hierfür die *AddChild*-Methode von *TTreeView.Items* zuständig ist, muss *Items* vorher mit *SetNodeClass* über die neue Klasse informiert werden.

```
procedure TDirectoryTreeView.CreateDirNode(Path, DirName: String;
  Parent: TTreeNode);
var
  N: TDirectoryNode;
```

```
begin
  Items.SetNodeClass(TDirectoryNode);
  N := Items.AddChild(Parent, DirName) as TDirectoryNode;
  N.HasChildren := True; // wird im Abschnitt
    // Effizienter Aufbau des Baumes erläutert
  N.ImageIndex := 1;
  N.SelectedIndex := 0;
  N.Path := Path;
  N.DirName := DirName;
  N.Expanded := False;
end;
```

Hinweis: Falls Sie bisher nur Ereignismethoden für Formulare programmiert haben und die Komponentenprogrammierung neu für Sie ist, beachten Sie bitte, dass in den hier gedruckten Methoden der Komponente *TDirectoryTreeView* die verwendete TreeView-Komponente nicht explizit genannt werden muss (Stichwort »*self*-Parameter«). In Ereignismethoden von Formularen müssen Sie natürlich weiterhin eine konkrete TreeView-Komponente angeben. So müsste beispielsweise *Items.AddChild* oben durch so etwas wie *TreeView1.Items.AddChild* ersetzt werden.

Wie die Methode *CreateDirNode* aufgerufen und in den Ablauf zum Aufbau des gesamten Baums eingebunden ist, können Sie den Listings im folgenden Abschnitt entnehmen.

Effizienter Aufbau des Baumes *R56*

Die wohl wichtigste Eigenschaft eines TreeViews ist, dass es einzelne Zweige des Baumes ein- und ausblenden kann. Normalerweise sind Baumstrukturen zu Beginn der Übersicht wegen nur zu einem kleinen Teil geöffnet oder ganz geschlossen (der KDE-Konqueror öffnet zum Beispiel nach dem Start nur die Zweige, die zu einem bestimmten Verzeichnis führen).

Es wäre aber sehr ungünstig, wenn zwar der Anwender von der Komplexität des Baumes verschont würde, das Programm den Baum aber immer komplett aufbauen müsste. Dies würde z.B. für den Verzeichnisbaum einer Festplatte sehr lange dauern. Jeder Baumeintrag hat daher ein Property namens *HasChildren*, in dem Sie mit dem Wert *True* angeben können, dass der Eintrag Unterelemente hat, obwohl Sie diese noch nicht in der Eintragsliste gespeichert haben.

Sie können die Einträge dann hinzufügen, sobald der Benutzer versucht, den Eintrag zu expandieren. In diesem Fall erzeugt *TTreeView* nämlich ein *OnExpanding*-Ereignis, das als Parameter unter anderem den Zweig hat, der gerade expandiert werden soll.

Sofern Sie das TreeView-Property *ShowButtons* bei seinem voreingestellten Wert *False* lassen, kennzeichnet der TreeView sowohl die Einträge, die bereits Kinder haben, als

auch die Einträge, bei denen nur das Property *HasChildren* auf *True* gesetzt ist, mit einem Plus-Symbol. Wenn der Anwender auf dieses Symbol klickt, wird der Zweig expandiert und das Symbol verwandelt sich in ein Minus-Zeichen.

Um wieder auf das Beispiel der Klasse *TDirectoryTreeView* zurückzukommen: Hier wird zu Beginn nur eine Verzeichnisebene angezeigt, indem die im Folgenden abgedruckte Methode *MakeChildNodes* für das Wurzelverzeichnis aufgerufen wird. Während die im letzten Abschnitt gezeigte *CreateDirNode* nur ein einziges Verzeichnis zu den *Items* hinzufügt, sucht die folgende Methode *alle* Unterverzeichnisse eines gegebenen Verzeichnisses, und zwar des Verzeichnisses, das durch den im Parameter *N* angegebenen TreeNode repräsentiert wird. Die einzelnen Knoten des Baums werden dann mit *CreateDirNode* erzeugt:

```
procedure TDirectoryTreeView.MakeChildNodes(N: TTreeNode);
var
  sr: TSearchRec;
  Path: String;
  DirNode: TDirectoryNode;
begin
  DirNode := N as TDirectoryNode;
  if DirNode.Expanded = False then begin
    DirNode.Expanded := True;
    Path := IncludeTrailingPathDelimiter(DirNode.Path);
    if FindFirst(Path+'*', faAnyFile, sr) = 0 then begin
      repeat
        if (sr.Attr and faDirectory) <> 0 then
          if (sr.Name <> '.') and (sr.Name <> '..') then
            CreateDirNode(Path+sr.Name, sr.Name, N);
      until FindNext(sr) <> 0;
      FindClose(sr);
    end else DirNode.HasChildren := False;
  end;
end;
```

Beachten Sie, dass *CreateDirNode* das Property *HasChildren* immer mit *True* initialisiert, ohne dass geprüft wird, ob überhaupt Unterverzeichnisse im aktuellen Verzeichnis existieren. Erst, wenn *MakeChildNodes* aufgerufen wird, dieses mit *FindFirst* nach Unterverzeichnissen sucht und keines findet, wird *HasChildren* auf *False* gesetzt.

Schließlich bedarf es noch einer Methode, die den ganzen Vorgang startet. Wenn der Verzeichnisbaum neu aufgebaut werden soll, löscht *BuildTree* alle bisherigen Einträge und ruft *MakeChildNodes* für das Wurzelverzeichnis auf (das Verzeichnis, welches die Wurzel des vom TreeView angezeigten Bereichs bildet, wird über das Property *TDirectoryTreeView.RootDirectory* angesprochen und intern in der Variablen *FRootDirectory* gespeichert):

```
procedure TDirectoryTreeView.BuildTree;
begin
  Items.Clear;
  CreateDirNode(FRootDirectory, FRootDirectory, nil);
end;
```

Entscheidend für den dynamischen Aufbau des Baums ist nun, dass die Methode *MakeChildNodes* immer aufgerufen wird, wenn der Benutzer des TreeViews eine Verzeichnisebene expandieren will, indem er auf das Plus-Symbol neben dem Knoten klickt. Wir könnten nun das Ereignis *OnExpanding* im Objektinspektor wählen und eine Methode dafür erstellen, in der *MakeChildNodes* für den expandierten Knoten aufgerufen würde.

```
procedure TForm1.TreeView1Expanding(Sender: TObject; Node: TTreeNode;
  var AllowExpansion: Boolean);
begin
  MakeChildNodes(Node);
  AllowExpansion := True;
end;
```

Da aber diese Funktionalität nicht erst im Formular hinzugefügt, sondern fest in die Komponente *TDirectoryTreeView* integriert werden soll, findet der *MakeChildNodes*-Aufruf in einer weiteren Methode von *TDirectoryTreeView* statt. Auf der Seite der Komponente entspricht dem *OnExpanding*-Ereignis die dynamische Methode *CanExpand*:

```
function TDirectoryTreeView.CanExpand(Node: TTreeNode): Boolean;
begin
  Result := True;
  MakeChildNodes(Node);
end;
```

Auch hier sei noch einmal auf das Kapitel zur Komponentenprogrammierung hingewiesen.

3.7 Frames und verwandte Techniken

Trotz der extremen Beschleunigung, die die Softwareentwicklung beim Entwurf der Benutzerschnittstelle durch Delphi und Kylix erfahren hat, bleibt noch enorm viel Spielraum für weitere Effizienzsteigerungen, wenn man vom Formular-Editor ausgeht, wie er auch in der Kylix-IDE anzutreffen ist. Zu häufig gibt es beim visuellen Entwurf wiederkehrende Aufgaben, die bei der x-ten Wiederholung sehr lästig werden können und die nach Vereinfachung verlangen: Manche Properties müssen immer auf dieselben Werte eingestellt werden (z. B. TPanel.Caption auf einen leeren String, damit die Panel-Oberfläche frei wird), einfache Formularbestandteile müssen immer aus denselben Komponenten zusammengebaut werden (so etwa die Kombination aus Label und Editierfeld) und in größeren Anwendungen mit vielen Formularen kann es

vorkommen, dass viele Formulare sich ähneln und zum Teil aus den gleichen Komponenten bestehen. Zu den Komponenten und deren Properties können dann auch noch Ereignismethoden kommen, die ebenfalls immer gleich sind.

Ein Beispiel für eine mehrfach vorkommende Komponentengruppe wäre z. B. in einer Firmendatenverwaltung mit verschiedenen Personenlisten für Kunden und Mitarbeiter, die zum Teil die gleichen Tabellenspalten wie Name, Vorname, Straße usw. enthalten, zum Teil aber auch andere Felder aufweisen. Die Dialoge zum Ändern von Kunden- und Mitarbeiterdaten würden dann in den Eingabefeldern für die Adresse bereinstimmen. Ein anderes Beispiel wäre es, wenn eine Komponentengruppe zur Auswahl eines Verzeichnisses und einer Datei in vielen verschiedenen Formularen verwendet wird.

Kylix bietet eine ganze Reihe von Mechanismen an, mit der sich der Entwurf von Formularen mit wiederkehrenden und ähnlichen Bestandteilen vereinfachen lässt. Die Palette dieser Mechanismen nimmt fast mit jeder neuen Delphi-Version zu und dies sei als Gelegenheit genutzt, einen kleinen geschichtlichen Ausflug zu den Vorfahren des Kylix-Entwicklungssystems zu machen:

- Seit Delphi 1 lassen sich komplette Formulare und Projekte bequem wiederverwenden, wenn Sie eine Schablone davon in der Objektablage ablegen (siehe auch Kapitel 1.6.4).
- In Delphi 2 wurde diese Verwendung von Formularschablonen näher an die Prinzipien der OOP angenähert, indem ein Formular nicht unbedingt eine exakte Kopie einer Schablone sein muss, sondern von einem Schablonenformular erben kann.
- Delphi 3 brachte der Entwickler-Gemeinde eine Technik zur Wiederverwendung von Komponentengruppen über den Weg von Komponentenschablonen, die in der Komponentenpalette abgelegt werden können.
- Eine ganz allgemeine und mächtige Technologie zur Wiederverwendung von Komponentengruppen liegt natürlich in der Entwicklung neuer Komponentenklassen. Bereits seit der ersten Delphi-Version ist es möglich, mehrere Komponenten zu einer neuen Komponente zusammenzufassen (siehe Kapitel 8.5.4), allerdings findet diese Komponentenentwicklung nicht-visuell statt, fällt also hier ein wenig aus dem Rahmen.
- Seit Delphi 5 gibt es das Konzept der *Frames*, bei dem das Prinzip der Vererbung (in Delphi 2 schon bei Formularen bekannt) auf Komponentengruppen Anwendung findet.

Ziel dieses Kapitels ist es, die Vor- und Nachteile dieser Techniken an Beispielen zu erläutern und den Einsatz von Formularvererbung und Frames zu demonstrieren.

3.7.1 Formularvererbung

Die Formularvererbung ist ein Teil des Funktionsangebots der Objektablage und wird in Kapitel 1.6.4 im Einzelnen beschrieben. An dieser Stelle geht es um ein Beispiel, das in Kapitel 3.7.2 über die Frames fortgesetzt wird.

Als Beispiel soll eine Situation dienen, in der sich die Vererbung der Formulare aus der Vererbung von nicht-visuellen Klassen des Quelltextes herleiten lässt. Nehmen wir eine Anwendung an, in der es verschiedene Klassen von Grafikobjekten gibt, die von einer gemeinsamen Basisklasse abgeleitet sind. Die Basisklasse definiert die Eigenschaften, die allen Grafikobjekten gemein sein sollen, so etwa die Koordinaten des umgebenden Rechtecks und die Stift- und Pinselattribute zum Zeichnen des Objekts. Als spezielle davon abgeleitete Klasse könnte es z. B. eine Klasse geben, die einen Farbverlauf zwischen zwei beliebigen Farben darstellen sollen. Sie müsste die Daten der Basisklasse durch zwei Farbwerte ergänzen. Ein anderes Beispiel wären spezielle Grafikobjekt-Klassen für Diagrammsymbole, wie sie in CASE-Tools zur Darstellung eines Software-Entwurfs verwendet werden. Die Objekte für diese Symbole könnten den Daten der Basisklasse z. B. diverse Beschriftungen für verschiedene Stellen des Symbols hinzufügen, wie sie z. B. in UML-Diagrammen zu finden sind.

Was an dieser Stelle nicht interessiert, ist die Problematik der Wahl der günstigsten Klassenaufteilung einer Grafikanwendung (so wäre es vielleicht günstiger, Farbverläufe schon als Eigenschaft der grundlegenden Basisklasse zu implementieren, so dass *alle* speziellen Grafikobjekte über die Option eines Farbverlaufs verfügen würden). Thema dieses Kapitels soll es ausschließlich sein, eine gegebene Vererbungshierarchie auf den Formularentwurf zu erweitern.

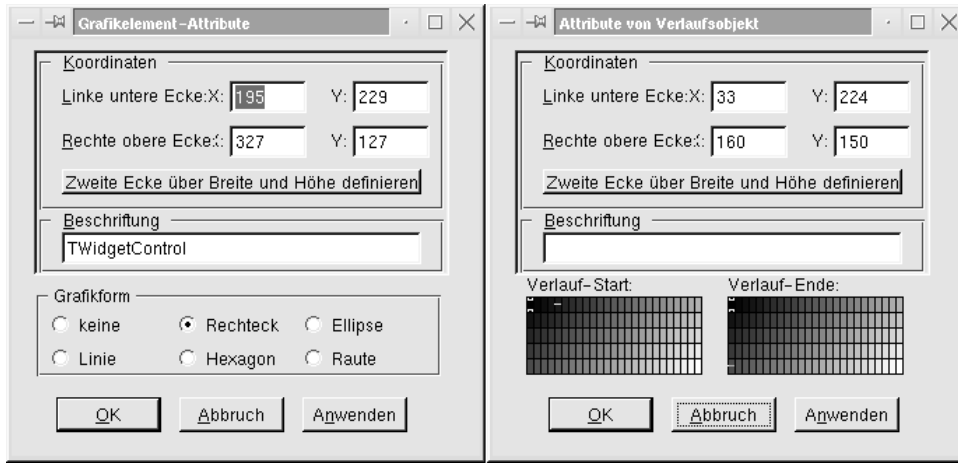


Abbildung 3.12: Das Ziel ist, diese beiden und eventuell viele weitere Grafikobjekt-Dialoge zu entwerfen, und dabei die gemeinsamen Komponenten nur einmal an einer zentralen Stelle zu definieren.

Abbildung 3.11 zeigt zwei Dialoge, die im beschriebenen Beispiel verwendet werden önnten. Das Problem für den Formular-Entwickler besteht nun darin, die gemeinsamen Teile des Dialogs nicht jedes Mal neu entwerfen zu müssen. Im Fall einer Anwendung mit nur zwei Dialogen kann man natürlich auf alle besonderen Funktionen von Kylix verzichten und die gemeinsamen Komponenten einfach über die Zwischenablage vom ersten in das zweite Formular kopieren. Diese Lösung liegt jedoch nicht im Sinn der OOP und greift bei einer Anwendung, die später um viele weitere Grafikklassen und somit auch etliche neue Dialoge erweitert werden soll, etwas kurz. Probleme gibt es spätestens dann, wenn der Entwurf der gemeinsamen Dialogelemente geändert werden soll: Wenn diese Elemente erst einmal in alle Dialogformulare kopiert sind, muss auch jede Änderung an ihnen erneut in jedes einzelne Formular übertragen werden.

Die in Kylix anzutreffende Objektablage bietet mit den Formularschablonen eine einfache Lösung für das vorliegende Problem. Abbildung 3.12 zeigt den Entwurf eines Formulars mit den gemeinsamen Komponenten der beiden Dialoge aus Abbildung 3.11 sowie den Dialog der Objektablage, den Sie mit DATEI | NEU aufrufen und über den Sie auf Grundlage des bestehenden Formulars ein neues Formular über die Formularvererbung erzeugen können. Die Formulare des aktuellen Projekts sind im DATEI | NEU- bzw. Objektablage-Dialog auf einer eigenen Seite mit dem Namen des Projekts angeordnet. Um die Vererbung zu nutzen, stellen Sie vor dem Drücken des Ok-Schalters sicher, dass im unteren Teil des Dialogs die Option VERERBEN gewählt ist.

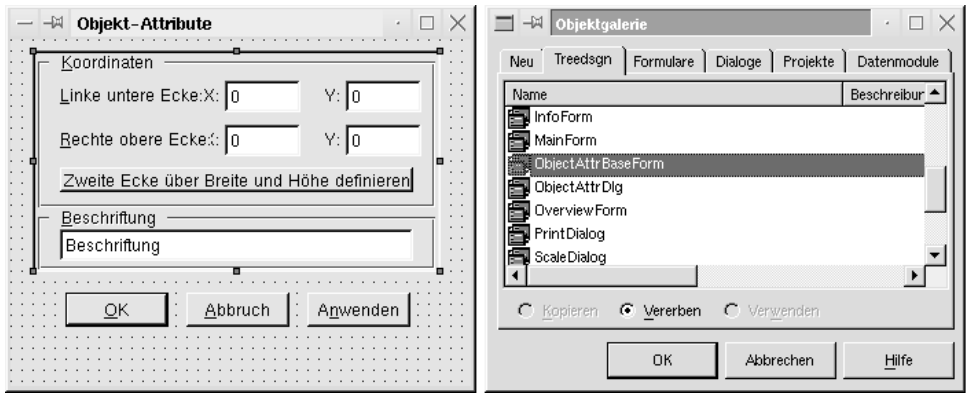


Abbildung 3.13: Eine Formularschablone mit den gemeinsamen Teilen des Dialogs; rechts das Erscheinen der hablone in der Objektgalerie

Wenn wir auf diese Weise im Beispiel zwei »Abzüge« dieser Schablone anfertigen, kommen wir durch Hinzufügen der fehlenden Komponenten exakt zu den in Abbildung 3.11 gezeigten Dialogen.

Hinweise: Im beschriebenen Beispiel werden zur Laufzeit des Programms nur die beiden abgeleiteten Formulare als sichtbare Fenster auf dem Bildschirm benötigt. Standardmäßig würde jedoch auch ein Exemplar des Basisformulars erzeugt werden. Um diesen unnötigen Aufwand zu vermeiden, können Sie die automatische Erzeugung dieses Formulars in den Projektoptionen auf der Seite FORMULARE abschalten.

Zur Option VERERBEN und den drei anderen Optionen der Objektablage siehe Kapitel 1.6.4. Die Option KOPIEREN ist übrigens in diesem Beispiel gleichwertig mit dem erwähnten Kopieren der übereinstimmenden Komponenten über die Zwischenablage.

Formularvererbung im Entwurfsmodus

Für die Veränderung eines Formulars, das von einem anderen Formular erbt, gibt es zwei Ansatzpunkte: Sie können das Basisformular oder das erbende Formular editieren. Die Kylix-IDE kann mit beiden Situationen gut umgehen. Sie können die beiden Formulare zur Entwurfszeit nebeneinander legen, im Basisformular ein paar Komponenten verschieben und beobachten, wie Kylix automatisch für eine Anpassung der Komponenten im abgeleiteten Formular sorgt. Bei einer Änderung im abgeleiteten Formular tut sich dagegen im Basisformular nichts.

Ganz allgemein wird jede Änderung am Basisformular zur Entwurfszeit sofort in das abgeleitete Formular übertragen, sofern die geänderte Eigenschaft dort nicht *überschrieben* wurde. Das Überschreiben von geerbten Eigenschaften (damit sind sowohl Properties und Events des Formulars selbst als auch die seiner Komponenten gemeint) entspricht dem Überschreiben von Klassenelementen im OOP.

Um eine geerbte Eigenschaft zu überschreiben, passen Sie sie einfach im abgeleiteten Formular an. Vor einer Änderung einer Eigenschaft des Basisformulars prüft Kylix, ob die Eigenschaft im abgeleiteten Formular denselben Wert aufweist. Ist das der Fall, wird sie auch im abgeleiteten Formular geändert, ansonsten betrachtet Kylix die Eigenschaft als überschrieben und belässt sie im abgeleiteten Formular bei ihrem individuellen Wert.

Hinweis: Dies zeigt, wie Sie das Überschreiben einer geerbten Eigenschaft rückgängig machen können: Setzen Sie die Eigenschaft einfach wieder auf den Wert, den die Eigenschaft im Basisformular hat. Ein anderer Weg besteht darin, das abgeleitete Formular im Textmodus zu betrachten und alle Eigenschaften, die unverändert vom Basisformular übernommen werden sollen, zu löschen.

berschreiben von Event-Handlern

Das Überschreiben von Ereignisbearbeitungsmethoden funktioniert prinzipiell genauso wie das Überschreiben von Property-Werten. Allerdings ist es im Falle der Events häufig wünschenswert, dass die geerbte Ereignisbehandlung nicht komplett ersetzt, sondern nur ergänzt wird. Kylix rechnet bereits damit und fügt in neue Ereignismethoden eines abgeleiteten Formulars automatisch die Anweisung *inherited* ein:

```
procedure TForm2.FormKeyPress(Sender: TObject; var Key: Char);
begin
  inherited;

end;
```

Inherited sorgt für den Aufruf einer eventuellen Bearbeitungsmethode im Basisformular. Es steht Ihnen jedoch frei, vor diesem Aufruf eigene Anweisungen einzufügen oder den Aufruf ganz zu löschen, wenn Sie die geerbte Ereignisbearbeitung ausschalten wollen.

In jedem Fall können Sie die geerbte Ereignisbearbeitung wiederherstellen, wenn Sie die Verknüpfung des Ereignisses im abgeleiteten Formular entfernen, also das entsprechende Ereignis-Feld im Objektinspektor löschen.

ererbungsbeispiel im TreeDesigner

Der in Kapitel 6 vorgestellte TreeDesigner gibt ein Beispiel für die Formularvererbung, und zwar ist er die Quelle der in Abbildung 3.11 gezeigten Dialoge. Allerdings sind diese Dialoge anders realisiert, als es bisher beschrieben wurde. Die in beiden Dialogen bereinstimmenden Komponenten wurden nicht über die Formularvererbung, sondern über Frames realisiert (hierzu mehr in Kapitel 3.7.2). Das in Abbildung 3.12 gezeigte Basisformular heißt im TreeDesigner *TObjectAttrBaseForm* und ist in `AttrDlg0.pas` definiert. Die beiden in Abbildung 3.11 gezeigten Dialoge sind *TObject AttrDlg* in `AttrDlg1.pas` und *TVerlaufObjAttrDlg* in `AttrDlg2.pas`.

Das Basisformular enthält auch die drei Schalter der Dialoge inklusive der dazu gehöenden Ereignismethoden. Da die Funktion dieser Methoden sehr eng mit der des Dialogformulars verknüpft ist und es kaum Sinn machen würde, diese drei Schalter in öllig andere (nicht von diesem Formular erbende) Formulare zu übernehmen, wurden sie auch nicht als Frame realisiert. Außerdem definiert das Basisformular noch den Rahmentyp (*BorderStyle* = *bsToolWindow*) und die Bildschirmposition (*Position* = *poMainFormCenter*) der beiden Dialoge

Hinweis: Um die Frames im TreeDesigner in Aktion zu sehen, laden Sie eine Grafikdatei in den TreeDesigner, klicken mit der rechten Maustaste auf ein Grafikelement und wählen den Menüpunkt ATTRIBUTE. Die Änderungen im Dialog werden wirksam, wenn Sie OK oder ANWENDEN drücken. Bei ANWENDEN bleibt der Dialog noch aktiv, so dass Sie weitere Änderungen vornehmen können, falls das bisherige Ergebnis Ihnen nicht gefällt.

Vererbung von Methoden

Neben der Vererbung von Komponenten und Ereignismethoden bietet es sich an, bei der Formularvererbung auch zusätzliche selbst definierte Methoden und Properties im Basisformular zu definieren, die vielleicht in allen abgeleiteten Formularen benötigt werden. Besonders flexibel wird die Vererbungsbeziehung zwischen zwei Formularen, wenn Sie virtuelle Methoden einsetzen.

Das TreeDesigner-Formular *TObjectAttrBaseForm* definiert beispielsweise ein Property, in dem das Grafikelement gespeichert wird, dessen Attribute im Dialog editiert werden:

property EditObject: TGraphicElement read GetEditObject write SetEditObject;

Hinzu kommen zwei virtuelle Methoden, die es immer dann aufruft, wenn der Inhalt der Steuerelemente auf das Grafikelement übertragen werden soll oder umgekehrt:

```
procedure TObjectAttrBaseForm.ReadObject; { virtuelle Methode }
begin // alle Steuerelemente mit den Daten von EditObject füllen
  BasicAttrFrame.CancelChanges;
end;

procedure TObjectAttrBaseForm.WriteObject; { virtuelle Methode }
begin // den Inhalt der Steuerelemente auf das EditObject anwenden
  BasicAttrFrame.ApplyChanges;
end;
```

Diese bedienen sich lediglich der Methoden *CancelChanges* und *ApplyChanges*, die in der Frame-Unit definiert und hier nicht weiter von Bedeutung sind. Interessant ist an dieser Stelle, dass die abgeleiteten Formulare diese Methoden überschreiben können. Zum Beispiel der Dialog für die Attribute eines »normalen« Grafikelements: Er enthält ja neben dem geerbten Frame mit den Koordinaten des Grafikelements eine *RadioGroup*, mit der die Form des Elements eingestellt werden kann. In seinen Methoden *ReadObject* und *WriteObject* überträgt er deshalb die Form des Grafikobjekts (*EditObject.ShapeType*) in die *RadioGroup* (*rgGrafikForm*) und umgekehrt:

```
procedure TObjectAttrDlg.ReadObject;
begin
  inherited ReadObject;
  rgGrafikForm.ItemIndex := integer(EditObject.ShapeType);
end;
```

```
procedure TObjectAttrDlg.WriteObject;
begin
  inherited WriteObject;
  EditObject.ShapeType := GrDoc.TShapeType(rgGrafikForm.ItemIndex);
end;
```

Das Basisformular ruft nun *ReadObject* und *WriteObject* immer dann auf, wenn es erforderlich ist: *ReadObject* wird aufgerufen, wenn das Property *EditObject* neu gesetzt wird. *WriteObject* wird aufgerufen, wenn einer der Schalter OK oder ANWENDEN gedrückt wird. Dank der virtuellen Methoden kann so die Funktionalität dieser Schalter und des Properties schon in der Formular-Basisklasse definiert werden, auch wenn in den abgeleiteten Formularen zusätzliche Aktionen hinzukommen.

Hinweis: Das Beispiel zeigt auch, dass Kylix mehrstufige Vererbung von Formularen unterstützt. Wenn Sie den Frame mit den grundlegenden Objektattributen im Entwurfsmodus verändern (mehr dazu in Abschnitt 3.7.2), wirkt sich dies nicht nur auf das Basisformular mit dem Frame aus, sondern auch die beiden von diesem abgeleiteten Formulare werden sofort geändert.

Umstellung bestehender Formulare auf Vererbung *R10*

Die beschriebene Verbindung zwischen Basisformularen und abgeleiteten Formularen zur Entwurfszeit bedeutet für die Kylix-IDE einigen Verwaltungsaufwand. Ebenso önnte die Tatsache, dass man ein geerbtes Formular über einen Dialog der Objektablage erzeugt, den Anschein erwecken, dass sich die Kylix-IDE lieber nicht vom Benutzer in die Formularvererbung hineinreden lässt. Dies ist jedoch ein Irrtum, denn auch bei der Formularvererbung sind Sie nicht von der Expertenhilfe der Kylix-IDE abhängig, sondern können sie auch »von Hand« durchführen.

Dies eröffnet Ihnen die Möglichkeit, ein bestehendes Formular nachträglich von einem anderen Formular abzuleiten. Führen Sie hierfür folgende Schritte aus:

- Falls das Formular, das Sie verändern wollen, Ihnen etwas bedeutet, legen Sie zunächst eine Sicherheitskopie davon an. Aber auch sonst kann eine Sicherheitskopie es Ihnen später erleichtern, eventuelle Fehler beim Editieren des Formular-Quelltextes rückgängig zu machen.
- Ändern Sie die Deklaration der Klasse des Formulars, das von einem anderen erben soll, im Object-Pascal-Quelltext. Ändern Sie also z.B. *TDialog2 = class(TForm)* in *TDialog2 = class(TBaseDialog)*.
- Editieren Sie das Formular im Textmodus (lokales Menü: ANSICHT ALS TEXT) und ersetzen Sie das erste Schlüsselwort *object* durch *inherited*. So wird beispielsweise aus *object Dialog2: TDialog2* die Zeile *inherited Dialog2: TDialog2*.

- Wenn Sie sich in der Textansicht des Formulars befinden, können Sie auch gleich alle Properties löschen, die das Formular von der Basisklasse erben, also nicht überschreiben soll. Löschen Sie z.B. die Properties *Width* und *Height*, falls das Formular seine Größe von der des Basisformulars übernehmen soll.
- Wenn Sie für das abgeleitete Formular Ereignisse bearbeiten, die auch im Basisformular bearbeitet werden, müssen Sie die geerbten Bearbeitungsmethoden mit *inherited* aufrufen, sofern Sie nicht auf sie verzichten wollen (siehe zurückliegendes Code-Beispiel *TForm1.FormKeyPress*).
- Wenn Sie einen Teil der Fuktionalität des abgeleiteten Formulars in das neue Basisformular auslagern und so an andere abgeleitete Formulare weitergeben wollen, können Sie die entsprechenden Methoden der abgeleiteten Klassen in die Basisklasse verschieben.

Komponenten-Templates

Abschließend sei noch einmal auf die Komponenten-Schablonen (Komponenten-Vorlagen, siehe Kapitel 1.6.4) hingewiesen. Sie kommen ohne komplexe Vererbungsmechanismen aus und ließen sich auch wieder im Beispiel des TreeDesigners verwenden, können jedoch als solche nicht als Teil des TreeDesigner-Projekts auf der CD zum Buch mitgeliefert werden, da Kylix die Komponenten-Schablonen in einem eigenen Verzeichnis verwaltet.

Dem TreeDesigner-Projekt fehlt damit jedoch kein wichtiger Teil, denn die einzige sinnvolle Anwendungsmöglichkeit für eine Komponentenschablone in den gezeigten Dialogen wären die drei Schalter, und auch dann dürfte die Komponentenschablone nur die Beschriftung, andere Äußerlichkeiten und sonstige Properties dieser Schalter definieren (z.B. *ModalResult*, *Cancel* und *Default*). Die Ereignismethoden der drei Schalter sind wie erwähnt von der speziellen Arbeitsweise dieser Dialoge im TreeDesigner abhängig und wären daher in einer Komponentenschablone nicht so gut aufgehoben. Ohne die Ereignis-Methoden ließe sich eine solche Dreischalter-Schablone aber leicht in anderen Formularen und Projekten wiederverwenden.

3.7.2 Frames

Die Funktionsweise der Frames entspricht der der Formularvererbung mit dem Unterschied, dass ein Frame nicht ein ganzes Formular, sondern nur einen rechteckigen Ausschnitt daraus an ein Formular weitergibt. Ein Frame wird wie eine (möglicherweise komplexe) Komponente in ein Formular eingefügt und findet sich danach sowohl in der Klassendeklaration des Formulars als auch in der Komponentenliste des Objektinspektors wieder.

Neue Frames entwerfen

Kylix wird ohne fertige Frames ausgeliefert. Das heißt, dass Sie einen Frame nicht so einfach in ein Formular einfügen können, wie Sie ein neues Formular auf Basis der von Kylix vorgegebenen Formularschablonen erstellen können. Sie müssen vielmehr zuerst selbst einen Frame erzeugen.

Und dies funktioniert im Prinzip genauso wie der Entwurf eines ganz normalen Formulars. Einziger Unterschied ist, dass Sie den Frame über den Menüpunkt DATEI | NEUER FRAME erzeugen. Sie erhalten dadurch einen neuen leeren Frame, der zur Entwurfszeit wie ein normales Formular aussieht, sowie eine neue Object-Pascal-Unit mit der zunächst einmal leeren Frame-Klasse. Zur Laufzeit, wenn der Frame als Teil eines Formular verwendet wird, verliert er den im Entwurfsmodus noch angezeigten Rahmen. Die Frame-Klasse in der Pascal-Unit wird statt von *TForm* von *TFrame* abgeleitet. Die Klasse *TFrame* ist gewissermaßen eine Großcousine von *TForm*, denn beide stammen von der Klasse *TScrollingWidget* ab – *TForm* über die Zwischenklasse *TCustomForm*, *TFrame* entsprechend über *TCustomFrame*.

ames verwenden

Der Entwurf, das Schreiben von Ereignismethoden und das Speichern der Frame-Dateien läuft so ab wie bei Formularen. Neuigkeiten gibt es erst, wenn Sie einen bestehenden Frame verwenden wollen. Dazu wählen Sie aus der Komponentenpalette den Schalter mit dem Hinweistext FRAMES (erster Schalter auf der Standard-Seite der Palette) und zeichnen den Umriss des Frames in das Formular ein (dieses Formular wird im Folgenden auch als das *Besitzer-Formular* bezeichnet). Der Frames-Schalter ist sozusagen eine Pseudokomponente, denn erst nach dem Einzeichnen des Umrisses wählen Sie die Klasse des Frames aus, den Sie einfügen wollen. Hierzu zeigt Ihnen Kylix automatisch eine Liste aller zur Verfügung stehenden Frames an.

Wenn Sie den Umriss des Frames zu klein wählen, wird der Frame soweit notwendig mit einem oder zwei Scrollbars ausgestattet, über die Sie die unsichtbaren Bereiche erreichen können (wobei man normalerweise wohl eher geneigt ist, den Umriss nachäglich so einzustellen, dass die Scrollbars wieder verschwinden).

ererbungsmechanismen

Es gibt einen wesentlichen Unterschied zwischen der Benutzung eines Frames und der einer Komponente, die andere Komponenten enthält. Er liegt darin, dass Sie im Formular noch auf die einzelnen Komponenten des Frames zugreifen können, während Sie die Komponente im Formulareditor nur als ganzes Objekt wählen können.

Dabei greifen dieselben, schon im Zusammenhang mit der Formularvererbung in Kapitel 3.7.1 beschriebenen Mechanismen, die im Folgenden noch einmal zusammengefasst sind. Auch im Zusammenhang mit den Frames können Sie:

- die Properties im Objektinspektor ändern (überschreiben), insbesondere die Position und Größe der Komponenten, die Sie natürlich mit der Maus wie gewohnt einstellen können.
- die Ereignisse der Komponenten bearbeiten. Kylix erzeugt automatisch Ereignismethoden mit einem Aufruf von *inherited*, also einem Aufruf der geerbten Ereignisbearbeitung, den Sie nach Wunsch durch Löschen unterbinden können.
- den Frame selbst verändern. Änderungen wirken sich sofort auf alle Formulare, in denen der Frame enthalten ist, aus, sofern sie dort nicht überschrieben wurden.
- das Überschreiben von Properties rückgängig machen, indem Sie den Wert im Objektinspektor auf den geerbten Wert zurücksetzen oder die entsprechende Angabe aus der Textdarstellung des Frames löschen.
- mehrstufige Vererbung verwenden, also z. B. Frames in Frames oder Frames in abgeleiteten Formularen.

Schließlich sei noch darauf hingewiesen, dass auch ein Frame selbst (wie ein Formular) über eigene Properties und Ereignisse verfügt und dass Sie Frames wie Formulare in der Objektablage speichern können.

Vorteile gegenüber der Formularvererbung

Richtig eingesetzt wirkt sich die Verwendung von Frames positiv auf die Struktur einer Anwendung aus. Die folgenden Vorteile sind beispielsweise durch die Formularvererbung nicht oder nur schwer zu erreichen:

- Es können mehrere Frames in ein Formular eingefügt werden, aber ein Formular kann immer nur von einem Formular erben (oder man muss mehrstufige Vererbung verwenden).
- Frames erlauben es, den Code für verschiedene Formularbereiche zu trennen und in verschiedenen Units zu verwalten.
- Durch Frames lassen sich alternative Implementierungen von Formularbereichen leicht austauschen. Sie benötigen dazu lediglich mehrere Frames mit der gleichen Schnittstelle (z. B. denselben Steuerelementen für den Benutzer oder denselben Properties und Methoden, die vom Besitzer-Formular aufgerufen werden können) und können dann in einem Formular, das eines dieser Frames verwendet, ein Frame mit einer anderen Implementierung einsetzen, ohne sonst noch etwas am Formular ändern zu müssen.

- Die Namen der im Frame enthaltenen Komponenten haben einen eigenen Namensraum (den der Frame-Klasse) und können daher nicht mit den Namen der Komponenten des Formulars in Konflikt geraten.

Auch gegenüber den anderen erwähnten Techniken zur Zusammenfassung mehrerer Komponenten weisen Frames Vorteile auf:

- Gegenüber einer Komponente, die mehrere Komponenten zusammenfasst, gibt es bei den Frames keine Kapselung der im Frame enthaltenen Komponenten. Was aus Sicht des OOP als Nachteil angesehen werden kann, erweist sich in der Praxis als Steigerung der Flexibilität, denn Sie können alle im Frame enthaltenen Komponenten auch im Besitzer-Formular des Frame noch ändern.
- Komponenten-Schablonen bieten zwar dieselbe Flexibilität wie die Frames, was die nachträgliche Veränderung der Einzelkomponenten im Formular angeht, ihnen fehlt aber die den Einzelkomponenten übergeordnete Klasse und die der Strukturierung dienende Unit.

Da Frames so etwas Ähnliches wie eine Verallgemeinerung der Formularvererbung darstellen, kann man lange erfolglos nach Nachteilen gegenüber dieser Vererbung suchen. Im Wesentlichen bestimmt das Einsatzgebiet, ob man die Formularvererbung braucht, denn dieser bleibt noch der wichtige Einsatzzweck vorbehalten, die Grundeinstellungen von Formularen zu definieren und zu vererben. Alle Property-Einstellungen und Ereignismethoden, die dem Formular selbst gehören, lassen sich eben nicht in einem Frame festhalten, sondern nur über eine Formularschablone. In bestimmten Fällen lassen sich ja beide Techniken auch kombinieren: Das leere Formular lässt sich, vielleicht mit einigen grundlegenden Aktionsschaltern, von einer Formularschablone ableiten. Im Formular enthaltene Komponentengruppen, die auch in anderen Formularen vorkommen, können dann über Frames realisiert werden (siehe Beispiel des TreeDesigners).

ames im TreeDesigner

Der in Abbildung 3.13 gezeigte Frame dient im TreeDesigner als Grundlage für den in Kapitel 3.7.1 beschriebenen Basisdialog zur Einstellungen von Grafikelement-Attributen. Er enthält ein Eingabefeld für die Beschriftung des Grafikelements sowie vier Felder für die Koordinaten des umgebenden Rechtecks. Außerdem enthält er noch einen Schalter, mit dem die Darstellung der Koordinaten zwischen *(x1/y1, x2/y2)* und *(x1/y1, Breite/Höhe)* gewechselt werden kann. Damit gibt dieser Schalter dem Frame einen guten Anlass, zu demonstrieren, dass ein Frame nicht nur aus Komponenten, sondern auch aus Programmlogik bestehen kann, denn die Eingaben in die Felder x2/y2 sollen ja durch das Programm in entsprechende Breiten- und Höhenangaben umgewandelt werden.

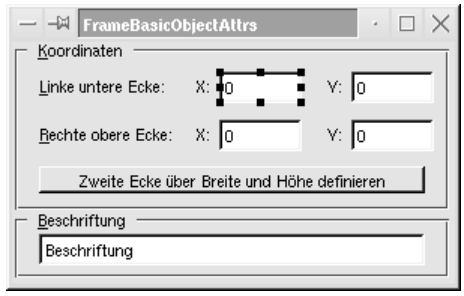


Abbildung 3.14: Der Frame für die Dialoge aus den Bildern 3.15 und 3.16 zur Entwurfszeit

Ändert der Benutzer beispielsweise den Wert *X2* (RECHTE OBERE ECKE X), wird die folgende Methode ausgeführt:

```
procedure TFrameBasicObjectAttrs.EditX2Change(Sender: TObject);
begin
  EditBreite.Number := EditX2.Number - EditX1.Number;
  { Die Breite des Objekts versteht sich inklusive der
    X1-Koordinate, aber exklusiv der X2-Koordinate }
end;
```

Außerdem enthält der Frame noch die bereits in Kapitel 3.7.1 verwendeten Methoden *ApplyChanges* und *CancelChanges*. Das folgende Listing der Frame-Klasse ist lediglich um die Variablen für die vielen Komponenten gekürzt:

```
type
 TFrameBasicObjectAttrs = class(TFrame)
   GroupBox1: TGroupBox;
   ...
   procedure SwitchRelAbsClick(Sender: TObject);
   procedure EditX2Change(Sender: TObject);
   procedure EditY1Change(Sender: TObject);
   procedure EditBreiteChange(Sender: TObject);
   procedure EditHoeheChange(Sender: TObject);
 private
   FEditObject: TGraphicElement;
   procedure SetEditObject(const Value: TGraphicElement);
 public
   property EditObject: TGraphicElement read FEditObject
                            write SetEditObject;
   procedure ApplyChanges;
   procedure CancelChanges;
 end;
```

TFrameBasicObjectAttrs stellt sich also im Prinzip als eine ganz normale Object-Pascal-Klasse dar, deren Methoden im Besitzer-Formular aufgerufen werden können. Aufrufe von *ApplyChanges* und *CancelChanges* waren ja bereits in früheren Code-Auszügen zu sehen. Daher soll nun die Methode *ApplyChanges* das TreeDesigner-Beispiel für dieses

Kapitel abschließen. Sie schreibt die Daten der Steuerelemente in das bearbeitete Grafikelement (Property *EditObject*). Das Besitzer-Formular des Frames ruft sie als Reaktion auf den Schalter ANWENDEN auf.

```
procedure TFrameBasicObjectAttrs.ApplyChanges;
var
  R: TRect;
begin
  if Assigned(FEditObject) then begin
    R.Left := EditX1.Number;
    R.Top := EditY1.Number;
    R.Right := EditX2.Number;
    R.Bottom := EditY2.Number;
    FEditObject.Text := Beschriftung.Text;
    FEditObject.SetNewRect(R);
  end;
end;
```

4 Weitere Klassen der CLX

Dieses Kapitel setzt die in Kapitel 3 begonnene Behandlung der CLX fort, beschäftigt sich aber nicht mehr in erster Linie mit den visuellen Komponenten, sondern mit anderen wichtigen Klassen sowie mit weiteren Themen, die mit der CLX zusammenhängen:

- Abschnitt 4.1 stellt die grundlegenden Datenstrukturen *TList* und *TStrings* vor, auf denen viele Properties von CLX-Klassen basieren. Dabei wird mit *THistoryList* eine neue Klasse von *TStringList* abgeleitet, die später zum Anhängen von Dateilisten an Menüs (Kapitel 4.6.2) und zur Implementierung der visuellen *THistoryCombo*-Komponente verwendet wird (Kapitel 8.5.3).
- Abschnitt 4.2 beschreibt, wie Anwendungen Einstellungs- und Konfigurationsdaten in Textformat-Dateien speichern können. Die CLX unterstützt solche Dateien mit der Klasse *TMemIniFile*.
- In Abschnitt 4.3 geht es um CLX-Klassen, die bei der Speicherung von beliebigen Daten in Dateien behilflich sind: *TStream*, *TReader* und *TWriter*. Weitere Themen sind die Speicherung polymorpher Objekte, für die einige Hilfsfunktionen der CLX benötigt werden, sowie die nicht in Dateien, sondern im Hauptspeicher liegenden Memory-Streams.
- Zentrale Klasse für die in Abschnitt 4.4 behandelte Grafikausgabe ist *TCanvas*, die eine Zeichenfläche repräsentiert und zahlreiche Methoden für die Grafikausgabe bereitstellt. Auch Zeichenwerkzeuge wie *TPen* und *TBrush* und besitzergezeichnete Komponenten werden in diesem Abschnitt erläutert.
- Abschnitt 4.5 behandelt die Klassen *TBitmap*, *TIcon*, *TDrawing* und *TPicture*. Objekte dieser Klassen enthalten bereits eine komplette Grafik und können sich selbst auf eine *TCanvas*-Zeichenfläche ausgeben.
- Abschnitt 4.6 zeigt, dass Sie die Flexibilität der Menüklassen der CLX, die Ihnen der Menü-Designer zur Entwurfszeit bereitstellt, zur Laufzeit an den Benutzer weitergeben können. Auch die eng mit Menüs in Verbindung stehenden Aktionslisten werden erläutert. Außerdem stellt dieser Abschnitt Ihnen eine Komponente vor, durch deren Einbindung Sie jeder Anwendung mit Hauptmenü oder Aktionsliste zu einem Tastenkürzeleditor verhelfen können, der eine geänderte Tastaturbelegung auch zwischen den Programmläufen speichern kann.

- In Abschnitt 4.8 geht es um die allgemeinen Multitasking-/Multithreading-Grundlagen sowie um die Klasse *TThread*, mit deren Hilfe Sie mehrere Programmteile innerhalb einer Anwendung quasi gleichzeitig ablaufen lassen können.
- Abschnitt 4.7 stellt schließlich die Klasse *TClipboard* vor und zeigt, wie Sie damit Ausschneiden- und Einfügen-Operationen in Kylix-Anwendungen vornehmen können.

4.1 Grundlegende Datenstrukturen: TList und TStrings

TList und *TStrings* sind die wichtigsten Klassen der CLX (sie gehört in den Bereich der *BaseCLX*), die sich mit der Datenverwaltung außerhalb von Datenbanken befassen. Besonders *TStrings* wird von den Komponenten der CLX häufig eingesetzt, z.B. in den Properties *TListBox.Items*, *TMemo.Lines* und *TScreen.Fonts*.

4.1.1 TStrings

Aufgabe der Klasse *TStrings* ist es, eine Liste von Strings zu verwalten, wobei Sie jeden String optional mit einem beliebigen Objekt verknüpfen können.

TStrings und TStringList

Die Klasse *TStrings* ist abstrakt und beschreibt nur die Schnittstelle, über die Sie in abgeleiteten Klassen auf die Strings zugreifen. Diese Schnittstelle besteht aus einigen Properties und einer Reihe von Methoden, die in *TStrings* teilweise abstrakt sind. Da die Verwaltung der Strings in dieser Klasse noch nicht implementiert ist, stellt *TStrings* nicht unbedingt eine String*liste*, sondern lediglich eine Sammlung von Strings dar, in der jeder String über einen Index ansprechbar ist.

Die wichtigste von *TStrings* abgeleitete Klasse ist *TStringList*. Sie überschreibt die abstrakten Methoden von *TStrings* und implementiert die Verwaltung der Strings unter Zuhilfenahme der Klasse *TList*. Zu ihren besonderen Fähigkeiten gehören das Abfangen von doppelten Strings (Property *Duplicates*), das automatische Sortieren der Liste (Property *Sorted*) und zwei Ereignisse, mit denen Sie sich über Änderungen informieren lassen können (*OnChange* und *OnChanging*). Wir beschränken uns in diesem Kapitel auf die allgemeinen Fähigkeiten, die in der Klasse *TStrings* definiert sind.

TStrings-Properties

Während *TStrings* über 20 Methoden besitzt, kommt sie mit sehr wenigen Properties aus, die Zugriff auf die Strings sowie die damit verbundenen Objekte gewähren und die Anzahl der Strings angeben (alle Array-Properties beginnen beim Index 0):

Property	Typ	Beschreibung
Count	Integer	Anzahl der gespeicherten Strings
Objects[Integer]	TObject-Array	mit den Strings assoziierte Objekte
Values[String]	String-Array	Wenn die Stringliste Strings der Form »Schlüssel=Value« enthält, finden Sie mit diesem Property den *Value* zu einem Schlüssel, wenn Sie diesen Schlüssel als Index für *Values* verwenden. Ein Beispiel finden Sie in Kapitel 4.2.1 unter *Lesen von kompletten Sektionen*.
Strings[Integer]	String-Array	gibt Zugriff auf die einzelnen Strings und ist das *default*-Property der Klasse.
DelimitedText	String	fasst alle Strings zu einem einzigen String zusammen, aus dem sich die einzelnen Strings eindeutig wiederherstellen lassen (z.B. wenn Sie den zusammengefügten String später wieder an das *DelimitedText*-Property zuweisen). Dazu werden die einzelnen Strings mit einem Trennzeichen getrennt und, falls sie Leerzeichen enthalten, mit Anführungszeichen versehen. Das Trennzeichen wird durch *Delimiter*, das Anführungszeichen durch *QuoteChar* vorgegeben.
CommaText	String	wie *DelimitedText*, nur mit festeingestelltem Komma als Trennzeichen (Anführungszeichen werden weiter durch *QuoteChar* festgelegt)
QuoteChar	Char	Anführungszeichen, das in *DelimitedText* und *CommaText* zum Einschließen von Strings verwendet wird. Enthält der String selbst bereits ein *QuoteChar*, so wird die in *DelimitedText* und *CommaText* verdoppelt (ähnlich der Hochkommatas in einem Object-Pascal-String).
Delimiter	Char	Trennzeichen für das *DelimitedText*-Property. Standardmäßig ist dies ein Komma, bei einer Liste von Pfaden bietet sich dagegen die Systemkonstante *PathDelimiter* als Alternative an.
Names[Integer]	String-Array	liefert passend zu *Values[i]* den Namen des Schlüssels (also den Stringteil vor dem Zeichen »=« bzw. einen Leerstring, falls kein »=« vorhanden ist).
Text	String	gibt alle Strings in einem einzigen String zurück, wobei die Einzel-Strings durch die in Textdateien verwendete Zeilenende-Markierung (angegeben durch die Systemkonstante *sLineBreak*) getrennt werden. Kann auch beschrieben werden, um alle Strings auf einmal neu zu setzen.

Da *Strings* das Standard-Property ist, können Sie auf einen String sowohl mit *StringListe.Strings[StringIndex]* als auch mit *StringListe[StringIndex]* zugreifen. Unter Zuhilfenahme von *Count* können Sie auf einfache Weise alle Strings der Liste durchlaufen, z.B. wie folgt:

```
for i := 0 to StringListe.Count-1 do StringListe[i] := ...
```

Die CLX definiert viele Properties und Klassen, die ähnlich zu bedienen sind wie *TStrings*, obwohl es sich nicht direkt um ein *TStrings*-Objekt handelt. So können Sie beispielsweise die Untermenüpunkte eines Menüpunkts (*TMenuItem*) mit *MenuePunkt[i]* ansprechen, *MenuePunkt.Count* liefert die Zahl der Menüpunkte. Intern ist *TMenuItem* keine Stringliste, sondern verwendet ein *TList*-Objekt zur Speicherung der Untermenüpunkte.

Assoziierte Objekte *R147*

Oft sind die Strings einer Liste mit anderen Objekten des Programms verknüpft. So önnten Sie in einer Listbox beispielsweise die gerade in Ihrer Anwendung geöffneten Fenster anhand ihrer Titelbezeichnungen aufzählen. Der Benutzer könnte eines dieser Fenster auswählen, um es in den Vordergrund zu holen. Sie würden dann den Titel des ausgewählten Fensters im Listbox-Property *Items.Strings[ItemIndex]* finden. Um jedoch das zugehörige Fenster zu finden, müssten Sie nun alle vorhandenen Fenster nach diesem Titel durchsuchen.

Anders verhält es sich in diesem Beispiel, wenn Sie die zugehörigen Fenster gleich mit in der *TListBox*-Komponente speichern. Wenn Sie die Fenstertitel nicht wie folgt einfügen ...

```
Listbox.Items.Add(Form.Caption);
```

... sondern mit dieser Zeile:

```
Listbox.Items.AddObject(Form.Caption, Form);
```

... speichern Sie zusammen mit dem Fenstertitel, der in der ListBox erscheint, eine Referenz auf das Formular (*Form*). *Form* wird von *AddObject* im *Objects*-Array der Stringliste abgelegt, und zwar unter demselben Index, den der im ersten Parameter angegebene String im *Items*-Array erhält.

Um in der beschriebenen Beispielsituation das Formular festzustellen, das zum gerade gewählten Listeneintrag gehört, könnten Sie nun den Ausdruck *TForm(ListBox.Items.Objects[Listbox.ItemIndex])* verwenden. Da *Objects* das Formular nicht als Typ *TForm*, sondern als *TObject* zurückgibt, ist hier eine explizite Typenumwandlung erforderlich.

Sie können auch nachträglich Objekte mit den Strings verknüpfen, indem Sie sie selbst in das *Objects*-Array schreiben, beispielsweise mit

```
with Listbox.Items do
  Objects[IndexOf(Form.Caption)] := Form;
```

... wobei die Methode *IndexOf* und weitere nützliche Methoden von *TStrings* in der ächsten Tabelle erläutert werden.

Wenn Sie auf diese Weise Objekte mit den Strings verbinden, behalten Sie weiter die volle Kontrolle über diese Objekte: *TStrings* verändert diese in keiner Weise. Wenn Sie den zugehörigen String löschen, gibt *TString* das verknüpfte Objekt nicht frei, sondern entfernt lediglich die Referenz zu ihm aus seinem *Objects*-Array.

Ein Praxisbeispiel zum Ausprobieren finden Sie im Projekt *AlarmClock4* von Kapitel 1.8.4.

Weitere TStrings-Methoden

Durch die Anwendung von einfachen Methoden können Sie noch viele weitere Aufgaben mit einem *TStrings*-Objekt durchführen. Die Parameter dieser Methoden zeigen und erklären sich zum größten Teil bereits durch die *Code-Parameter*-Funktion des Kylix-Editors, daher sind sie im Folgenden nicht extra erwähnt. Zuerst einmal gibt es natürlich grundlegende Funktionen, die man meistens von einer Liste erwartet:

- Einzelne Strings der Liste löschen Sie mit *Delete*, vertauschen Sie mit *Exchange* und verschieben Sie mit *Move*.
- Um alle Strings zu löschen, rufen Sie *Clear* auf.
- Um Strings an einer bestimmten Position einzufügen, verwenden Sie statt *Add* und *AddObject* die Methoden *Insert* und *InsertObject*.
- Zum Suchen von Einträgen dienen die Methoden *IndexOf* (sucht anhand des String-Textes), *IndexOfObject* (sucht anhand des assoziierten Objekts) und *IndexOfName* (sucht in Strings, die nach dem Muster »Name = Wert« aufgebaut sind, siehe Property *Names*).

Des Weiteren verfügt *TStrings* über eine Reihe schöner »fortgeschrittener« Funktionen:

- Wenn Sie gleich zwei *TStrings*-Kollektionen haben, können Sie sie mit *Equals* vergleichen, mit *AddStrings* aneinander hängen oder mit *Assign* abgleichen.
- Wenn der Inhalt des *TStrings*-Objekts in einer visuellen Komponente angezeigt wird, können Sie am Anfang einer längeren Aktualisierungsoperation (beispielsweise Aufrufe von *Add* und *Delete*) die Methode *BeginUpdate* aufrufen, damit nicht alle Änderungen am Bildschirm langatmig nachvollzogen werden. Wenn die Stringliste fertig ist, erreichen Sie mit *EndUpdate*, dass die visuelle Komponente einmal und von nun an wieder regelmäßig aktualisiert wird.
- Dateioperationen: Mit *LoadFromFile* laden Sie den Inhalt einer Textdatei in das *TStrings*-Objekt. Dabei wird jede Zeile der Datei zu einem String, und wenn Sie den Inhalt des *TStrings*-Objekts nachher über das Property *Text* abfragen, erhalten Sie wieder genau den Inhalt der Textdatei. Entsprechend speichern Sie die Stringliste mit der Methode *SaveToFile*. Diese Methoden sind von weit reichender Bedeutung,

da mit ihnen der Inhalt der verschiedensten Komponenten gespeichert werden kann, wie etwa *TListBox.Items* und *TMemo.Lines* (siehe z.B. die Speicherung des Listbox-Inhalts in Kapitel 1.8.3).

- Alternativ zu den letztgenannten Methoden können Sie auch *LoadFromStream* und *SaveToStream* verwenden, wenn Sie die Stringliste als Teil eines Streams speichern wollen, in dem sich noch andere Daten befinden.

4.1.2 Ableiten einer History-Liste von TStringList

Ein kleines Anwendungsbeispiel gibt die Klasse *THistoryList*, die einerseits in Kapitel 8.5.3 bei der Programmierung der visuellen *THistoryCombo*-Komponente verwendet, andererseits in Kapitel 4.6.2 um Menüfunktionen erweitert wird. Der Kern dieser Klasse wird an dieser Stelle beschrieben.

History-Listen *R64*

THistoryList ist von *TStringList* abgeleitet und wird damit zu einer spezialisierten Stringlisten-Klasse. Die Spezialisierung liegt darin, dass *THistoryList*

- eine Längenbegrenzung für die Liste einführt,
- diese Längenbegrenzung bei Hinzufügen von Strings mit der Methode *AddString* überprüft und gegebenenfalls den ältesten String der Liste entfernt
- und dass die Methode *AddString* einen String aus der Mitte der Liste entfernt, wenn er am Anfang erneut eingefügt wird.

iele Dialogboxen der Kylix-IDE geben Beispiele für History-Listen, die sich genauso verhalten, beispielsweise die aufklappbare Liste im *Suchen*-Dialog: Eine neue Eingabe verschiebt die alten Eingaben eine Listenposition nach unten, die Auswahl einer Eingabe aus der Liste fügt diese Eingabe wieder an den Anfang der Liste, und wenn die Liste eine gewisse Größe erreicht, fallen die letzten Strings am Ende heraus.

Die vereinfachte Deklaration und der Konstruktor von *THistoryList* sehen wie folgt aus:

```
THistoryList = class(TStringList)
(* Die THistoryList-Funktionen, die mit Menüs zusammenhängen,
   werden an anderer Stelle beschrieben. *)
private
  FMaxLen: Integer;
  procedure SetMaxLen(AnInt: Integer);
public
  constructor Create;
  property MaxLen: Integer read FMaxLen write SetMaxLen;

  procedure AddString(s: string);
```

```
    procedure SaveToIni(IniName, IniSection: string);
    procedure LoadFromIni(IniName, IniSection: string);
  end;

constructor THistoryList.Create;
begin
  inherited Create;
  FMaxLen := 10;
end;
```

Die Methode *AddString* verwendet die von *TStringList* geerbten Elemente *IndexOf*, *Delete*, *Insert* und *Count*, um die oben beschriebene Funktionalität zu gewährleisten:

```
procedure THistoryList.AddString(s: string);
var
  OldIndex: Integer;
begin
  { String schon vorhanden? }
  OldIndex := IndexOf(s);
  if OldIndex <> -1 then
    Delete(OldIndex); { dann das Duplikat löschen }
  Insert(0, s); { auf alle Fälle an erster Stelle einfügen }
  { Maximallänge überschritten? }
  if Count > FMaxLen then
    Delete(Count-1); { letzten Eintrag löschen }
  { Menü-Methoden, siehe Kapitel 4.6.2: }
  RemoveMenuItems; { Menü in Ursprungszustand versetzen... }
  AddMenuItems; { ...und alle Einträge neu anhängen. }
end;
```

Da das Property *MaxLen* jederzeit änderbar ist, muss die zugehörige Schreibmethode bei einer solchen Änderung gegebenenfalls die Länge der Liste anpassen:

```
procedure THistoryList.SetMaxLen(AInt: Integer);
begin
  { nur Maximallängen zwischen 1 und 100 akzeptieren: }
  if (AInt >= 1) and (AInt <= 100) then begin
    FMaxLen := AInt;
    while Count > FMaxLen do
      Delete(Count-1); { alle überzähligen Einträge löschen }
  end;
  ... Menü-Aktualisierung ...
end;
```

Zum Speichern und Laden der Liste bedient sich *THistoryList* der Klassen *TIniFile* und *TRegistryIniFile*, daher werden die Methoden *SaveToIni* und *LoadFromIni* sowie das Property *UseRegistry* erst in Kapitel 4.2.2 erläutert.

4.1.3 TList

TList ist direkt von *TObject* abgeleitet; und da sie eine wichtige Grundlage für die interne Implementierung der Klasse *TStringList* liefert, verwundert es nicht, dass ihre Methoden und Properties denen der Stringlisten ähneln.

Wichtigster Unterschied zu *TStrings*/*TStringList* ist der Typ der Daten, die *TList* speichert: *TList* erwartet nämlich *keinen* Typ, sondern lediglich einen untypisierten Zeiger (siehe *Pointer* in Kapitel 2.4.6) auf ein beliebiges Objekt, so dass Sie nicht nur Objekte, sondern auch einfache Daten wie Strings in einer *TList*-Liste speichern können. Zahlen können Sie sogar direkt darin speichern, indem Sie sie als *LongInt* ansehen und dann diesen *LongInt* in einen *Pointer* umwandeln, den Sie dem *TList*-Objekt übergeben (dieser *Pointer* ist dann als Zeiger natürlich ungültig).

Wenn Sie ein Objekt wieder aus der Liste auslesen, müssen Sie den Typ *Pointer* wieder in den ursprünglichen Typ zurückverwandeln, und zwar funktioniert hier nur die normale Typenumwandlung (Kapitel 2.4.7), nicht aber die Umwandlung mit dem *as*-Operator (Kapitel 2.3.4). Ein Beispiel dafür gibt die von *TList* abgeleitete Klasse *TGraphicDoc* in Kapitel 6.3.2.

Properties und Methoden

Die Elemente der Liste können Sie über das Property *Items* ansprechen, das bis auf den Typ der Elemente wie das *Strings*-Property von *TStrings* funktioniert. Entsprechend gibt es auch hier ein Property *Count* für die Anzahl der gespeicherten Elemente.

Die meisten Methoden heißen so wie entsprechende Methoden von *TStrings* und arbeiten auch so, deshalb seien sie hier nur aufgezählt: *Add*, *Clear*, *Delete*, *Exchange*, *IndexOf*, *Insert*, *Move* und *Remove* (die Auswahl der Methoden ist natürlich deutlich kleiner als bei *TStrings*).

Die zusätzlichen Methoden *First*, *Last*, *Remove* und *Expand* werden Sie wahrscheinlich selten benötigen, daher sei an dieser Stelle auf die Online-Hilfe verwiesen.

Interne Funktionsweise

Auch die Properties *List* und *Capacity* sowie die Methode *Pack* brauchen Sie wahrscheinlich nicht zu verwenden, jedoch geben diese Aufschluss über die interne Funktionsweise von *TList*. Anders als der Name vermuten lassen könnte, sind die Daten in *TList* nämlich nicht in einer per Zeiger verknüpften Liste, sondern in einem *Pointer*-Array verknüpft.

Wie viele Zeiger dieses Array fassen kann, gibt *Capacity* an. Das Array selbst finden Sie im Property *List*. Falls es vollständig gefüllt sein sollte, wird es beim Hinzufügen eines neuen Elements mit *Add* automatisch erweitert.

In Kapitel 6.3.2 finden Sie ein Beispiel für die Verwendung von *TList* zum Ableiten einer neuen Klasse (der schon erwähnten Klasse *TGraphicDoc*).

4.1.4 Andere Container-Klassen

Während die bisher besprochenen Listenklassen in der zentralen Unit *Classes* definiert werden, gibt es noch einige weitere Listenklassen, die in der vergleichsweise winzigen Unit *Contnrs* untergebracht sind. *Contnrs* stellt einige von *TList* abgeleitete Klassen bereit, die auf spezielle Objekte oder auf kleine Zusatzfunktionen spezialisiert sind. Dieses Kapitel soll nur eine kurze Übersicht geben, da die Klassen unter Nutzung der Online-Hilfe sehr einfach zu verwenden sind.

Während die Elemente von *TList* einfache *pointer* sind, sind die Container-Klassen für konkretere Elemente bzw. Objekte ausgelegt. Diese Objekte könnten Sie zwar auch direkt in einer *TList* speichern, bei der Abfrage eines Listenelements müssten Sie jedoch immer den abstrakten *pointer* in den tatsächlichen Typ umwandeln (z.B. mit *TComponent(List.Items[i])*.

- *TObjectList* ist direkt von *TList* abgeleitet und speichert Instanzen von *TObject* oder davon abgeleiteten Klassen. Sie führt ein neues Property namens *OwnsObjects* ein. Indem Sie dieses auf *True* setzen (was bereits die Voreinstellung ist), weisen Sie die *ObjectList* an, die Objekte automatisch mit *Free* freizugeben, wenn sie aus der Liste gelöscht werden.
- *TComponentList* und *TClassList* verrichten Ähnliches für Listenelemente der Typen *TComponent* und *TClass*.

Einen anderen Vererbungsweg geht *TOrderedList*. Sie ist eine abstrakte Klasse zur Definition von geordneten Listen, auf die mit *Push*, *Pop* und *Peek* zugegriffen werden kann. Methoden wie *Add* und *Delete* sind in dieser abstrakten Definition nicht vorgesehen. *TOrderedList* ist auch nicht von *TList*, sondern von *TObject* abgeleitet und verwendet ein *TList*-Objekt nur intern. Spezielle Ableitungen von *TOrderedList* sind *TStack/TObjectStack* und *TQueue/TObjectQueue*, für die hiermit auf die Online-Hilfe verwiesen wird.

4.2 Textbasierte Konfigurationsdateien

Eine wichtiges Merkmal vieler benutzerfreundlicher Anwendungen ist die Fähigkeit, Einstellungen des Benutzers in Konfigurationsdateien zu speichern, so dass sich der Benutzer nach dem nächsten Programmstart in derselben Umgebung befindet wie vor dem letzten Verlassen des Programms. Unter Linux werden dazu häufig keine Binärdateien, sondern menschenlesbare Textdateien verwendet. Aufgrund seiner Herkunft bringt Kylix bereits eine Klasse mit, die Textdateien im von Windows bekannten INI-Format verwaltet.

4.2.1 Dateien im INI-Format

Kylix selbst liefert einige Beispiele für die Verwendung von Textdateien in diesem Format: Sowohl die globale Optionen wie die Umgebungsoptionen als auch projektspezifische Optionen werden in diesem Format abgelegt. Die globalen Optionen finden Sie im Verzeichnis `<home>/.borland` (z.B. in der Datei `delphi60rc`), die projektspezifischen Optionen im selben Verzeichnis wie die `dpr`-Projektdatei. Hierzu gehören zunächst die Einstellungen des Projektoptions-Dialogs, die in Dateien mit der Endung `.kof` gespeichert werden; und der Aufbau des Desktops findet sich in Dateien mit der Endung `desk` wieder (sofern Sie die Umgebungsoptionen entsprechend eingestellt haben – Seite VORGABEN, Gruppe OPTIONEN FÜR AUTOSPEICHERN). Beide Dateien können Sie in einem normalen Texteditor einsehen und eventuell sogar sinnvoll verändern.

Wenn Sie schon mit Delphi für Windows INI-Dateien geschrieben haben, dann müssen Sie nun unter Kylix vor allem eine Sache beachten: Die Klasse *TIniFile* der CLX ist nur ein Aliasname für die Klasse *TMemIniFile*. Anders als die *TIniFile* der VCL unter Windows, welche das Windows-API verwendet, lädt *TMemIniFile* immer eine komplette INI-Datei in den Speicher und alle Änderungen (Lese- und Schreiboperationen) finden nur im Speicher statt. Die Änderungen selbst können Sie wie unter Windows vornehmen, jedoch müssen Sie manuell dafür sorgen, dass die Änderungen auch wieder in die Datei zurückgeschrieben werden, indem Sie die Methode *TMemIniFile.UpdateFile* aufrufen (wobei Sie diese Methode übrigens ruhig auch unter Windows aufrufen können, um eventuelle von Windows verwendete Speicher-Caches zu umgehen).

Aufbau der INI-Dateien

INI-Dateien sind in mehrere *Sektionen* gegliedert, die jeweils aus Zeilen der Form Schlüssel=Wert« bestehen. Die von *THistoryList* (Kapitel 4.1.2) angelegte Sektion Files« kann beispielsweise wie folgt aussehen:

```
[FILES]
History0=Ellipse.TVF
History1=KlassenHierarchie.TVF
History2=CLX2.TVF
History3=Lines.TVF
```

TMemIniFile

Mit einem Objekt der Klasse *TMemIniFile* können Sie objektorientiert auf die Einträge einer INI-Datei zugreifen. Nachdem Sie die Unit *IniFiles* eingebunden haben, müssen Sie zunächst ein Objekt der genannten Klasse initialisieren, indem Sie deren Konstruktor mit dem Namen der Datei aufrufen:

```
IniFile := TMemIniFile.Create(IniName); // manuelle Freigabe erforderlich
```

Danach können Sie mit Hilfe der Methoden *ReadInteger, ReadString* und *ReadBoolean* sowie mit *WriteInteger, WriteString* und *WriteBoolean* auf die einzelnen Einträge zugreifen. Die Methoden arbeiten nach dem folgenden Muster:

```
Wert := IniFile.Read...(Sektion, Schluessel, Voreinstellung);
Write...(Sektion, Schluessel, Wert);
```

Die *Read*-Methoden geben den Wert des Schlüssels einer bestimmten Sektion zurück. Falls der angegebene Schlüssel nicht existiert, liefern sie den Wert, den Sie im dritten Parameter als Voreinstellung angegeben haben. *ReadInteger* und *ReadBoolean* ersparen es Ihnen, die Zeichenkette aus der Datei in einen Integer- bzw. Boolean-Wert umzuwandeln.

Auch die *Write*-Methoden erhalten in den ersten beiden Parametern den Sektionsnamen und den Schlüssel. Im dritten Parameter geben Sie den Wert an, der als Zeichenkette in die INI-Datei geschrieben wird.

Lesen von kompletten Sektionen *R122*

Nützlich sind auch die *TIniFile*-Methoden *ReadSection* und *ReadSectionValues*. Sie lesen eine komplette Sektion ein und fügen die Einträge in eine als Parameter übergebene Stringliste ein:

```
  ReadSection(IniSection, StringList)
{ Dieser Aufruf füllt StringList nach dem folgenden Muster:
    History0
    History1  usw. }
  ReadSectionValues(IniSection, Stringlist)
{ füllt StringList nach dem folgenden Muster:
    History0=Ellipse.TVF
    History1=TRY.TVF        usw. }
```

Nur mit *ReadSectionValues* erhalten Sie auch den Teil der Einträge nach dem Gleichheitszeichen. In diesem Fall können Sie das *TStrings*-Property *Values* verwenden, um nur diesen Teil abzufragen. So liefert beispielsweise *Stringlist.Value['History0']* den Wert »Ellipse.TVF«.

Im Zusammenhang mit Sektionen sind übrigens auch die Methode *EraseSection*, die eine Sektion komplett löscht, und die Methode *ReadSections*, die eine Liste aller vorhandenen Sektionen erzeugt, erwähnenswert.

4.2.2 Speichern der History-Liste in einer INI-Datei

Als Beispiel zur Verwendung der Klasse *TMemIniFile* dient nun die in Kapitel 4.1.2 vorgestellte History-Liste, die Funktionen zum Speichern und Laden ihres Inhalts zur Verfügung stellt, jedoch nicht nur die einfachen und von *TStringList* geerbten *LoadFromFile*

und *SaveToFile* (diese erzeugen ja immer eine komplette neue Datei), sondern die selbst definierten neuen Methoden *SaveToIni* und *LoadFromIni*, die sich nur auf eine Sektion einer ansonsten unangetasteten Ini-Datei beziehen.

Die Namen der einzelnen Dateneinträge entstehen durch Verknüpfung des Strings History« mit einem Index, so dass sich ein INI-Dateiabschnitt wie in Kapitel 4.2.1 gezeigt ergibt. Das folgende Listing zeigt die Methode zum Speichern der Liste:

```
procedure THistoryList.SaveToIni(IniName, IniSection : string);
var
  i : integer;
  IniFile : TIniFile;
begin
  IniFile := TIniFile.Create(ExcludeTrailingPathDelimiter(IniName));
  try
    for i:=0 to Count-1 do
      IniFile.WriteString(IniSection, 'History'+IntToStr(i), Strings[i]);
  finally
    IniFile.UpdateFile; // ist hier in Wirklichkeit eine TMemIniFile!
    IniFile.Free;
  end;
end;
```

Hinweis: Eine kleine Verbesserung der obigen Methode könnte darin bestehen, dass überzählige Einträge in der INI-Datei gelöscht werden. Überzählige Einträge gibt es, falls die INI-Datei mehr Einträge enthält als der aktuelle Listeninhalt. Nach dem Aufruf von *SaveToIni* würden ohne die Verbesserung nämlich alte Einträge in der INI-Datei stehen bleiben und beim nächsten Laden wieder gelesen. Die Liste wäre dann nicht mehr dieselbe wie beim Speichern.

Entsprechend liest *LoadFromIni* die Einträge mit *ReadString* wieder ein. Dabei kann es sein, dass weniger als *MaxLen* Strings in der INI-Datei gespeichert sind. In diesem Fall liefert *ReadString* für die letzten Strings den als Voreinstellung angegebenen Leerstring zurück:

```
{ Ausschnitt aus THistoryList.LoadFromIni }
  for i := 0 to FMaxLen-1 do begin
    s := IniFile.ReadString(IniSection, 'History'+IntToStr(i), '');
    if s <> '' then
      Add(s);
  end;
```

Damit ist alles Wichtige zur Beispielklasse *THistoryList* gesagt. Ein Wiedersehen mit dieser Klasse wird es in Kapitel 8.5.3 geben, denn dort wird sie in die Komponente *THistoryCombo* eingebaut.

4.2.3 Die TStateSaver-Komponente

Es gibt sehr viele kleine Anwendungsmöglichkeiten für Konfigurationsdateien, die nicht genutzt werden, weil das Implementieren des Codes zum Speichern und Wiederherstellen der Einstellungen zu umständlich ist. Dies gilt besonders für Software-Entwickler, die während der Programmentwicklung ein bestimmtes Programm ständig neu starten und zum Testen einer bestimmten Funktion immer wieder dieselben Einstellungen im Programmfenster vornehmen müssen, z.B. bestimmte Markierungsschalter wählen, einen Text eingeben, ein Verzeichnis aus dem tief strukturierten Verzeichnisbaum der Festplatte auswählen usw. Mit Kylix können Sie sich hier zwar teilweise Erleichterung verschaffen, indem Sie die häufig benötigten Einstellungen bereits zur Entwurfszeit im Formular vornehmen. Doch wenn Sie sich dann während eines weiteren Testlaufs dazu entschließen, die Einstellungen zu ändern, dann hat das keine Auswirkungen auf die Entwurfszeit-Einstellungen und beim nächsten Programmstart werden wieder die alten Einstellungen verwendet.

TStateSaver *R120*

In der Komponentenbibliothek dieses Buches finden Sie mit *TStateSaver* eine Komponente, die hier Abhilfe schaffen soll. Es genügt, ein Exemplar dieser nicht-visuellen, also zur Entwurfszeit unsichtbaren Komponente auf ein Formular abzulegen und die folgenden Einstellungen vorzunehmen:

- In *ResFileName* geben Sie den Namen einer Konfigurationsdatei ein, sofern Sie nicht mit der Voreinstellung zufrieden sind.
- In *Section* geben Sie an, in welchem Abschnitt der INI-Datei die automatische Speicherung stattfinden soll. Sie können also in *ResFileName* eine Datei angeben, die Sie an anderer Stelle auch manuell mit *TIniFile* bearbeiten. Durch die eigene Sektions-Überschrift sind alle von *TStateSaver* automatisch gespeicherten Daten von Ihren manuell gespeicherten Daten klar getrennt.

  In *UseHomeDirectory* legen Sie fest, ob die INI-Datei im Home-Verzeichnis des aktuellen Benutzers abgelegt werden soll. Ist dieses Property *False*, wird als Speicherort das Verzeichnis der ausführbaren Datei verwendet.

Dies genügt bereits, damit Sie eine automatische Speicherung der folgenden Properties von Komponenten Ihres Formulars erhalten: *T(Custom)Edit.Text*, *TSpinEdit.Value*, *TMaskEdit.Text*, *TTrackBar.Position*, *TCheckBox.Checked*, *TRadioButton.Checked*, *TRadioGroup.ItemIndex*, *TTabControl.TabIndex*, *TPageControl.ActivePageIndex*. All diese Eigenschaften werden beim nächsten Programmstart automatisch wieder hergestellt. Die Liste dürfte wohl die am häufigsten verwendeten Komponenten mit ihren wichtigsten zur Laufzeit veränderten Properties enthalten, jedoch können Sie die Liste auch leicht auf Ihre speziellen Erfordernisse erweitern.

TStateSaver erweitern

Hierzu genügt es nämlich, die *TStateSaver*-Methoden *SaveState* und *RestoreState* zu erweitern. Zur Illustration sei die *SaveState*-Methode (etwas gekürzt) wiedergegeben, die *RestoreState*-Methode unterscheidet sich im Prinzip nur durch die Verwendung von *Read...*- statt *Write...*-Methoden und soll daher unabgedruckt auf der CD verbleiben:

```
procedure TStateSaver.SaveState;
var
  i: Integer;
  IniFile: TMemIniFile;
  C: TComponent;
begin
  if Owner <> nil then begin
    IniFile := TMemIniFile.Create(GetFullIniPath);
    try
      if SaveWindowState and (Owner is TCustomForm) then
        SaveWindowPos(IniFile, FSection, Owner as TCustomForm);
      for i := 0 to Owner.ComponentCount - 1 do begin
        C := Owner.Components[i];
        if C.Name = '' then continue; // weiter mit nächstem Fall
        if FExcludeList.IndexOf(C.Name) <> -1 then continue;
        if C is TCustomEdit then with C as TEdit do
          IniFile.WriteString(FSection, Name, Text)
        else if C is TTrackBar then with C as TTrackbar do
          IniFile.WriteInteger(FSection, Name, Position)
        else if C is TSpinEdit then with C as TSpinEdit do
          IniFile.WriteInteger(FSection, Name, Value)
        else if C is TRadioGroup then with C as TRadioGroup do
          IniFile.WriteInteger(FSection, Name, ItemIndex)
        ...
        ... (ein Fall pro unterstützter Komponentenklasse) ...
        ...
        else if C is TCheckBox then with C as TCheckBox do
          IniFile.WriteInteger(FSection, Name, Integer(Checked))
      end;
    finally
      IniFile.UpdateFile;
      IniFile.Free;
    end;
  end;
end;
```

Um dies z.B. durch eine von Ihnen gewählte Komponente *TMyControl* zu erweitern, ügen Sie einfach einen neuen Fall in *SaveState* ein:

```
...
else if C is TMyControl then with C as TMyControl do
   IniFile.WriteType(FSection, Name, PropertyName)
```

Analog dazu gilt es, in die hier nicht abgedruckte Methode *RestoreState* Folgendes einzufügen:

```
else if C is TMyControl then with C as TMyControl do
   TabIndex := IniFile.ReadType(FSection, Name, PropertyName)
```

Dabei setzen Sie für *PropertyName* den Namen des Properties ein, das gespeichert werden soll, und entsprechend seines Typs passen Sie auch den Aufruf Write*Type* an.

Funktionsweise

Das Listing von *SaveState* enthüllt bereits das Funktionsprinzip von *TStateSaver*: Da die Komponente in das Formular eingefügt wird, wird das Formular zu ihrem Besitzer und ist folglich aus der Sicht der Komponente über das *Owner*-Property anzusprechen. Über das Formular erhält *TStateSaver* nun Zugriff auf alle darin enthaltenen Komponenten. *SaveState* muss also nur das *Components*-Array durchlaufen und den Typ der diversen Komponenten testen. Falls es sich um einen unterstützten Typ handelt, wird dafür ein Eintrag in der INI-Datei erzeugt, dessen Name sich aus dem Namen der Komponente aus dem *Components*-Array ableitet (theoretisch könnten auch mehrere Einträge für mehrere Properties der Komponente erzeugt werden). Das Ergebnis von *SaveState* könnte beispielsweise so aussehen (natürlich ohne die Kommentare):

```
[StateSaverAutoSave] // <- Name im ResFileName-Property
Edit1=tempfilename.xyz // TEdit1.Text wurde gespeichert
CheckBox1=1            // TCheckBox.Checked als Zahl
TrackBarIterations=40  // TTrackBar.Position
```

Der automatische Aufruf der *SaveState*- und *RestoreState*-Methoden kommt wie folgt zustande: *RestoreState* wird in der virtuell überschriebenen Methode *Loaded* aufgerufen, die von der CLX aufgerufen wird, wenn die Komponente aus einer Formulardatei geladen wurde (siehe Kapitel 8.4.2), zum Aufruf der *SaveState*-Methode klinkt sich die Komponente in das *OnClose*-Ereignis des Formulars ein. Sie verknüpft dieses mit der folgenden Ereignisbearbeitungsmethode:

```
procedure TStateSaver.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  if Assigned(FOldCloseEventMethod) then
    FOldCloseEventMethod(Sender, Action);
  SaveState;
end;
```

Darin wird auch die vor der Verknüpfung zwischengespeicherte alte Bearbeitungsmethode für das *OnClose*-Ereignis (falls Sie also zur Entwurfszeit auch eine Methode für *OnClose* geschrieben haben) aufgerufen. Zu Konflikten zwischen Ihrer *OnClose*-Methode und der StateSaver-*OnClose*-Methode kommt es nur dann, wenn Sie *OnClose*

zur Laufzeit verändern wollen. Bei Verwendung von *TStateSaver* sollten Sie dies also vermeiden (oder Sie müssen die Automatik von *TStateSaver* abschalten und *TStateSaver.SaveState* manuell beim Schließen des Formulars aufrufen).

Weitere Properties von TStateSaver *R121*

StateSaver besitzt noch zwei weitere Properties, die zum Abschluss noch erwähnt werden sollen:

- *SaveWindowState* ist ein boolesches Property, in dem Sie die Komponente anweisen können, Formularposition, -größe und -zustand (*WindowState*) ebenfalls zu speichern (im gleichen Abschnitt der INI-Datei wie die anderen Daten; die Namen der Fensterstatus-Einträge beginnen mit »FormState«). Die Wiederherstellung findet beim nächsten Programmstart automatisch statt.
- *ExcludeList* ist eine *TStringList*, in der Sie eine Liste von Komponentennamen angeben, die *nicht* gespeichert werden sollen, was immer es für einen Grund dafür geben mag (die Komponente wird bereits vom Programm mit einem passenden Vorgabewert initialisiert, sie zeigt vertrauliche Informationen an etc.)

4.3 Streams und Ablageobjekte

Die objektorientierte Entsprechung für die in Kapitel 2.8.2 beschriebenen Dateien sind die Streams, deren grundsätzliches Wesen von der abstrakten Basisklasse *TStream* definiert wird.

4.3.1 Stream-Klassen

TStream definiert nur die virtuellen und abstrakten Methoden, mit denen Sie Daten in einem Stream speichern und von dort lesen können. Es ist Aufgabe der von *TStream* abgeleiteten Klassen, zu definieren, *wo* diese Daten gespeichert werden. Die CLX-Unit *Classes* enthält bereits einige konkrete Stream-Typen: *THandleStream*, *TFileStream*, *TStringStream* und *TMemoryStream*.

Da *THandleStream* nur eine Zwischenstufe zu *TFileStream* ist und Sie einen *TMemoryStream* mit einem einfachen *Create*-Konstruktor erstellen können, konzentrieren wir uns hier auf *TFileStream*. Sie öffnen einen solchen Stream und damit eine Datei wie folgt:

```
Stream := TStream.Create(Dateiname, fmOpenRead);
```

Im ersten Parameter geben Sie einen Dateinamen an, bei Bedarf mit Pfad, im zweiten Parameter legen Sie den so genannten Dateimodus fest, der besagt, wie Ihre Anwendung und andere Anwendungen auf die Datei zugreifen können, solange diese geöffnet ist.

Mit *fmOpenRead*, *fmOpenWrite* und *fmOpenReadWrite* erhalten Sie Lese- oder Schreibzugriff auf eine bestehende Datei oder beides gleichzeitig. Wenn Sie eine neue Datei anlegen wollen, die eine eventuell schon bestehende Datei mit dem gleichen Namen überschreibt, verwenden Sie den Modus *fmCreate*. Diese Konstanten können Sie in einer *or*-Verknüpfung mit den Konstanten *fmShareExclusive*, *fmShareDenyWrite* und *fmShareDenyNone* verknüpfen, um anderen Anwendungen jedweden Zugriff auf diese Datei oder nur Lese- bzw. nur Schreibzugriff zu untersagen (z. B. *fmOpenRead or dmShareDenyWrite*).

4.3.2 Der Inhalt von Streams

In Streams können Sie natürlich alle Daten schreiben, die Sie auch in normalen Dateien speichern können, jedoch kennen Streams keinerlei Datentypen. Stattdessen erwarten sie einen beliebigen Variablenparameter, dessen Größe Sie in einem zweiten Parameter angeben. Die typische Speicherung von Daten in einem Stream sieht damit wie folgt aus:

```
Stream.WriteBuffer(Points, sizeof(Points));
Stream.WriteBuffer(FShapeType, sizeof(FShapeType));
Stream.WriteBuffer(FText, length(FText)+1);
```

Entsprechend können Sie die Daten mit *ReadBuffer* wieder lesen. *WriteBuffer* und *ReadBuffer* erzeugen eine *EReadError*- bzw. *EWriteError*-Exception, falls es zu einem Lese- oder Schreibfehler kommt. Beide Methoden gehören noch nicht zu den grundlegendsten Methoden und Properties von *TStream*, die in der folgenden Tabelle zusammengefasst sind:

Typ	Name	Beschreibung
property	Position: LongInt	gibt die aktuelle Lese-/Schreibposition im Stream an.
property	Size: LongInt	gibt die Größe des Streams (bzw. die Dateigröße) in Byte an.
function	Write(var Buffer, Count): LongInt	schreibt *Count* Bytes von *Buffer* in den Stream, Funktionsergebnis ist die Zahl der erfolgreich geschriebenen Bytes.
function	Read(var Buffer, Count): LongInt	liest *Count* Bytes aus dem Stream und speichert sie in *Buffer*, liefert die Zahl der tatsächlich gelesenen Bytes.
function	Seek(Offset, Modus): LongInt	*Modus=soFromBeginning* – setzt die aktuelle Position auf *Offset*, *Modus=soFromCurrent* – bewegt die aktuelle Position um *Offset* Bytes nach vorne oder nach hinten, *Modus=soFromEnd* – positioniert die Schreib-/Lesemarke *Offset* Bytes vom Stream-Ende entfernt.

Alle anderen Lese- und Schreibmethoden bauen auf *Read* und *Write* auf. Zum einen gibt es die schon erwähnten *ReadBuffer* und *WriteBuffer*, die lediglich die Methoden *Read* bzw. *Write* aufrufen und dabei überprüfen, ob deren Ergebnis mit der gewünschten Byte-Anzahl (*Count*) übereinstimmt, und im Fehlerfall die genannten Exceptions erzeugen. Zum anderen dienen die Methoden *WriteComponent(Res)* und *ReadComponent(Res)* dazu, Komponenten innerhalb von Formularen zu speichern, allerdings berufen sich diese Methoden hauptsächlich auf die im nächsten Kapitel besprochenen Hilfsklassen.

Ein vollständiges Beispiel zum Speichern eines Dokuments in einem Stream finden Sie in Kapitel 6.3.3.

4.3.3 Polymorphe Objekte speichern

Mit Hilfe von RTTI (siehe Kapitel 2.3.6), virtuellen Konstruktoren und Klassenreferenzen (jeweils Kapitel 2.3.4) sowie mit einigen Routinen der Unit *Classes* können Sie auch Objekte von beliebigen Klassen so speichern, dass Sie sie beim Lesen komplett rekonsuieren können. Dazu genügt es nicht, nur die Daten des Objekts in die Datei zu schreiben. Damit beim Lesen der richtige Konstruktor aufgerufen werden kann, müssen Sie auch die Klasse des Objekts speichern. Ohne die Neuerungen von Object Pascal önnte das Lesen einer solchen Datei beispielsweise so aussehen:

```
{ Lesen der Objektklasse }
Stream.Read(ClassId, sizeof(ClassId));
{ Erstellen des Objekts }
case ClassId of
  1: NewObject := Class1.Create;
  2: NewObject := Class2.Create;
end;
{ Lesen der Objektdaten }
NewObject.ReadData
```

Schon in Kapitel 2.3.4 konnte eine derartige Fallunterscheidung durch die Verwendung von Klassenreferenzen vermieden werden, und auch in diesem Fall lassen sich Klassenreferenzen nutzbringend einsetzen. Vereinfacht gesehen können wir das obige Beispiel wie folgt ändern:

```
ClassRef := LiesKlassenreferenz(Stream);
NewObject := ClassRef.Create;
NewObject.ReadData;
```

Das Problem hierbei ist die Funktion *LiesKlassenreferenz*, denn sie existiert nicht. Die folgenden Routinen der Unit *Classes* helfen dabei, eine solche Funktion selbst zu implementieren:

```
procedure RegisterClass(AClass: TPersistentClass);
function FindClass(const ClassName: string): TPersistentClass;
```

Mit *RegisterClass* nehmen Sie die im Parameter angegebene Klassenreferenz in einer CLX-internen Liste auf, so dass Sie diese mit *FindClass* später über den Namen der Klasse abfragen können. Wenn Sie dann den Namen einer Klasse im Stream speichern, etwa wie in der folgenden Prozedur für das Objekt *obj* ...

```
procedure WriteClassRef(Writer: TWriter; obj: TObject);
begin
  Writer.WriteString(obj.ClassName);
end;
```

... können Sie mit Hilfe der *FindClass*-Funktion eine Klassenreferenz des gespeicherten Objekts wiederherstellen:

```
function ReadClassRef(R: TReader): TClass;
var
  ClassName: String;
begin
  ClassName := R.ReadString;
  Result := FindClass(ClassName);
end;
```

Die dritte Version des obigen Beispiels lautet dann:

```
ClassRef := ReadClassRef(ClassName);
NewObject := ClassRef.Create;
NewObject.ReadData;
```

Der Einfachheit halber verwendet dieses Beispiel die in Kapitel 4.3.4 behandelten *TFiler*-Klassen, welche die oben verwendeten Methoden *ReadString* und *WriteString* zum Schreiben und Lesen von Strings zur Verfügung stellen.

Vollständiges Beispiel zur polymorphen Objektspeicherung *R115*

Da es an dieser Stelle nicht darum gehen soll, die Daten einer Klasse zu speichern (ein Thema, das in Kapitel 6.3.3 erörtert wird), verwendet dieses Beispiel zwei »leere« Klassen, für die aber zur Veranschaulichung schon Methoden zum Speichern und Laden der Daten deklariert werden. Wie Sie der Deklaration von *RegisterClass* und *FindClass* aus der CLX entnehmen können, akzeptieren diese Funktionen nur Klassen, die von *TPersistent* abgeleitet sind, daher müssen auch die beiden Klassen des Beispiels von *TPersistent* abstammen. Für die Polymorphie des gesamten Vorgangs ist es wichtig, dass die Methoden zum Speichern und Laden der Daten aus der Datei ebenfalls polymorph sind, also in einer gemeinsamen Basisklasse, hier *TBaseClass*, als virtuell deklariert sind:

```
type
  TBaseClass = class(TPersistent)
    procedure WriteToFile(W: TWriter); virtual; abstract;
    procedure ReadFromFile(R: TReader); virtual; abstract;
  end;
```

```
  TBaseClassRef = class of TBaseClass;
  Class1= class(TBaseClass)
    procedure WriteToFile(W: TWriter); override;
    procedure ReadFromFile(R: TReader); override;
  end;
  Class2= class(TBaseClass)
    procedure WriteToFile(W: TWriter); override;
    procedure ReadFromFile(R: TReader); override;
  end;
```

Die folgende Prozedur wählt per Zufallszahl eine dieser beiden Klassen aus, erzeugt eine Objektinstanz davon und schreibt den Klassennamen in eine Datei (diesen Code finden Sie auf der CD eingebunden in das Projekt *PolyStrm*):

```
procedure PolymorpheObjekteSpeichern;
var
  S: TStream;
  Writer: TWriter;
  object1: TBaseClass;
begin
  S := TFileStream.Create(FileName, fmCreate);
  Writer := TWriter.Create(S, 2000);
  Randomize;
  if Random(10) < 5 then object1 := Class1.Create
    else object1 := Class2.Create;
  object2 := Class2.Create;
  WriteClassRef(Writer, object1);
  object1.WriteToFile(Writer);

  Writer.Free;
  S.Free;
end;
```

Die folgende Methode liest die von *SpeichernBeispiel* erzeugte Datei und rekonstruiert dabei das oben zufällig ausgewählte Objekt, wobei sie wie schon beschrieben die CLX-Methode *FindClass* verwendet. Vorher muss sie jedoch zuerst alle in Frage kommenden Klassen mit *RegisterClass* registrieren:

```
procedure TForm1.LesenPolymorpherDaten;
var
  S: TStream;
  Reader: TReader;
  ClassRef: TBaseClassRef;
  object1: TBaseClass;
begin
  RegisterClass(Class1);
  RegisterClass(Class2);

  S := TFileStream.Create(FileName, fmOpenRead);
  Reader := TReader.Create(S, 2000);
  ClassRef := TBaseClassRef(ReadClassRef(Reader));
```

```
    object1 := ClassRef.Create;
    object1.ReadFromFile(Reader);

    Reader.Free;
    S.Free;
  end;
```

Sie können auf diese Weise beliebige von *TPersistent* abgeleitete Klassen registrieren und brauchen dann nur noch den Namen der Klasse im Stream zu speichern, um die Objekte später wie im obigen Beispiel mit Hilfe der Klassenreferenz wiederherstellen zu können.

Weitere Funktionen der Unit *Classes* erlauben Ihnen, die Registrierung zurückzunehmen, Alias-Namen zu definieren und ganze Arrays von Klassenreferenzen auf einmal an- und abzumelden. Informationen dazu finden Sie in der Online-Hilfe zur *Classes*-Unit[1] oder im CLX-Quelltext. Die CLX selbst verwendet *RegisterClass*, *FindClass* und die verwandten Routinen dazu, die Komponenten eines Formulars ähnlich wie in den obigen Beispielen zu speichern.

Klassenregistrierung unabhängig von TPersistent

Die Beschränkung der CLX-Methoden *FindClass* und *RegisterClass* auf *TPersistent*-Ableitungen ist nicht immer akzeptabel, zumal sie sich sehr leicht umgehen lässt, indem man einfach eigene Versionen von *FindClass* und *RegisterClass* schreibt, die beliebige Klassen erlauben. Der TreeDesigner aus Kapitel 6 etwa speichert polymorphe Grafikelemente, deren Klasse *TGraphicElement* von *TObject* abgeleitet ist. Es wäre sicher nicht besonders lohnend, wenn nun jedes einzelne *TGraphicElement* zu einem *TPersistent*-Objekt gemacht werden würde, nur um die CLX-Methoden *RegisterClass* und *FindClass* verwenden zu können.

Stattdessen verwendet der TreeDesigner also eine neue Version dieser beiden Methoden, die Sie für Ihre eigene Verwendung in der Unit *StreamUtil* finden. Ansonsten folgt die polymorphe Speicherung von Grafikobjekten im TreeDesigner dem oben beschriebenen Muster.

```
var
  ClassList: TList;

function FindClass(AClassName: String): TClass;
var
  i: Integer;
begin
  for i := 0 to ClassList.Count - 1 do begin
```

1. In der dem Autor vorliegenden ursprünglichen Kylix-Version ist die Übersicht zur *Classes*-Unit nicht über den Index zu finden, aber vorhanden – mit *RegisterClass* als Suchwort stößt man in der Themenliste auf den Punkt *Classes routines*.

```
    Result := ClassList[i];
    if Result.ClassNameIs(AClassName) then Exit;
  end;
end;

procedure RegisterClass(C: TClass);
begin
  if ClassList.IndexOf(C) = -1 then
    ClassList.Add(C);
end;
```

4.3.4 TReader und TWriter

TReader- und *TWriter*-Objekte sind Hilfsobjekte, die die Handhabung eines Streams vereinfachen. Während *TReader* komfortable Methoden zur Verfügung stellt, mit denen Sie Daten aus einem Stream lesen können, widmet sich *TWriter* dem ebenso komfortablen Schreiben der Daten.

Die gemeinsame Basisklasse dieser beiden Klassen ist *TFiler* (sie beschreibt ein allgemeines »Ablageobjekt«, das mit einem Stream verbunden ist). Als wichtige Gemeinsamkeiten definiert sie die Methoden *DefineProperty* und *DefineBinaryProperty*, die jedoch erst bei der Programmierung von Komponenten interessant werden.

Initialisierung

Um ein *TReader*- oder *TWriter*-Objekt zu erstellen, rufen Sie den Konstruktor der Klasse wie folgt auf:

```
R := TReader.Create(Stream, 2048);
W := TWriter.Create(Stream, 2048);
```

Im ersten Parameter geben Sie ein bereits initialisiertes Stream-Objekt an, aus dem *TReader* seine Daten bezieht bzw. in das *TWriter* seine Daten schreibt. Der zweite Parameter ist in beiden Fällen die Größe, die das Objekt seinem internen Datenpuffer zuweisen soll.

Arbeitsteilung zwischen Streams und Ablageobjekten

Sinn der Unterscheidung zwischen den *TStream*- und den *TFiler*-Klassen ist eine größere Unabhängigkeit zwischen der Speicherung eines Datenstroms (Streams) und dem Aufbau der Daten. Während *TStream* nur unstrukturierte Datenblöcke kennt und sich darauf konzentriert, die Daten an einem bestimmten Ort, beispielsweise in einer Datei, unterzubringen, kennen die *TFiler*-Klassen verschiedene Datentypen wie Fließkommazahlen, Integerzahlen und Strings, verlassen sich beim Datentransport aber ganz auf *TStream*. Auch die scheinbar leistungsfähigen *ReadComponent*- und *WriteComponent*-Methoden von *TStream* arbeiten intern mit *TFiler*-Objekten.

Aufgrund dieser Unabhängigkeit können Sie von *TFiler* oder *TReader* theoretisch eigene Klassen ableiten, die weitere Datentypen speichern könnten und mit denen Sie zum Beispiel sowohl *TMemoryStreams* als auch *TFileStreams* benutzen könnten (wäre die *TReader*/*TFiler*-Funktionalität in der Klasse *TStream* integriert, müssten Sie dazu *zwei* neue Klassen ableiten – jeweils eine von *TMemoryStream* und eine von *TFileStream*).

Trotz dieser flexiblen allgemeinen Struktur sind auch die *TFiler*-Klassen sehr stark auf das Speichern von Formularen und Komponenten ausgerichtet, so dass hier nur die einfachen Möglichkeiten dieser Klassen zusammengefasst werden.

Verwendung

Die einfachste Verwendung der *TFiler*-Klassen besteht in der Speicherung der einfachen Datentypen: Hierfür definiert *TReader* die Methoden *ReadBoolean*, *ReadChar*, *ReadInteger*, *ReadFloat*, *ReadCurrency*, *ReadString*, *ReadWideString*, *ReadVariant*, *ReadInt64* und weitere Methoden für selten verwendete Typen, wobei Sie mit *ReadInteger* alle Integertypen bis einschließlich *LongInt* lesen können und von *ReadFloat* Fließkommazahlen des Typs *Extended* erhalten. *TWriter* enthält die entsprechenden Schreibmethoden. Der Aufruf ist in allen Fällen denkbar einfach:

```
Writer.WriteInteger(IntegerVar);
IntegerVar := Reader.ReadInteger;
```

Listen

Als etwas komplexere Aktion unterstützen die beiden Klassen auch das Schreiben und Lesen von Listen. Eine solche Liste besteht aus einer Startmarkierung, einer Folge von Datenblöcken, deren Struktur Sie definieren, und einer Endemarkierung.

- Das Schreiben einer Liste beginnt mit dem Aufruf der Methode *WriteListBegin*, die die Startmarke setzt. Danach speichern Sie die einzelnen Listenelemente und rufen schließlich die Methode *WriteListEnd* auf.
- Das Lesen der Liste beginnt mit der Methode *ReadListBegin*. Danach können Sie in einer Schleife alle Listenelemente auslesen, wobei Sie von der Methode *TReader.EndOfList* erfahren, wann Sie das Listenende erreicht haben. Schließlich beenden Sie das Lesen durch den Aufruf von *ReadListEnd*.

Wie der gesamte Ablauf konkret aussieht, demonstriert ein vollständiges Beispiel zur Verwendung von *TReader* und *TWriter* ab Seite 481 in den Abschnitten *Speichern der Tastenkürzel* und *Lesen der Tastenkürzel*.

Interne Arbeitsweise

Die oben beschriebenen Methoden sind zwar sehr komfortabel, verursachen aber einigen Mehraufwand: Anstatt lediglich die reinen Daten zu speichern, also zwei Bytes für eine *ShortInt*-Zahl, ein Byte für ein Zeichen usw., stellen sie jedem Datenbestandteil eine Typenkennung voran. So überprüft beispielsweise *WriteInteger*, ob der übergebene Zahlenwert in den Bereich eines Bytes passt, und schreibt in diesem Fall nur dieses eine Byte – plus dem Byte für die Typenkennung. Eine möglichst effiziente Speicherung größerer Datenmengen ist also nur mit den *TStream*-Klassen möglich (ein Beispiel dazu gibt Kapitel 6.3.3).

Diese Umstände haben jedoch den Vorteil, dass die mit *TReader* und *TWriter* geschriebenen Dateien problemlos zwischen verschiedenen Plattformen ausgetauscht werden önnen, ob das nun zwischen einer 16-Bit- und 32-Bit-Versionen einer Anwendung ist oder zwischen den heutigen 32-Bit-Anwendungen und zukünftigen 64-Bit-Nachfolgern. Wenn Sie beispielsweise früher unter 16-Bit-Windows mit Delphi 1 einen (16-Bit-) *Integer* mit *WriteInteger* geschrieben haben und diesen heute in einer 32-Bit-Version der Anwendung mit *ReadInteger* in einen (32-Bit-)Integer lesen, stellt *TReader* fest, dass der Wert in der Datei nur zwei Bytes beansprucht (wenn er im Wertebereich eines Bytes liegt, nur ein Byte). Sie liest dann nur die in die Datei geschriebenen Bytes und schreibt sie in den 32-Bit-Integer-Wert.

Zum Vergleich damit würde *TStream* heute, wenn Sie etwa mit

```
var
  Int: Integer;
begin
  Stream.Read(Int, sizeof(Int));
```

versuchen würden, einen 32-Bit-Integer zu erhalten, den 16-Bit-Wert aus der alten Datei nicht korrekt lesen. Es würden vier statt zwei Bytes gelesen und alle nachfolgenden Leseoperationen würden von der falschen Position starten.

4.3.5 Memory-Streams

Nicht immer sollen die Daten eines Programms in Dateien des Dateisystems gespeichert werden – auch für eine Übertragung zu einer Anwendung auf demselben PC oder über das Internet zu einem anderen Rechner wäre es denkbar, dass das Programm seine Daten zuerst in einem »temporären Paket« speichert und sie dann über- ägt. Das temporäre Paket kann z. B. ein Stream sein, der nicht mit einer Datei auf dem festen Datenträger verbunden ist, sondern sich nur im Hauptspeicher befindet. Ein besonders häufig vorkommendes Beispiel ist wohl, dass Daten über die Ausschneiden/Einfügen-Operationen von einer Anwendung in eine andere eingefügt werden oder innerhalb derselben Anwendung vervielfältigt werden sollen. Dabei kommt es

häufig vor, dass genau dieselben Daten, die mit *SaveToStream* oder ähnlichen Methoden in eine Datei geschrieben werden können, z. B. in die Zwischenablage kopiert werden sollen.

Die Polymorphie stellt einen Mechanismus dar, mit dem vorhandene *SaveToStream*-Methoden für die Datenablage in der Zwischenablage wiederverwendet werden können. *TStream* ist ja eine abstrakte Klasse und jede Methode, die in ein *TStream*-Objekt schreibt, weiß nicht, was sich tatsächlich hinter diesem Stream verbirgt (es sei denn, sie ist besonders neugierig und verwendet Abfragen wie *ClassName* oder den *is*-Operator).

TMemoryStream ist eine von *TStream* abgeleitete Klasse, die die von *TStream* definierten Lese- und Schreibmethoden für einen Bereich des Hauptspeichers implementiert. Das Beschreiben eines solchen Streams ist denkbar einfach, denn *TMemoryStream* passt den Speicherbereich von selbst so an, dass alle Daten hineinpassen. Sie können also munter drauflosschreiben wie in eine Datei.

Am Schluss stellt sich allerdings die Frage, was mit dem Memory-Stream geschehen soll. Wird das Stream-Objekt nämlich freigegeben, gehen die Daten verloren. Zum Kopieren der Daten in die Zwischenablage benötigen Sie daher direkten Zugriff auf den zugrunde liegenden Speicherbereich des Memory-Streams. Dieser ist über die Properties *TMemoryStream.Memory* (Zeiger auf den Bereich) und *Size* (Größe des Bereichs in Byte) möglich.

Die folgenden Zeilen zeigen ganz allgemein die grundsätzlichen Schritte zur Verwendung eines Memory-Streams:

```
MemoryStream := TMemoryStream.Create;
try
  // Schreiben der Daten:
  MemoryStream.Write(...);
  XYZ.SaveToStream(MemoryStream);
  // Endgültiges Ablegen der Daten (Registry, Zwischenablage,...):
  Lege_Daten_ab(MemoryStream.Memory, MemoryStream.Size);
finally
  MemoryStream.Free;
end;
```

Das Lesen eines Memory-Streams ist nur ein wenig komplizierter, denn nun kommen die Daten z. B. von außen aus der Zwischenablage und müssen erst wieder in den Memory-Stream zurückbefördert werden. Dazu stellen Sie den Memory-Stream *vorher* mit *SetSize* auf die erforderliche Größe und kopieren danach die Daten in seinen Speicherbereich, z. B. mit *Move*. Erst dann können die Daten wie aus jedem anderen Stream ausgelesen werden:

```
MemoryStream := TMemoryStream.Create;
try
  MemoryStream.SetSize(GetDataSize(Datenblock));
```

```
    Move(Datenblock, MemoryStream.Memory^, MemoryStream.Size);
    MemoryStream.Read(...);
    XYZ.ReadFromStream(MemoryStream);
  finally
    MemoryStream.Free;
  end;
```

Eine sehr nützliche Anwendung von Memory-Streams ergibt sich beim Kopieren eigener Datenformate in die Zwischenablage, wie es in Kapitel 4.7.3 gezeigt wird.

4.4 Grafikausgabe

Dieses Kapitel beschreibt die Komponenten der CLX, mit denen Sie den größten Teil der Grafikausgabe durchführen können. In der Praxis können Sie zwar so gut wie berall in der Anwendung Methoden zur Grafikausgabe aufrufen, in den meisten Fällen ist dies jedoch nur in Zusammenhang mit dem *OnPaint*-Ereignis sinnvoll. Wie Sie dieses Ereignis indirekt erzeugen, wenn sich die Grafik geändert hat, sowie Weiteres zu der Steuerung der Grafikausgabe lesen Sie in Kapitel 6.5; in diesem Kapitel geht es lediglich um die Grafikausgabe als solche.

4.4.1 Die Klasse TCanvas

Zentrales Objekt bei der Grafikausgabe mit der CLX ist immer eine Zeichenfläche der Klasse *TCanvas*. Sie zeichnet sich durch eine sehr einfache Handhabung aus. Während man auf einer niedrigeren Ebene diverse an der Grafikausgabe beteiligten Objekte erst initialisieren und nach dem Zeichnen dann Aufräumarbeiten vornehmen muss, entällt dies alles mit *TCanvas*. Wenn Sie das richtige Ereignis abwarten – wie z. B. das schon erwähnte *OnPaint* –, können Sie in der Ereignismethode sofort mit dem eigentlichen Zeichnen anfangen, das *Canvas*-Objekt initialisiert sich dann automatisch und soweit erforderlich.

Grafik-Properties und -Methoden

Betrachten wir zunächst ein kleines Beispiel zum Zeichnen einer drei Pixel breiten horizontalen Linie 200 Pixel unterhalb der Titelzeile eines Fensters:

```
with Canvas do begin
  Pen.Width := 3;
  Pen.Color := clRed;
  MoveTo(0, 200);
  LineTo(ClientWidth, 200);
end;
```

In diesem Beispiel zeigen sich zwei Grundprinzipien der Verwendung von *TCanvas*:

- Die Attribute der Grafik (Breite und Farbe der Linien, Farbe zum Füllen etc.) werden eingestellt, indem Sie Properties verändern. Viele Attribute sind den Kategorien Stift und Pinsel zugeordnet, die Properties der Klasse *TCanvas* sind. Um beispielsweise die Stiftfarbe zu verändern, müssen Sie ein darin geschachteltes Property verändern: *TCanvas.Pen.Color*.
- Die eigentliche Grafikausgabe findet in *TCanvas* mit einfachen Methoden statt, z. B. *MoveTo/LineTo*, *Rectangle*, *Ellipse*.

Wir werden in Kapitel 4.4.2 zunächst auf die Attribute der Grafik und damit auf die Zeichenwerkzeuge zurückkommen und dann in Kapitel 4.4.3 auf die Grafikausgabemethoden.

TCanvas-Instanzen

Um diese Properties und Methoden nutzen zu können, benötigen Sie zuerst einmal eine Instanz der Klasse *TCanvas*, die für die Zeichenfläche Ihres Fensters zuständig ist. Sie brauchen eine solche weder selbst zu erzeugen, noch müssen Sie lange nach einer bestehenden Instanz suchen, denn jedes Formular besitzt bereits ein *Canvas*-Property. Sie können also den Code aus dem obigen Beispiel direkt in eine mit dem *OnPaint*-Event verknüpfte Methode oder eine andere Methode Ihres Formulars einsetzen.

Oft soll jedoch nicht die gesamte Fläche des Formulars für die Grafikausgabe verwendet werden, weil auch noch Platz für einige andere Komponenten benötigt wird. Manchmal nimmt die Grafik auch nur einen besonders kleinen Teil des Formulars ein, z. B. in Dialogboxen, die in einer kleinen Vorschaugrafik anzeigen, wie sich die gerade eingestellten Optionen auswirken. In diesen Fällen ist es sinnvoller, auf das *Canvas*-Objekt des Formulars zu verzichten und die Grafikausgabe auf eine eigene Komponente zu beschränken.

Folgende Komponenten verfügen bereits über ein *Canvas*-Property:

- *TPaintBox* sollten Sie verwenden, um die Grafik auf einen Teil des Formulars zu beschränken. Ihr einziger Zweck ist es, eine Zeichenfläche zur Verfügung zu stellen; ihr besonderer Vorteil ist, dass sie nicht nur kleiner, sondern auch größer als das Formular sein kann, was sich zum Scrollen ausnutzen lässt (siehe Kapitel 6.5.4).
- *TPrinter* stellt das Papier, Folien, T-Shirts oder was immer bedruckt wird, als *TCanvas*-Objekt dar, das Sie grundsätzlich so behandeln können wie jedes andere *Canvas*-Objekt auch. Beachtenswerte Details und ein Beispiel finden Sie in Kapitel 6.6.3.
- *TBitmap*, die Klasse für Pixelgrafiken (Bitmaps), stellt eine Zeichenfläche zur Verfügung, die nicht unbedingt am Bildschirm sichtbar sein muss; mehr über *TBitmap* lesen Sie in Kapitel 4.5.2.

- *TDrawing* ist eine Art von Metadatei, gewissermaßen ein Recorder für Grafikoperationen. Alles, was Sie mit dem *Canvas*-Property dieser Klasse anstellen, lässt sich später durch einen einzigen Befehl auf anderen Zeichenflächen reproduzieren (Kapitel 4.5.1).
- In manchen Komponenten, die sich normalerweise selbst zeichnen, können Sie die Kontrolle über die Grafikausgabe übernehmen, so z. B. in *TListBox*, *TIconView*, *TComboBox* und *TStringGrid*. Auch bei diesen zeichnen Sie mit einem *Canvas*-Property, müssen dabei aber die Grafikausgabe selbst auf die zu zeichnende Zeile bzw. Zelle des Steuerelements beschränken. Ein Beispiel für eine selbst gezeichnete Listbox finden Sie in den Beispielprogrammen *OwnerDraw1* und *OwnerDraw2*, die in Kapitel 4.5.4 besprochen werden; eine selbst gezeichnete *TDrawGrid*-Komponente im Programm *DrawGrid* (siehe Grafikbeispiel in Kapitel 4.4.3).
- Wenn Sie in selbst geschriebenen Komponenten eine Grafik zeichnen möchten, leiten Sie die Komponente entweder von *TGraphicControl* oder von *TCustomControl* ab, je nachdem, ob die Komponente mit einem Qt-Widget verbunden sein soll. Beide Klassen definieren gleichermaßen ein geschütztes *Canvas*-Property, das nur in bestimmten Klassen wie *TImage* als *public* deklariert wird.
- Der Vollständigkeit halber soll auch das Formular als Besitzer eines *Canvas*-Objekts nicht in dieser Aufzählung fehlen.

Wenn Sie nicht das *Canvas* des Formulars, sondern das einer Komponente verwenden, haben Sie den Vorteil, dass die Grafikausgabe am Rand der Komponente abgeschnitten wird (was als *Clipping* bezeichnet wird) und dass das Koordinatensystem relativ zur Komponente und nicht relativ zum Formular ist (die Koordinate (0/0) befindet sich also in der linken oberen Ecke der Komponente bzw. von deren Arbeitsbereich). Durch diese Unabhängigkeit vom Formular ist es auch leicht möglich, die gesamte Grafikausgabe an eine andere Stelle des Formulars zu verschieben, ohne die Methodenaufrufe zur Grafikausgabe zu verändern.

ür die Kenner der Gerätekontexte und Handles, wie sie auf Systemebene vorkommen, sei noch einmal betont, dass dieses *Canvas*-Property wie alle anderen Properties funktioniert. Es steht zur Verfügung, solange sein Besitzer, in diesem Fall also das Formular, existiert. Das bedeutet, dass Sie jederzeit Attribute verändern oder zeichnen önnen. Die gesetzten Attribute, wie z. B. die Breite des Stifts, bleiben dauerhaft gültig, obwohl die zugehörigen Objekte der darunter liegenden Schnittstellen von Qt und X Window zwischendurch freigegeben werden können.

4.4.2 Zeichenwerkzeuge

Das Beispiel des letzten Abschnitts hat gezeigt, wie leicht es ist, grafische Objekte auszugeben. Die Methoden, die diese Objekte zeichnen, erwarten normalerweise nur die

Koordinaten der Objekte. Die anderen Angaben wie Farbe, Füllmuster und Zeichenmodus werden als Eigenschaften der Zeichenfläche behandelt. Das hat den Vorteil, dass Sie nicht jedes Mal alle Attribute als Methodenparameter angegeben müssen, wenn Sie bei vielen aufeinander folgenden Grafikbefehlen immer dieselben Attribute verwenden.

Damit bei der großen Zahl der Zeichenattribute die Übersichtlichkeit nicht verloren geht, sind einige davon zu den Zeichenwerkzeugen Stift und Pinsel gruppiert worden; die Gruppierung der Schriftattribute zu Schriftarten ist wohl selbstverständlich. Auch die Schriftarten sollen hier als »Zeichenwerkzeuge« behandelt werden.

Die CLX besitzt natürlich auch für jeden Zeichenwerkzeugtyp eine eigene Klasse und für diese drei ähnlichen Klassen eine Basisklasse: *TGraphicsObject*. Diese weist jedoch als einziges neues und öffentliches Element nur das Ereignis *OnChange* auf, das bei jeder Veränderung eines der Properties, die in den abgeleiteten Klassen hinzugefügt werden, eintritt. Weitere Neuigkeiten gibt es erst bei den drei speziellen Zeichenwerkzeug-Klassen.

Stifte

Ein Stift (Klasse *TPen*) ist ein Zeichenwerkzeug mit den vier Eigenschaften *Color*, *Width*, *Style* und *Mode*: Dem *Color*-Property können Sie einen der 16,7 Millionen Farbwerte oder eine der Systemfarben (z.B. *clActiveCaption*) zuweisen und in *Width* geben Sie an, wie viele logische Einheiten (per Voreinstellung sind dies die Pixel) die vom Stift gezogene Linie breit sein soll. Eine Breite von 0 ist möglich, meint aber in jedem Fall die Breite eines Pixels, unabhängig davon, wie breit eine logische Einheit ist.

Um wirklich die Breite 0, also eine unsichtbare Linie zu erhalten, müssen Sie den Stil des Stifts auf *psClear* setzen. Die anderen möglichen Stile sind *psSolid*, *psDash*, *psDot*, *psDashDot* und *psDashDotDot* und sorgen für verschiedene Arten von gestrichelten und gepunkteten Linien. Sie können diese Stile einfach ausprobieren, indem Sie eine *TShape*-Komponente (Seite *Zusätzlich*) auf ein Formular setzen und das Property *Pen.Style* auf den gewünschten Stil einstellen.

Die Zeichenfunktionen verwenden den aktuellen Stift (*TCanvas.Pen*) für die äußeren Linien von Flächen wie Rechtecken, Ellipsen und Polygonen und natürlich für einzelne Linien selbst. Stiftfarbe, -breite, -stil und -modus sind somit einerseits vier von vielen Attributen der Zeichenfläche, andererseits sind sie als eigenständiges *TPen*-Objekt manipulierbar.

Hinweis: Anders als mit der VCL von Delphi für Windows können Sie in der CLX – dank der Fähigkeiten von Qt – gepunktete und gestrichelte Linien auch in einer Breite größer als einem Pixel ziehen.

TPen.Mode

Anders als reale Stifte können die Stifte von grafischen Benutzeroberflächen die Zeichenfläche perfekt abdecken, ohne sich mit der vorher da gewesenen Farbe zu vermischen oder diese durchschimmern zu lassen. Dies ist jedoch nur die Voreinstellung: In *Mode* können Sie neben dem Standard *pmCopy* 15 verschiedene andere Zeichenmodi einstellen, die jeweils eine andere Kombination von Stiftfarbe und Farbe der Zeichenäche bewirken.

Diese Modi können allerlei Effekte verursachen oder die additive Farbmischung von Wasserfarben simulieren, in den meisten Fällen wird jedoch ein ganz bestimmter Modus benötigt: Wenn Sie zwei gleiche Linien im *pmNotXor*-Modus übereinander zeichnen, heben sich beide gegenseitig auf, so dass sich dieser Modus besonders gut dafür eignet, vorläufige Linien bei Mausziehoperationen zu zeichnen (siehe Kapitel 6.4).

In den anderen Modi können Sie Stift und Zeichenfläche auch mit AND und OR verknüpfen und optional das Ergebnis mit NOT umkehren. Mit den Modi *pmBlack* und *pmWhite* bringen Sie Stifte aller Farben dazu, schwarz bzw. weiß auf jedem Untergrund zu zeichnen. Das *DrawGrid*-Beispiel in Kapitel 4.4.4 verwendet den Modus *pmNot*, um ein Rechteck zu invertieren.

Wichtig ist, dass *Pen.Mode* eigentlich nicht dem Stift als Attribut zuzurechnen ist, denn es wirkt sich auf *alle* Zeichenvorgänge aus, insbesondere also auch auf das, was eigentlich mit dem »Pinsel« gezeichnet wird. Sie können dies bereits mit der schon erwähnten *TShape*-Komponente ausprobieren. Zeichnen Sie zwei sich teilweise überlappende *Shapes* in ein Formular ein und verändern Sie *Color* und *Mode* der oberen Komponente, um die Effekte zu beobachten.

arben

ür die Farben können Sie dieselben Werte und vorgegebenen Farbkonstanten wie im Objektinspektor in den *Color*-Properties der Steuerelemente und des Formulars (z. B. *clBlack*, *clWhite*, *clBackground*, *clButton*) verwenden. Ausgewählte Farbtöne aus den 16,7 Millionen möglichen Werten können Sie aus drei Einzelwerten für Rot, Grün und Blau, jeweils zwischen 0 und 255 liegend, selbst zusammensetzen:

```
function RGB(r, g, b: Byte): TColor;
begin
  Result := (r or (g shl 8) or (b shl 16));
end;
```

Diese Funktion reiht per Bit-Shift-Operator *shl* die jeweils 8 Bit der Einzelwerte zu einem 24-Bit-Wert zusammen und kann etwa wie folgt verwendet werden, um die Stiftfarbe zu setzen:

```
Pen.Color := RGB(255, 120, 80); { maximales Rot, 120 grün, 80 blau }
```

TColor selbst ist übrigens ein 32-Bit-Wert. Sie sind als Steuerbyte zu betrachten, das bei einem Wert von ungleich Null besagt, dass der Farbwert anders zu interpretieren ist. Nur, wenn das obere Byte den Wert Null hat, können Sie die unteren drei Bytes wieder als Rot-, Grün- und Blauwerten interpretieren.

```
{ Rot, Gruen, Blau: Byte; }
Rot := byte(color);
Gruen := byte(color shr 8);
Blau := byte(color shr 16);
```

Pinsel

Mit Pinseln füllt Qt das Innere der gezeichneten Formen. Die Attribute eines Pinsels sind Farbe, Stil und Bitmap, für die jeweils ein Property zuständig ist:

- In der Farbe *Color* geben Sie wie bei *TPen.Color* einen *TColor*-Farbwert an.
- In *Style* wählen Sie eines der vordefinierten Muster aus: *bsSolid* für vollständige Füllung mit der Farbe, *bsClear* zum Abschalten der Füllung, *bsBDiagonal*, *bsFDiagonal*, *bsHorizontal*, *bsVertical*, *bsCross* und *bsDiagCross* für schräg ansteigende, schräg absteigende, horizontale und vertikale Linien sowie für zwei verschiedene Arten der Schraffur (auch diese Stile können Sie mit *TShape*-Komponenten ausprobieren, da *TShape* auch über ein Property *Brush.Style* verfügt). In der CLX gibt es außerdem noch die Stile *bsDense1* bis *bsDense7*, die aus verschieden dichten Punktmustern bestehen.
- Falls diese vorgegebenen Muster nicht reichen, können Sie im Property *Bitmap* ein *TBitmap*-Objekt angeben. Dieses wird dann anstelle der Farbe zum Füllen der Fläche verwendet. Zu *TBitmap* siehe Kapitel 4.5.2.

Schriftarten

Anders als Stift und Pinsel handelt es sich bei den Schriftarten nicht nur um eine Sammlung von Attributen der Zeichenfläche, sondern um Objekte, die in eigenen Dateien definiert werden (z.B. in TTF-Dateien für TrueType-Schriften), wobei die Dateien für die Programmierung keine Rolle spielen.

Die Klasse *TFont* beinhaltet eine vereinfachte Beschreibung einer Schriftart in Form der folgenden Properties:

Property	Typ	Bedeutung
Color	TColor	Farbe der Schrift
Height	Integer	Schrifthöhe in Pixeln
Name	TFontName	Name der Schriftart

Property	Typ	Bedeutung
Pitch	Integer	legt mit den Werten *fpDefault*, *fpVariable*, *fpFixed* fest, ob die Abstände der Zeichen so groß sein sollen, wie von der gewünschten Schriftart vordefiniert, oder ob Qt variablen ('I' ist schmaler als 'M') oder festen ('I' ist genauso breit wie 'M') Zeichenabstand einhalten und dafür ggf. die Schriftart wechseln soll.
PixelsPerInch	Integer	wird von der CLX intern verwendet, um *Height* in *Size* umzurechnen und umgekehrt.
Size	Integer	Höhe der Schrift in Punkten
Style	TFontStyle	eine Menge, die die Werte *fsBold*, *fsItalic*, *fsUnderline* und *fsStrikeOut* enthalten kann (fett, kursiv, unterstrichen und durchgestrichen)

Die meisten Eigenschaften der Schriftart sind stets auch in den Dialogen zur Schriftauswahl anzutreffen, einen von ihnen stellt der Objektinspektor als Property-Editor für die *Font*-Properties der verschiedenen Komponenten zur Verfügung.

Mit mehreren Zeichenwerkzeugen arbeiten

Um die Stiftattribute Ihres *Canvas*-Objekts zu setzen, haben Sie zwei Möglichkeiten: Normalerweise gehen Sie so vor wie im obigen Beispiel und setzen die Properties von *Canvas.Pen*, *Canvas.Brush* und *Canvas.Font* einzeln. Da die Objekte *Pen*, *Brush* und *Font* jedoch selbst Properties sind, können Sie sie auch als Ganzes neu setzen.

Sie können z.B. dynamisch neue Stift-Objekte erstellen und diese dem Property *Canvas.Pen* zuweisen. Das folgende Beispiel erzeugt ein dynamisches *TPen*-Objekt:

```
NewPen := TPen.Create;
... { Einstellen der Properties }
Canvas.Pen := NewPen;
...
Pen.Free;
```

In diesem Fall sind Sie lediglich dafür verantwortlich, das mit *Create* erstellte Objekt wieder mit *Free* (oder *Destroy*) zu löschen. Sie können es direkt löschen, nachdem Sie es dem Property *Canvas.Pen* zugewiesen haben, ohne *Canvas.Pen* dadurch in irgendeiner Form zu beeinträchtigen.

Ein praktisches Beispiel für die dynamische Erzeugung von Schrift-, Pinsel- und Stiftobjekten gibt der TreeDesigner – jedes seiner Grafikobjekte enthält ein Exemplar jedes dieser Objekttypen (siehe Kapitel 6.3.3).

4.4.3 Grafikmethoden

Die folgende Tabelle fasst die einfachen Methoden zur Grafikausgabe zusammen:

TCanvas-Methode	Parameter	Beschreibung
MoveTo	x, y	bewegt den Stift an eine neue Position, ohne zu zeichnen.
LineTo	x, y	zeichnet von der aktuellen Position bis zu (x, y) und setzt die aktuelle Stiftposition auf diesen Wert (der Punkt [x, y] selbst wird nicht gezeichnet).
Rectangle	x1, y1, x2, y2	zeichnet Quadrate und Rechtecke.
RoundRect	x1, y1, x2, y2, x3, y3	rundet die Ecken der Rechtecke mit Ellipsenvierteln ab. Die Größe der Ellipse, die geviertelt an die Rechteckränder verteilt werden soll, geben Sie in den Parametern *x3* und *y3* (Breite und Höhe) an.
Ellipse	x1, y1, x2, y2	zeichnet Kreise und Ellipsen.
Arc Cord Pie	x1, y1, x2, y2, x3, y3, x4, y4	zeichnen Ellipsenbögen, »abgeschnittene« Ellipsen und »Tortenstücke«.
PolyBezier, PolyBezierTo	array of TPoint	zeichnen eine Folge von Bézier-Kurven. Der Array-Parameter besteht aus einer Folge von Punkten, wobei sich hier jeweils ein Stützpunkt und zwei Kontrollpunkte abwechseln.
Polygon, Polyline	array of TPoint	zeichnen Linienfolgen und gefüllte Polygone.
TextOut	x, y, Text	Textausgabe.
TextRect	TRect, x, y,	Textausgabe innerhalb eines Rechtecks.

Linien zeichnen

Um eine einzelne Linie zu zeichnen, benötigen Sie zwei Methodenaufrufe: Zuerst setzen Sie den Stift mit *MoveTo* auf den Beginn der Linie und zeichnen die Linie dann mit *LineTo*. Genau genommen zeichnet *LineTo* jedoch den Endpunkt nicht mit, damit dieser, wenn er bei der nächsten Linie wieder als Startpunkt verwendet wird, nicht doppelt gezeichnet wird (was in bestimmten *Pen.Mode*-Einstellungen durchaus sichtbar werden kann).

Vorteilhaft wirkt sich die Trennung des Linienziehens in *LineTo* und *MoveTo* erst aus, wenn Sie mehrere verbundene Linien zeichnen. Hierfür bietet sich auch die gleich besprochene Methode *PolyLine* an.

ormen zeichnen

Zum Zeichnen von Rechtecken und Ellipsen genügt der Aufruf einer einzigen der in der obigen Tabelle gezeigten Methoden. Als Parameter geben Sie jeweils ein Rechteck an, in dem sich die Form befinden soll. Ähnlich wie bei den Linien werden auch hier nicht alle Koordinaten in die Zeichnung eingeschlossen: die Koordinaten *x2/y2* gehören nicht mehr zum gezeichneten Rechteck.

Die Ellipsenfunktion ist zwar durch die Rechteck-Koordinaten sehr einfach zu bedienen, hat aber auch einen Nachteil: Sie kann keine »schrägen« Ellipsen zeichnen, also solche, die entstünden, würde man die einfachen Ellipsen um weniger als 90 Grad drehen. Um solche Ellipsen zu erhalten, müssten Sie direkt auf die Qt-Bibliothek zugreifen und dort mit Koordinatensystem-Transformationen arbeiten (ein Beispiel hierfür, wenn auch ohne Ellipsen, zeigt Kapitel 5.4.3).

Alle Funktionen zum Zeichnen von Formen bedienen sich des gerade aktiven Stifts und füllen die Fläche außerdem mit dem gewählten Pinsel.

Arc, Chord und Pie

Ellipsen spielen auch bei drei weiteren Methoden eine Rolle. Beispielhafte Werke der Methoden *Arc*, *Chord* und *Pie* sehen Sie in Abbildung 4.1 (das Programm heißt CanvasMagnetism). *Arc* ist eine reine Linienfunktion und zeichnet einen Abschnitt der Ellipsenumrandung. *Chord* arbeitet so, als teilte sie die Fläche einer Ellipse mit einer Geraden in zwei Teile, von denen nur einer gezeichnet wird. *Pie* schneidet ein mehr oder weniger großes Tortenstück aus der Ellipse und zeichnet einen der beiden daraus entstehenden Flächenteile.

Die ersten vier Parameter aller drei Funktionen geben das umgebende Rechteck der grundlegenden Ellipse an. Dazu gesellen sich vier weitere Parameter, die zwei Punkte angeben, aus denen sich der Winkel errechnen lässt, der gezeichnet werden soll. Dies geschieht folgendermaßen: Von jedem der beiden Punkte berechnen die Funktionen eine Gerade, die die Ellipse teilt und dabei durch den Mittelpunkt geht. Zwei dieser Geraden teilen die Ellipse in vier Teile. Die Funktionen bewegen sich dabei gegen den Uhrzeigersinn von der ersten zur zweiten Geraden und zeichnen diesen Abschnitt. (Im Programm *CanvasMagnetism* aus Abbildung 4.1 können Sie den Punkt (x3/y3) mit der linken, den Punkt (x4/y4) mit der rechten Maustaste festlegen.)

Auch diese drei Funktionen bedienen sich des gerade aktiven Stifts, *Chord* und *Pie* füllen die Fläche außerdem mit dem gewählten Pinsel.

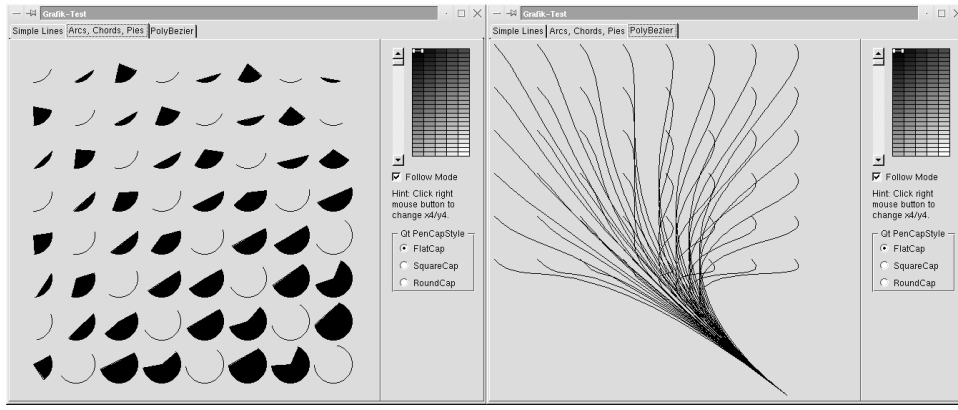


Abbildung 4.1: Beispielprogramm CanvasMagnetism. Linkes Fenster, unterste Reihe von links nach rechts: Pie, Arc und Chord. Rechts: Bézier-Kurven. Zum Verständnis des Programmnamens müssen Sie die Maus über die verschiedenen Seiten des Fensters bewegen.

Komplexere Objekte

Die Methoden *Polyline* und *Polygon* erlauben es schließlich, auch Formen mit komplizierterer Umrandung zu zeichnen, solange diese aus geraden Linien besteht. Diese Linien werden als Array von Punkten (*TPoint*) angegeben, so als würden die Linien mit einem *MoveTo* und vielen aufeinander folgenden *LineTo*-Befehlen gezeichnet werden. Genau das tut *Polyline* auch, während *Polygon* das entstandene Konstrukt auch noch ausfüllt und dabei den Endpunkt der letzten Linie mit dem Anfangspunkt der ersten Linie verbindet.

Der folgende Ausschnitt zeichnet mit *PolyLine* ein nicht ausgefülltes Dreieck, wobei das Array direkt angegeben ist:

```
Canvas.Polyline([Point(90, 10), Point(210, 10), Point(150, 120)]);
```

Polygon wird im Folgenden dazu verwendet, ein Sechseck zu zeichnen, dessen Berechnung z.B. so aussehen könnte wie in der Methode *PaintHexagon* des TreeDesigners (siehe Kapitel 6.5.2):

```
var
  Points: array[0..5] of TPoint;
begin
  { Berechnung von Points }
  ...
  Canvas.Polygon(Points);
```

Die Methoden *PolyBezier* und *PolyBezierTo* werden ähnlich aufgerufen. *PolyBezier* benötigt zumindest vier Punktangaben: den Startpunkt, zwei Kontrollpunkte sowie einen Endpunkt. Für jeden weiteren Stützpunkt verlängern Sie diese Liste um jeweils zwei

Kontrollpunkte, gefolgt von dem jeweiligen Stützpunkt. Das Beispielprogramm *CanvasMagnetism* zeichnet eine Reihen von einfachen Linien mit nur zwei Stützpunkten (siehe Abbildung 4.1).

extausgabe

Wenn Sie die Methode *TextOut* einfach so verwenden, setzt sie die linke obere Ecke des extes an die in den Parametern übergebene Position. Einen wichtigen Unterschied zur VCL von Windows betrifft den Hintergrund des Textes: Während Sie in der VCL den Hintergrund automatisch löschen bzw. mit dem aktuellen Pinsel füllen lassen können (wenn *Brush.Style* den Wert *bsSolid* hat), zeichnet Qt den Text immer transparent, der bisherige Hintergrund bleibt also unter dem Text sichtbar.

In vielen Fällen ist die Textausgabe nicht ganz so einfach: So soll z. B. oft die Ausrichtung des Textes berücksichtigt werden, und sofern Sie keine Schrift mit festem Zeichenabstand verwenden, ist es oft interessant, den Platzbedarf einer Zeichenkette zu erfahren.

ür letztere beiden Aufgaben besitzt *TCanvas* weitere Methoden: Mit *TextWidth* erhalten Sie die Breite, die der String mit der aktuellen Schriftart (*TCanvas.Font*) benötigen würde, mit *TextHeight* die Höhe. Beide Funktionen geben Abmessungen in virtuellen Koordinaten zurück, per Voreinstellung also in Pixeln.

Die folgende Zeile berechnet mit *TextHeight* die Y-Position *TopPos*, an die ein String geschrieben werden muss, damit seine vertikale Mitte in der Höhe *VMiddle* liegt:

```
TopPos := VMiddle-CanvasRef.TextHeight('A') div 2;
```

Dazu zieht die Anweisung die halbe Texthöhe von der gewünschten Mitte ab. Dieses Beispiel ist der Textausgabe aus Kapitel 6.5.2 entnommen, einem Kapitel, das auch zeigt, wie Sie andere Ausrichtungen des Textes erreichen.

Die zweite Textausgabemethode von *TCanvas* ist *TextRect*. Sie (bzw. die von ihr aufgeufene Qt-Methode) führt automatisches Clipping des Textes an einem im ersten Parameter angegebenen Rechteck durch und verhält sich ansonsten wie *TextOut* (Beispiel siehe Kapitel 6.5.5).

Weitere TCanvas-Methoden

Zu den restlichen Methoden gehören weitere Variationen zum Zeichnen eines Rechtecks, eine Methode zum Füllen einer Fläche und Methoden, die Grafiken oder Bitmaps zeichnen. Die folgende Übersicht verzichtet zugunsten der Online-Hilfe auf Parameterbeschreibungen:

Methode	Beschreibung
DrawFocusRect	zeichnet einen Rechteck-Umriss im XOR-Modus (siehe Pen.Mode, *pmNotXor*).
FillRect	zeichnet die Füllung eines Rechtecks – ohne Umriss.
CopyRect	kopiert ein Rechteck aus einem anderen *TCanvas*-Objekt auf die Zeichenfläche.
Draw	zeichnet ein *TGraphic*-Objekt an eine bestimmte Position (Beispiel in Kapitel 4.5.4).
StretchDraw	füllt ein Rechteck mit einem *TGraphic*-Objekt aus (Beispiel in Kapitel 4.5.4).

Grafikausgabe mit Qt

Die für die Grafikausgabe zuständige Qt-Klasse *QPainter* besitzt viele Methoden mehr als *TCanvas*. Wenn Sie diese verwenden wollen, müssen Sie direkt auf das von *TCanvas* verwendete *QPainter*-Objekt zugreifen (über *TCanvas.Handle*). Auf diese Weise können Sie zum Beispiel

- unterschiedliche Abbildungsmodi (virtuelle bzw. logische Koordinatensysteme) einstellen (siehe Kapitel 6.6.1),
- eigenes Clipping durchführen (siehe Kapitel 6.5.5) und
- mehr Kontrolle über die Textausgabe ausüben (z.B. mit *ExtTextOut* oder mit *SetTextAlign*, für ein Beispiel zu letzterer Funktion siehe Kapitel 6.5.2).

Alle Methoden von *QPainter* und von weiteren mit der Grafikausgabe zusammenhängenden Qt-Klassen sind in der separat erhältlichen Qt-Dokumentation beschrieben.

Zeichenwerkzeug-Eigenschaften von Qt

Als winziges Beispiel dafür, dass die CLX nicht alle Grafikfähigkeiten der Qt-Bibliothek ausreizt, verwendet das Beispielprojekt *CanvasMagnetism* auch eine dieser speziellen Qt-Fähigkeiten, und zwar können Sie in der Radioschalter-Gruppe *Qt PenCapStyle* wählen, in welchem Stil die Endstücke einzelner Linien gezeichnet werden sollen. Dies betrifft nur Linien, die mehr als einen Pixel breit sind, denn solche sind ja im Prinzip keine Linien, sondern Rechtecke – es sei denn, der *PenCapStyle* wurde auf den Wert *PenCapStyle_RoundCap* gestellt, denn dann werden die Linienenden abgerundet. Mit *FlatCap* erhalten Sie dagegen einen rechtwinkligen Abschluss und mit *SquareCap* ebenso, jedoch werden die Enden dann noch etwas verlängert, um sich evtl. besser mit anderen Linienabschnitten zu überlappen.

In *CanvasMagnetism* finden Sie die folgende Methode zur Bearbeitung des Radiogruppen-*OnClick*-Ereignis (es dient hier eigentlich nur als Zusammenfassung der verschiedenen *PenCapStyles*):

```
//  CapStyle: Qt.PenCapStyle;
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
```

```
    case (Sender as TRadioGroup).ItemIndex of
      0: CapStyle := PenCapStyle_FlatCap;
      1: CapStyle := PenCapStyle_SquareCap;
      2: CapStyle := PenCapStyle_RoundCap;
    end;
    Paintbox1.Invalidate;
  end;
```

Die entscheidende Anweisung findet sich in der *OnPaint*-Methode des Beispielformulars. Vor der Ausgabe jeglicher Grafik wird die folgende Routine aufgerufen, um den Linienstil gemäß der Wahl des Benutzers zu setzen:

```
QPen_setCapStyle(Paintbox1.Canvas.Pen.Handle, CapStyle);
```

4.4.4 Besitzergezeichnete Komponenten

Da dieses Buch mit dem TreeDesigner aus Kapitel 6 über eine umfangreiche Beispielanwendung verfügt, die Grafiken im Arbeitsbereich von Fenstern ausgibt, sollen sich die Beispiele für die Grafikausgabe in diesem Kapitel eines anderen Gebiets annehmen: dem der besitzergezeichneten Steuerelemente. Unter »Besitzer« kann man zweierlei verstehen: das Formular, das die Methoden zum Zeichnen eines Steuerelements enthält, oder eben die Person, die das Formular programmiert.

Grundsätzliche Vorgehensweise

iele Komponenten der CLX bieten neben ihrem Standardaussehen und einigen Properties, mit denen Sie dieses beeinflussen können, die Möglichkeit, die gesamte Komponente oder Teile davon selbst zu zeichnen. Auch dieser Prozess beruht wieder auf der Ereignisorientierung: Die Komponenten senden Ihnen eine Nachricht, wenn das Neuzeichnen erforderlich ist, und Sie stellen eine Methode bereit, die auf diese Nachricht reagiert. Während sich Zeichenflächen (*TPaintBox*, siehe Kapitel 6.5.2) bei einem *OnPaint*-Ereignis neu zeichnen, enthalten die Namen der Ereignisse von benutzergezeichneten Komponenten normalerweise den Begriff *OnDraw*, z. B. *OnDrawItem* oder *OnDrawCell*. Typisch für diese Ereignisse sind drei Arten von Parametern:

- ein Parameter, der besagt, welches von mehreren Elementen gezeichnet werden soll, z. B. welcher Eintrag einer Listbox oder eines TreeViews,
- die Koordinaten des Rechtecks, in welchem das Element gezeichnet werden soll.
- ein Statusparameter, der den Zustand des Elements angibt, z. B. ob das Element ausgewählt, fokussiert, deaktiviert ist usw.

Die Methode, die das *OnDraw...*-Ereignis bearbeitet, kann die Grafik nun mit Hilfe dieser Parameter und des *Canvas*-Objekts, das von der Komponente zur Verfügung gestellt wird, ausgeben.

Des Weiteren unterscheidet man bei besitzergezeichneten Komponenten zwei wichtige Kategorien: Komponenten, deren dargestellte Elemente eine feste Größe haben, und Komponenten, bei denen diese Größe variiert. Bei *TListBox* können Sie z. B. mit den Stilen *lbOwnerDrawFixed* und *lbOwnerDrawVariable* zwischen beidem wählen und bei *TDrawGrid* ist die Größe immer fest vorgegeben.

Falls die Größe der Elemente variabel ist, gilt es normalerweise, neben *OnDraw...* noch ein zweites Ereignis zu bearbeiten, dessen Name mit *OnMeasure* beginnt. Dieses stellt Ihnen einen Variablenparameter zur Verfügung, in dem Sie angeben, wie viel Platz ein bestimmtes Element in Anspruch nimmt.

Verwenden von TDrawGrid *R94*

Das Beispielprogramm *DrawGrid* (Abbildung 4.2) verwendet die 64x64 Zellen eines *DrawGrids* dafür, einen Farbübergang zu zeichnen, wobei jede Zelle mit Zeilen- und Spaltennummern beschriftet wird. Wir benötigen hierfür nur eine einzige Methode, die jede einzelne Zelle ausgibt. Sie ist mit dem *TDrawGrid*-Ereignis *OnDrawCell* verknüpft und hat die folgenden Parameter:

- *Col* und *Row* spezifizieren die zu zeichnende Zelle.
- *Rect* gibt an, in welchem Bereich der Zeichenfläche (*TDrawGrid.Canvas*) gezeichnet werden soll.
- *State* enthält Flags, die über den Status der Zelle Aufschluss geben: *gdSelected* (markiert; dieses Flag ist nur dann bei mehreren Zellen gleichzeitig möglich, wenn die Option *goRangeSelect* in *Options* gewählt ist), *gdFixed* (am Rand befestigte Zelle) und *gdFocused* (Zelle hat den Tastaturfokus; dies ist immer nur bei einer Zelle der Fall, *TDrawGrid* zeichnet bereits automatisch eine Umrandung für diese Zelle).

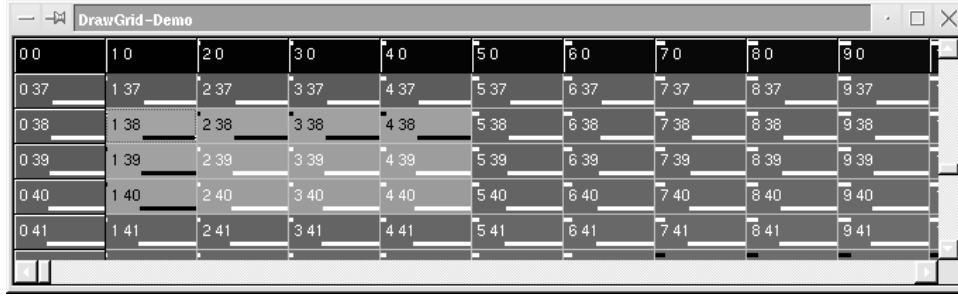


Abbildung 4.2: DrawGrid-Komponenten lassen ihre Zellinhalte vom Besitzerformular zeichnen.

Das Beispielprogramm berücksichtigt von *State* nur das Flag *gdSelected*, um alle gewählten Zellen zu invertieren. Zum Invertieren wird über das bereits vorher mit *FillRect*

gezeichnete Rechteck ein weiteres Rechteck, diesmal im Zeichenmodus *pmNot* gezeichnet. Dieser Modus besagt, dass beim Zeichnen nur der aktuelle Bildschirminhalt und nicht die aktuelle Farbe von Stift oder Pinsel berücksichtigt werden. Und der Bildschirminhalt wird in der Weise berücksichtigt, dass er invertiert wird. Neben dieser kleinen Besonderheit besteht die Zeichenmethode nur aus Standard-Grafikoperationen:

```
procedure TForm1.DrawGrid1DrawCell(Sender: TObject;
   Col, Row: LongInt; Rect: TRect; State: TGridDrawState);
begin
  with DrawGrid1.Canvas do begin
    { Zeichnen eines Hintergrunds: }
    Brush.Style := bsSolid;
    Brush.Color := RGB(Col*4, Row*4, 0);
    FillRect(Rect);
    { Zeichnen des Balkens über den Zahlen }
    Brush.Color := clWhite;
    Rect2 := ...(langweilige Berechnung, siehe CD-ROM)
    FillRect(Rect2);
    { Zeichnen des Balkens unter den Zahlen }
    Rect2 := ...(langweilige Berechnung, siehe CD-ROM)
    FillRect(Rect2);
    { Beschriftung der Zelle }
    Brush.Style := bsClear; { transparente Textausgabe }
    with Rect do
      TextOut(Left+4, Top+4, Format('%d %d', [Col, Row]));
    { Zusätzliche Hervorhebungen der selektierten Zellen
      durch Invertierung: }
    if gdSelected in State then begin
      Brush.Style := bsSolid;
      Pen.Mode := pmNot;
      Rectangle(Rect); // wirkt nun wie InvertRect unter Windows
      Pen.Mode := pmCopy; // wieder normalen Modus einstellen
    end;
  end;
end;
```

Hinweis: ***TDrawGrid*** besitzt im Gegensatz zu *TStringGrid* kein *Cells*-Property. Es verfügt jedoch wie *TStringGrid* über eine Möglichkeit, den Inhalt der Zellen zu editieren (Flag *goEditing* im Property *Options*). Wenn Sie den vom Benutzer eingegebenen Text speichern wollen, müssen Sie dies beim Ereignis *OnSetEditText* tun. Wenn der Benutzer erneut versucht, den Text zu verändern, erstellt *TDrawGrid* wieder ein Editierfeld und fragt Sie über das Ereignis *OnGetEditText*, wie der bisherige Text lautet.

TStringGrid erbt von *TDrawGrid* die Möglichkeit, vom Besitzer gezeichnet zu werden. Wenn Sie also ein Gitter-Element selbst zeichnen und gleichzeitig Text in jeder Zelle speichern wollen, ist die Verwendung von *TStringGrid* die einfachste Lösung.

4.5 Grafiken als Objekte

Neben den im letzten Kapitel besprochenen grafischen Objekten, die Sie mit *TCanvas*-Methoden erzeugen, können Sie auch komplette Grafiken am Stück zeichnen. Die CLX stellt für drei Typen dieser Grafiken Klassen bereit. Zwei davon sind auf Qt zurückzuführen und eine wird durch die CLX selbst implementiert:

- *TBitmap*-Objekte sind Bitmapgrafiken beliebiger Größe, die Sie neu erzeugen oder aus einer Datei und aus der Zwischenablage laden können.
- *TIcon* steht für ein einzelnes Icon, einer speziellen Bitmap, die nur bestimmte Größen annehmen darf. Diese ist die von der CLX selbst implementierte Klasse. Intern verwendet sie zwar auch die Qt-Bitmap-Klasse, aber der Bitmap-Funktionalität fügt *TIcon* noch die Logik zum Lesen von Icon-Ressourcen hinzu (z. B. solchen Icon-Ressourcen, die noch von Windows-Anwendungen stammen).
- *TDrawing* schließlich kapselt eine aufgezeichnete Folge von Qt-Grafikbefehlen. Eine *TDrawing*-Grafik setzt sich also nicht aus Pixeln, sondern aus den Objekten zusammen, die Sie z. B. mit den *TCanvas*-Methoden zeichnen können. *TDrawing* ist vergleichbar mit der VCL-Klasse *TMetafile*, wenn die von beiden erzeugten Dateiformate nicht miteinander kompatibel sind.

4.5.1 Die drei TGraphic-Klassen

TGraphic ist die gemeinsame Basisklasse der drei in der Einleitung genannten Typen und definiert abstrakte virtuelle Methoden, mit denen Sie eine Grafik aus einer Datei oder aus der Zwischenablage laden bzw. sie dort speichern können.

Da die drei abgeleiteten Klassen die folgenden Methoden jeweils auf ihre Weise implementieren (soweit notwendig), können Sie diese bei allen drei Grafiktypen gleichermaßen verwenden:

Methode	Beschreibung
Assign(Source: TPersistent)	überschreibt *TPersistent.Assign*, weist dem Grafik-Objekt den Inhalt eines *TPicture*-Objekts zu.
LoadFromFile(FileName: String)	liest die Grafik aus einer Datei (*TIcon* und *TDrawing* erfordern spezielle Formate, zu den von *TBitmap* unterstützten Formaten siehe Kapitel 1.4.9).
SaveToFile(FileName: String)	schreibt die Grafik in eine neue Datei.
LoadFromStream(TStream)	liest die Grafik von der aktuellen Stream-Position (wobei der Stream beispielsweise eine offene Datei sein kann).
SaveToStream(TStream)	schreibt die Grafik an die aktuelle Stream-Position.

Außerdem gibt es die folgenden Properties:

Property/Ereignis	Beschreibung
Empty	*True*, wenn noch keine Grafik mit den oben aufgezählten Methoden geladen oder zugewiesen wurde
Height	Höhe der Grafik in Pixeln
Modified	*True*, wenn die Grafik seit dem letzten Speichern geändert wurde
Width	Breite der Grafik in Pixeln
OnChange	Verknüpfung zu einer Methode, die bei jeder Änderung der Grafik aufgerufen wird, auch bei jeder Änderung des *Canvas* einer Bitmap

Während die Bitmaps und Metadateien die nachfolgenden Teile von Kapitel 4.5 ganz ür sich haben, wird *TIcon* im Folgenden kurz beschrieben.

Icons

Icons sind Bitmaps, die vorwiegend zur Symbolisierung von Anwendungen und Dokumenten verwendet werden (einige Klassen der CLX wie *TApplication* und *TForm* verfügen zu diesem Zweck über ein *Icon*-Property). Da Icons an strenge Größenvorgaben gebunden sind, werden für viele alltägliche Aufgaben innerhalb einer Anwendung Bitmaps statt Icons verwendet, so z. B. für die Bilder auf den SpeedButtons. Während Icon-Symbole entweder 32x32, 32x16 oder 64x64 Pixel groß sein müssen, önnen Sie Bitmaps beispielsweise auch in der Größe 20x20 erzeugen.

Ein Vorteil von Icons gegenüber Bitmaps ist, dass Sie mehrere in Farbe und Auflösung unterschiedliche Versionen eines Icons als ein Bild (eine Icon-Ressource, eine Icon-Datei) speichern können. So kann zur Laufzeit eine Version des Icons ausgewählt werden, die die Anforderungen des aktuellen Grafikmodus möglichst gut erfüllt. Leider verfügt Kylix nicht wie Delphi für Windows über einen Bildeditor, mit dem Sie bequem Dateien dieses Formats erstellen können. Normalerweise werden unter Linux und X Window aber auch Bitmaps statt Icons verwendet. Die Klasse *TIcon* in der CLX ist daher insbesondere für diejenigen ideal, die eine Anwendung von Windows nach Linux portieren.

Sie können *TIcon*-Objekte zur Laufzeit einfach über den parameterlosen Konstruktor *Create* erstellen, mit den von *TGraphic* geerbten Methoden laden und speichern sowie mit den *TCanvas*-Methoden *Draw* und *StretchDraw* am Bildschirm ausgeben wie eine Bitmap. Falls Sie das *Icon*-Property eines Formulars oder der Anwendung verändern wollen, brauchen Sie diesem jedoch kein neues Icon-Objekt zuzuweisen, sondern können es direkt verändern:

```
Application.Icon.LoadFromFile('BUSY.ICO');
```

4.5.2 Bitmaps

In den meisten Fällen, in denen einzelne Pixel manipuliert werden, werden diese nicht direkt auf dem Bildschirm verändert, sondern in einer Bitmap, die sich im Speicher befindet. Mit Methoden von *TCanvas* können Sie Bitmaps vom Speicher auf den Bildschirm kopieren.

Vorteile von Speicherbitmaps

Programme, die sehr intensiven Gebrauch von Bitmaps machen, sind Mal- und Bildbearbeitungsprogramme. Bei diesen werden durch die Verwendung von Bitmaps mehrere Probleme gelöst:

- Da andere Fenster das Fenster des Bildbearbeitungsprogramms auf dem Bildschirm verdecken können, muss das Bild noch an einer anderen Stelle als auf dem Bildschirm gespeichert werden.
- Bitmaps im Speicher sind nicht an die Abmessungen des Bildschirms gebunden, sondern können viel größer sein.
- Auch die Farben eines Bitmaps lassen sich im Speicher geräteunabhängig behandeln. So kann ein Bildbearbeitungsprogramm ein Echtfarben-Bild laden und intern in einer Echtfarben-Bitmap speichern, die dann in einem Bildschirmmodus mit nur 256 oder 64 K Farben dargestellt wird. Bei der Darstellung muss das System die 24-Bit-Farbinformationen dann in Farbindizes einer 256-Farben-Palette umrechnen bzw. auf die 16 Bit des 64-K-Farbmodus beschneiden.
- Schließlich können Bitmaps, wenn sie z.B. vom Speicher auf den Bildschirm kopiert werden, auch in der Auflösung angepasst werden, damit die Größen von Original und Abbildung übereinstimmen. Wenn Sie z.B. ein mit 300 dpi gescanntes Bild auf dem Bildschirm in gleicher Größe darstellen wollen, muss die Auflösung theoretisch auf die Auflösung des Monitors umgerechnet werden (z.B. 120 dpi). Bei der Größenveränderung einer Bitmap kommt es natürlich zu Informationsverlusten (wenn bei der Verkleinerung Pixel weggelassen werden) oder zu Klötzchenbildung (wenn bei der Vergrößerung Pixel einfach vervielfacht werden).

Geräteunabhängigkeit bei Bitmaps

Die genannten Punkte, besonders die letzten drei, lassen sich als Geräteunabhängigkeit zusammenfassen, wobei die Unabhängigkeit wegen der Verzerrungen und Informationsverluste bei der Größenänderung weit weniger groß ist als bei Metadateien.

TBitmap und TCanvas

Eine Bitmap weist sehr große Ähnlichkeit mit einem *TCanvas*-Objekt auf: Sie können auf beiden verschiedene grafische Objekte ausgeben, einzelne Pixel verändern und Sie önnen Ausschnitte zwischen beiden hin- und herkopieren. Diese Ähnlichkeiten gründen ausnahmsweise nicht darin, dass die eine Klasse von der anderen abgeleitet ist oder dass beide eine gemeinsame Basisklasse besitzen, sondern darin, dass *TBitmap* ein *TCanvas*-Objekt *enthält* (anzusprechen unter dem Property *Canvas*).

Da *TCanvas* bereits eine Zeichenfläche repräsentiert und mit dem *Pixels*-Array über den wesentlichen Bestandteil einer Bitmap verfügt, benötigt *TBitmap* nicht viel mehr als sein *Canvas*-Property und die von *TGraphic* geerbten (und überschriebenen) Methoden.

Zu diesen wenigen Erweiterungen gehören die Methode *Dormant* und die Properties *Monochrome* und *Format*: Im Property *Monochrome* erfahren Sie, ob die Bitmap einfarbig ist, und in *Handle* finden Sie eine Referenz auf das Qt-Objekt, welches die Bitmap verwaltet (ein *QPixmap*), und *Format* gibt eines der zahlreichen von Qt unterstützten Grafikformate an, in dem die Bitmap erzeugt wurde. Mit der Methode *Dormant* können Sie die durch die Bitmap belegten Qt-Ressourcen freigeben, ohne die Bitmap selbst zu verlieren (*TBitmap* speichert die Daten dann in einem selbst verwalteten Speicherbereich und erzeugt erst bei Bedarf wieder eine Qt-Pixmap).

Bitmaps erzeugen *R90*

Durch das *Canvas*-Property hat also die Klasse *TBitmap* ihrer Verwandten *TIcon* die Möglichkeit voraus, die einmal geladene Grafik auch zu verändern. Passend dazu können Sie mit *TBitmap* auch ganz neue Bitmaps erzeugen. Dazu genügt es, ein neues Bitmap-Objekt mit *Create* zu erzeugen und die Properties *Height* und *Width* zu verändern, anstatt die Bitmap mit einer der *TGraphic*-Methoden zu laden:

```
Bitmap := TBitmap.Create;
Bitmap.Height := 200;
Bitmap.Width := 200;
{ Ausgabe von Grafik: }
Bitmap.Canvas.Pixels[0, 0] := RGB(rot, gruen, blau);{ einen Punkt setzen }
Bitmap.Canvas.Polygon(...); { ein Polygon zeichnen }
```

4.5.3 TPicture

Die Klasse *TPicture* ist eine Klasse, die Metadateien, Bitmaps und Icons abstrakt als Bilder« ansieht und die durch ihren Namen leicht mit der Klasse *TGraphic* verwechselt werden kann. Während jedoch *TGraphic* als Elternklasse für einen der drei Grafiktypen steht, *enthält* ein Objekt der Klasse *TPicture* ein von *TGraphic* abstammendes Objekt. Sinn der Klasse *TPicture* ist es, zur Laufzeit zwischen den drei Grafiktypen zu wechseln, ohne dass Sie dafür verschiedene Objekte verwalten müssen.

TImage

Zu allem Überfluss gibt es auch noch die Komponente *TImage*, deren Name zwar so gut wie das Gleiche meint wie *TPicture*, die aber eine visuelle Komponente ist, welche ein *TPicture*-Objekt enthält. Sie erlaubt Ihnen, zur Entwurfszeit im Formular einen Platz für ein Bild zu reservieren oder auch schon ein Bild zu laden (über das *Picture*-Property im Objektinspektor). Zur Laufzeit müssen Sie normalerweise nur mit dem *Picture*-Property umgehen, so dass Sie alle im Folgenden beschriebenen Vorgehensweisen von *TPicture* auf *TImage* übertragen können.

TPicture-Methoden

Die einfachste Verwendung von *TPicture* besteht in der Verwendung der Methoden *LoadToFile* und *SaveToFile*, die den Typ der Datei anhand der Endung automatisch erkennen und ein entsprechendes Objekt erstellen:

```
Picture1.LoadFromFile('greeting.xpm');
Picture2.LoadFromFile('logo.pic');
```

TPicture verfügt auch über die Properties *Height* und *Width*, die die Größe der Grafik zurückgeben. Grafiken, die Sie wie im obigen Beispiel laden, können Sie mit den *TCanvas*-Methoden *Draw* und *StretchDraw* ausgeben, so dass Sie für diese einfache Verwendung gar nicht mehr über *TPicture* zu wissen brauchen. Unter dieser Oberfläche ist *TPicture* jedoch eine sehr flexible Grafikobjekt-Verwaltungszentrale.

Aufbau von TPicture

Zentraler Bestandteil eines *TPicture*-Objekts ist ein *TGraphic*-Objekt, das Sie über das Property *Graphic* ansprechen können. Wenn es sich um eine Bitmap oder ein Icon handelt, können Sie es auch über die Properties *Bitmap* und *Icon* ansprechen. Der große Vorteil dieser Properties gegenüber eigenständigen *TGraphic*-Objekten ist, dass Sie sich um die Konstruktion und die Freigabe dieser Objekte nicht zu kümmern brauchen. So können Sie nacheinander Grafiken verschiedenen Typs in das *Picture* laden:

```
Picture.Bitmap.LoadFromFile(...);
Canvas.Draw(0, 0, Picture);
Picture.Icon.LoadFromFile(...)
```

Jedes Mal, wenn Sie eines der Properties *Bitmap* und *Icon* ansprechen, prüft *TPicture*, welcher Objekttyp gerade aktiv ist. Ist dies ein anderer als der von Ihnen gewählte, gibt *TPicture* das alte Objekt frei und erzeugt ein neues. Im obigen Beispiel wird also die zuerst geladene Bitmap beim zweiten *LoadFromFile*-Aufruf durch ein *TIcon*-Objekt ersetzt.

Wenn Sie allerdings *TDrawing*-Objekte in *TPicture* verwenden wollen, müssen Sie unter Verzicht auf ein *Picture.Drawing*-Property die Datei mit *Picture.LoadFromFile* laden. *TPicture* erkennt die Datei jedoch nur, wenn sie die Endung `.ddw` aufweist.

Das einzige Objekt, das Sie noch selbst konstruieren und freigeben müssen, ist *TPicture* selbst (*Picture := TPicture.Create* und *Picture.Free*), allerdings können Sie sich auch das ersparen, wenn Sie beim Entwurf des Formulars eine *TImage*-Komponente einfügen, die ja wie alle anderen Komponenten des Formulars automatisch verwaltet wird. Auf deren Property *Picture* können Sie so zugreifen wie auf ein eigenständiges *TPicture*-Objekt.

Grafikobjekte kopieren

ber den bisher gezeigten Zugriff hinaus können Sie den Properties *Graphic*, *Bitmap* und *Icon* auch selbst erzeugte Objekte zuweisen:

```
  Bitmap := TBitmap.Create;
  Picture.Bitmap := Bitmap;
{ oder Picture.Graphic := Bitmap }
  Bitmap.Free;
```

TPicture kopiert den Inhalt der so zugewiesenen Objekte mit der in *TPersistent* eingeführten Methode *Assign* (siehe Kapitel 3.1.1) in sein eigenes Grafik-Objekt, verändert das zugewiesene Original-Objekt also nicht, so dass Sie dies wie im obigen Beispiel selbst wieder freigeben müssen.

Sie können auch jegliche Grafik aus dem *TPicture*-Objekt entfernen, indem Sie dem entsprechenden Property den Wert *nil* zuweisen. Da all dies auch für das *Picture*-Property der *TImage*-Komponente gilt, zum Abschluss ein Beispiel für diese Komponente:

```
{ Löschen der gerade angezeigten Grafik: }
  Image.Picture.Graphic := nil;
{ oder einfach: Image.Picture := nil }
```

Zusammenfassung der Begriffe

Wer nun einen Gedächtnistest machen will, möge sich zu erinnern versuchen, was die Unterschiede sind zwischen *Image*, *Picture*, *Drawing* und *Graphic*.

TImage	**visuelle Komponente, die ein *TPicture* enthält**
TPicture	Komfort-Klasse, die Grafiken verschiedenen Typs verwaltet
TGraphic	abstrakte Klasse, von der Grafiken verschiedenen Typs abgeleitet sind (*TBitmap*, *TDrawing* und *TIcon*)
TDrawing	spezielle von *TGraphic* abgeleitete Grafikklasse, die – und jetzt wird es mit der Benennung langsam gefährlich – ein *QPicture* kapselt

4.5.4 Bitmaps für eine OwnerDraw-Listbox

Das Beispielprogramm *OwnerDraw1* enthält eine besitzergezeichnete Listbox, in der alle Bitmap-Dateien eines Verzeichnisses angezeigt werden (siehe Abbildung 4.3). Zur Wahl dieses Verzeichnisses dient eine *DirectoryListBox*-Komponente aus der Komponentenbibliothek des Buchs. Die Dateien werden in einer normalen Listbox angezeigt, deren *Style*-Property auf *lbOwnerDrawVariable* eingestellt wurde und um die es hier geht.

Die erste Programmversion geht übrigens nicht besonders schonend mit dem Speicher um, da sie alle Bitmaps gleichzeitig und unkomprimiert in den Speicher lädt. Zur Ansicht von Verzeichnissen mit vielen großen Bitmaps sollten Sie daher unbedingt die zweite Programmversion *OwnerDraw2* verwenden, die am Schluss dieses Kapitels vorgestellt wird.

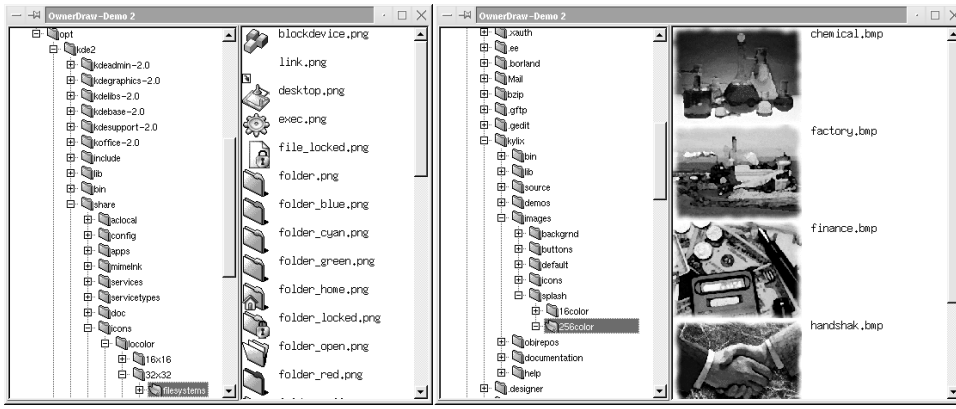


Abbildung 4.3: Eine OwnerDraw-ListBox im Beispielprogramm OwnerDraw2

Die Formularklasse enthält drei zusätzliche Variablen und zwei selbst erzeugte Methoden:

```
TForm1 = class(TForm)
  ...
public
  Dir: string;
  MaxHeight: Integer;
  Bitmaps: TList;
  procedure LoadFiles;
  procedure FreeListItems;
end;
```

Dir enthält das Verzeichnis, dessen Bitmap-Dateien geladen werden sollen. Es wird von der Laufzeitbibliothek automatisch als leerer String initialisiert, so dass die Dateien des aktuellen Verzeichnisses gezeigt werden.

MaxHeight gibt die maximale Höhe in Pixeln an, die eine Bitmap in der Listbox-Komponente für sich beanspruchen darf. Sie wird beim *OnCreate*-Ereignis mit dem Wert 90 initialisiert.

Bitmaps ist eine Liste der Klasse *TList*, die ebenfalls in der *OnCreate*-Methode initialisiert und mit einem Aufruf von *LoadFiles* mit *TBitmap*-Objekten gefüllt wird:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MaxHeight := 90;
  Bitmaps := TList.Create;
  LoadFiles;
end;
```

Bitmaps laden *R148*

LoadFiles löscht zunächst den Inhalt der Listbox und die Bitmapliste und durchsucht dann das gerade gesetzte Verzeichnis (*Dir*) nach Bitmap-Dateien (zu den Standardfunktionen *FindFirst* und *FindNext* siehe Kapitel 2.8.1):

```
procedure TForm1.LoadFiles;
var
  SearchRec : TSearchRec;
  Bitmap: TBitmap;
  PassCount: Integer;
const
  MaskList: array[0..5] of string =
    ('*.BMP', '*.bmp', '*.xpm', '*.png', '*.gif', '*.jpg');
begin
  with ListBox1 do begin
    Clear;
    FreeListItems;
    for PassCount := Low(MaskList) to High(MaskList) do begin
      if FindFirst(Dir+MaskList[PassCount],
                   faAnyFile, SearchRec)=0 then begin
        repeat
          Bitmap:=TBitmap.Create;
          Bitmap.LoadFromFile(Dir+SearchRec.Name);
          Bitmaps.Add(Bitmap);
          Items.Add(LowerCase(SearchRec.Name));
        until FindNext(SearchRec)<>0;
        SysUtils.FindClose(SearchRec);
      end;
    end;
  end;
end;
```

Für jede gefundene Datei erstellt die Methode mit *TBitmap.Create* ein neues Bitmap-Objekt und beauftragt dieses, die Datei zu lesen. Daraufhin fügt die Methode die Bitmap an die Bitmap-Liste an und erweitert die Listbox um den Namen der Datei.

Hinweis: Normalerweise wäre das Property *TListBox.Items.Objects* ein geeigneter Ort, zu jedem Dateinamen die zugehörige Bitmap zu speichern. Dazu müssten wir hier statt *Items.Add* die Methode *Items.AddObject* aufrufen (siehe Kapitel 4.1.1). In diesem Fall funktioniert das jedoch nicht, denn der interne Ablauf von *AddObject* führt dazu, dass das im Folgenden noch erklärte *OnMeasureItem*-Ereignis auftritt, bevor die Bitmap in der Liste gespeichert wird. Da wir die Bitmap jedoch schon in der Bearbeitung von *OnMeasureItem* benötigen, wird sie hier in einer separaten Liste gespeichert.

Neue Verzeichnisse wählen

Jedes Mal, wenn der Benutzer das Verzeichnis wechselt, muss die Liste der Bitmaps und der Inhalt der Listbox neu aufgebaut werden. Das *OnChange*-Event der *DirectoryListBox* muss also selbst bearbeitet werden:

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject;
  Node: TTreeNode);
begin
  Dir := IncludeTrailingPathDelimiter(DirectoryListBox1.Directory);
  ListBox1.Clear;
  LoadFiles;
end;
```

Bitmaps zeichnen

Nach dem erfolgreichen Aufbau der Bitmapliste können wir uns nun der Hauptaufgabe zuwenden, dem Zeichnen der Bitmaps. Hierbei spielt die Bitmap nicht mehr die aktive Rolle, d.h. es gibt keine Methode *DrawToCanvas*, die zu den Methoden *SaveToFile*, *CopyToClipboard* usw. passen würde. Stattdessen rufen Sie eine Methode des *Canvas* auf, in dem die Bitmap gezeichnet werden soll, und übergeben dieser die Bitmap als Parameter.

Zur Auswahl stehen die beiden *TCanvas*-Methoden *Draw* und *StretchDraw*, die in diesem Beispielprogramm beide zum Einsatz kommen: Falls die Bitmap nicht breiter ist als die Listbox und nicht höher als die in *MaxHeight* festgelegte maximale Höhe, wird sie mit *Draw* unverzerrt gezeichnet, andernfalls wird sie mit *StretchDraw* in das zur Verfügung stehende Rechteck gepresst unter Missachtung dessen, ob die Seitenverhältnisse noch stimmen. Der entsprechende Code-Ausschnitt sieht wie folgt aus (*Rect* ist das Rechteck, das der Eintrag in der Zeichenfläche der Listbox beansprucht):

```
if (Bitmap.Width > ListBox1.ClientWidth) or
   (Bitmap.Height > MaxHeight) then
  ListBox1.Canvas.StretchDraw(Rect, Bitmap)
else
  ListBox1.Canvas.Draw(Rect.Left, Rect.Top, Bitmap);
```

Bevor wir zur vollständigen Methode kommen, müssen wir uns etwas mit der besitzergezeichneten Listbox beschäftigen.

Besitzergezeichnete TListBox-Komponenten

TListBox ist eine der in Kapitel 4.4.4 erwähnten Komponenten, deren Elemente variable Größe haben können, wenn sie durch den Besitzer gezeichnet werden. Sie können im Property *Style* mit den Stilen *lbOwnerDrawFixed* und *lbOwnerDrawVariable* zwischen beidem wählen. Im letztgenannten Stil muss außer dem Zeichenereignis *OnDrawItem* noch ein zweites Ereignis namens *OnMeasureItem* bearbeitet werden. Dieses stellt Ihnen einen Variablenparameter zur Verfügung, in dem Sie angeben, wie viel Platz ein bestimmtes Element in Anspruch nimmt. Nachdem so jeder einzelne Eintrag abgemessen wird, erhält *OnDrawItem* jeweils die folgenden Parameter:

- den Index des zu zeichnenden Eintrags,
- die Koordinaten des Rechtecks, in dem der Eintrag gezeichnet werden soll,
- als Statusparameter eine Menge, die die Elemente *odSelected*, *odFocused* und *odDisabled* enthalten kann (für »Eintrag ist ausgewählt, fokussiert, deaktiviert«).

Zeichnen einer OwnerDraw-Listbox *R95*

Die Listbox des Beispielprogramms hat den Stil *lbOwnerDrawVariable* (angegeben im Property *Style*), zeichnet also jede Bitmap in ihrer jeweiligen Größe, wobei das selbst gesetzte Limit bei 90 Pixeln Höhe liegt.

Das Ereignis *OnDrawItem* liefert, wie oben beschrieben, die drei Parameter für das zu zeichnende Element (*Index*, verwendbar als Index für die Arrays *Bitmaps* und *TListbox.Items*), das Ausgaberechteck (*Rect*) und den Hervorhebungsstatus (*State*). Letzterer wird in diesem Beispielprogramm allerdings ignoriert. Wie oben schon beschrieben, gibt die Zeichen-Methode die Bitmap mit *Draw* und *StretchDraw* aus; vorher füllt sie den Hintergrund schwarz, und nachher zeichnet sie noch den Dateinamen auf die Bitmap:

```
procedure TForm1.ListBox1DrawItem(Sender: TObject; Index: Integer;
  Rect: TRect; State: TOwnerDrawState; var Handled: Boolean);
var
  Bitmap: TBitmap;
  Text: string;
  Stretched: Boolean;
begin
  { zu zeichnen sind: }
```

```
    Bitmap := Bitmaps[Index];  { die Bitmap ... }
    Text := ListBox1.Items[Index]; { ... und der Dateiname }
    { Schrift einstellen }
    if Assigned(Bitmap) then
      with ListBox1.Canvas do begin
        { Hintergrund schwarz füllen }
        Font.Color := clBlack;
        Font.Name := 'Fixed';
        Font.Size := 13;
        Brush.Style := bsSolid;
        Brush.Color := clWhite;
        FillRect(Rect);
        { Bitmap zeichnen }
        if (Bitmap.Width>ListBox1.ClientWidth) or
           (Bitmap.Height>MaxHeight) then begin
          StretchDraw(Rect, Bitmap);
          Stretched := True;
        end else begin
          Draw(Rect.Left, Rect.Top, Bitmap);
          Stretched := False;
        end;
        { Bitmap beschriften }
        if Stretched or (Bitmap.Width>ListBox1.ClientWidth div 2) then begin
          { große Bitmap -> Text darauf ausgeben }
          { Hintergrund des Textes einfarbig füllen }
          Brush.Style := bsSolid;
          Brush.Color := clBlack;
          FillRect(Types.Rect(Rect.Left+5,
                              Rect.Bottom-TextHeight('A'),
                              Rect.Left+15+TextWidth(Text),
                              Rect.Bottom));
          TextOut(Rect.Left+10, Rect.Bottom-TextHeight('A'), Text)
        end else begin
          { kleine Bitmap -> Text daneben ausgeben: }
          Brush.Style := bsClear; { transparenter Texthintergrund }
          TextOut(Rect.Left+Bitmap.Width+10, Rect.Top, Text);
        end;
        Font.Color := clBlack;
      end;
    Handled := True;
  end;
```

Zum Schluss setzt die Methode noch den *var*-Parameter *Handled* auf *True*, und gibt damit bekannt, dass das Zeichnen erledigt ist. Andernfalls würde danach auch *TListBox* noch ihre Ausgabefunktion durchführen.

Der Parameter *Rect* wird in vielen der obigen Anweisungen dazu verwendet, die Grafik an der richtigen Position auszugeben; lediglich beim Größenvergleich zieht die Methode auch die Variablen *MaxHeight* und *ListBox1.ClientWidth* zu Hilfe (nur der Einfachheit halber, um Berechnungen wie *Rect.Right-Rect.Left* zu vermeiden).

Hinweis: Da die Listbox des Beispielprogramms wie üblich keinen horizontalen Scrollbar enthält, braucht sie nicht wie das TreeView-Beispiel aus Kapitel 4.4.4 die Scroll-Position zu berücksichtigen. Bei Listboxen *mit* horizontaler Bildlaufleiste stellt sich jedoch das gleiche Problem wie in Kapitel 4.4.4.

Abmessungen selbst gezeichneter Elemente

Auch die Methode für das *OnMeasureItem*-Event holt sich zunächst die in *Index* angegebene Bitmap aus der *Bitmaps*-Liste, um deren Größe feststellen zu können. Sie schreibt dann die erforderliche Höhe in den Variablenparameter *Height*, die selbst gesetzte Höhenbegrenzung *MaxHeight* (90 Pixel) einbeziehend:

```
procedure TForm1.ListBox1MeasureItem(Control: TWidgetControl;
     Index: Integer; var Height: Integer);
var
  Bitmap: TBitmap;
begin
  Height := 16;
  if Index < Listbox1.Items.Count then begin
    Bitmap := TBitmap(Bitmaps[Index]);
    if Assigned(Bitmap) then
      Height := Bitmap.Height;
    if Height > MaxHeight then Height := MaxHeight;
  end;
end;
```

Die Freigabe der dynamischen Bitmap-Liste *R149*

Da wir die Bitmaps hier nicht in Komponenten der Klasse *TImage* eingebunden haben, müssen wir sie selbst freigeben. Dazu genügt es nicht, die Liste freizugeben, sondern es muss für jedes einzelne Bitmap die Methode *Free* aufgerufen werden. Dies erledigt die Formularmethode *FreeListItems*. Sie wird nicht nur beim Ende des Programms aufgerufen, sondern jedes Mal, wenn ein neues Verzeichnis ausgewählt wurde und die Liste neu aufgebaut werden muss (siehe *DirectoryListBox1Change*):

```
procedure TForm1.FreeListItems;
begin
  while Bitmaps.Count > 0 do begin
    TBitmap(Bitmaps[0]).Free;
    Bitmaps.Delete(0);
  end;
end;
```

Beim Ende des Programms wird zusätzlich zu dieser Methode auch die Liste *Bitmaps* eigegeben, die in *FormCreate* dynamisch erzeugt wurde:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeListItems;
  Bitmaps.Free;
end;
```

Optimierung der Bitmap-Liste R91

Ein großer Nachteil der oben gezeigten Bitmap-Lesemethode ist, wie schon gesagt, der enorme Speicherbedarf großer Bitmaps, denn obwohl auch große Bitmaps nur sehr klein dargestellt werden, werden sie komplett in den Speicher geladen. Es ist jedoch nicht schwer, diesen Nachteil zu beheben und sicherzustellen, dass für jede Bitmap nur so viel Speicher verbraucht wird, wie zur Darstellung in der Listbox notwendig ist.

In der verbesserten Programmversion *OwnerDraw2* ist die Methode *LoadFiles* so geändert, dass Bitmaps, die größer als 150x90 Pixel sind, auf diese Maximalgröße verkleinert werden, bevor sie endgültig in der Liste gespeichert werden.

Dazu benötigt die Methode *LoadFiles* eine neue lokale Variable ...

```
var
  SmallBitmap: TBitmap;
```

... und erhält eine neue *repeat*-Schleife, die die bisherige Schleife komplett ersetzt:

```
repeat
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile(Dir+SearchRec.Name);
  if (Bitmap.Height > 90) or
     (Bitmap.Width > 150) then
  begin
    SmallBitmap := TBitmap.Create;
    SmallBitmap.Height := 90;
    SmallBitmap.Width := 150;
    SmallBitmap.Canvas.StretchDraw(Rect(0, 0, 150, 90), Bitmap);
    Bitmaps.Add(SmallBitmap);
    Bitmap.Free; { die unverkleinerte Bitmap freigeben }
  end else
    Bitmaps.Add(Bitmap); { Bitmap ist schon klein genug }

  ListBox1.Items.Add(LowerCase(SearchRec.Name));
until FindNext(SearchRec) <> 0;
```

Falls die Maximalgröße nicht überschritten wird, läuft die Methode wie bisher ab. Andernfalls erstellt sie, wie in Kapitel 4.5.2 beschrieben, eine völlig neue zweite Bitmap mit dem Namen *SmallBitmap*. In diese zeichnet sie mit *StretchDraw* eine kleine Version der großen Bitmap. Anschließend wird *SmallBitmap* statt *Bitmap* zur Bitmap-Liste hinzugefügt und *Bitmap* freigegeben. Für alle anderen Methoden des Programms ändert sich nichts.

Hinweis: Das simple *StretchDraw* im obigen Code-Auszug führt dazu, dass Bitmaps mit extremen Seitenverhältnissen (z.B. 1000 Pixel breit, aber nur 16 Pixel hoch) bis zur Unkenntlichkeit verzerrt werden, da sie immer in die 150x90 Pixel große Fläche gepresst werden. Die Programmversion auf der CD-ROM wurde daher noch einmal verbessert – sie behält nun nach Möglichkeit bei der Verkleinerung noch das Größenverhältnis der Bitmap bei.

4.5.5 TDrawing und Metadateien

TDrawing ist eine Kapselung der Qt-Klasse *QPicture*. Da der Name *TPicture* aus historischen Gründen in Delphi anderweitig verwendet wird (und zwar für die in Kapitel 4.5.3 erläuterten Zwecke), musste Borland auf einen anderen Namen ausweichen.

Die Grafiken, die in *TDrawing* gespeichert wird, stellen die einfachste Form von geräteunabhängiger Grafik dar, denn sie bestehen intern einfach aus einer Folge von Grafikausgabebefehlen. Diese sind spezifisch für die Qt-Bibliothek, aber da Qt auf verschiedenen Plattformen verfügbar ist, ist so auch die in *TDrawing* gespeicherte Grafik plattformunabhängig, wenn Sie sie etwa in einer Datei speichern.

om Prinzip her sind *Drawings* bzw. *Pictures* (im Sinne von Qt) das Gleiche wie die Metadateien von Windows, die in der VCL von *TMetafile* gekapselt werden, jedoch haben die Qt-Metadateien nicht unwesentliche Vorteile: Sie haben keinerlei Einschränkungen, was die in ihnen speicherbaren Grafikbefehle angeht. Bei Windows-Metadateien sind dagegen diverse Verbote zu beachten für Grafikoperationen, die man zwar auf dem Bildschirm ausführen kann, nicht aber in einer Metadatei.

Dank der Basisklasse *TGraphic* lässt sich *TDrawing* genauso einfach aus einer Datei oder aus der Zwischenablage laden, am Bildschirm ausgeben und in eine Datei oder in die Zwischenablage kopieren wie *TIcon*. Das Ausgeben einer Metadatei am Bildschirm besteht darin, dass die Grafikbefehle, die in der Datei gespeichert sind, auf die *Canvas*-Zeichenfläche angewendet (»abgespielt«) werden.

Metadateien erzeugen und verwenden *R103*

Im Gegensatz zu Icons ist es bei Metadateien sehr leicht vorstellbar, dass Sie sie auch im Programm verändern bzw. von Grund auf erzeugen wollen. Das Projekt *Metafile* auf der CD (siehe Abbildung 4.4) enthält ein Formular, welches eine Metadatei ausgibt, die es beim *OnCreate*-Ereignis selbst erzeugt hat. Die Metadatei wird nicht auf Datenäger geschrieben, sondern verbleibt immer im Speicher, und zwar in der Variablen *Drawing*:

```
type
  TForm1 = class(TForm)
    ...
  public
```

```
    Drawing: TDrawing;
  end;

implementation

procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Drawing := TDrawing.Create;
  Drawing.Height := 300;
  Drawing.Width := 300;
  Drawing.Canvas.Start(True);
  for i:=0 to 150 do begin
    Drawing.Canvas.Rectangle(2*i, 2*i, 2*i+75, 2*i+75);
    Drawing.Canvas.Rectangle(300-2*i, 2*i, 300-2*i+75, 2*i+75);
  end;
  Drawing.Canvas.Stop;
// Drawing.SaveToFile('DrawingTest.ddw'); (Speichern nicht erforderlich)
end;
```

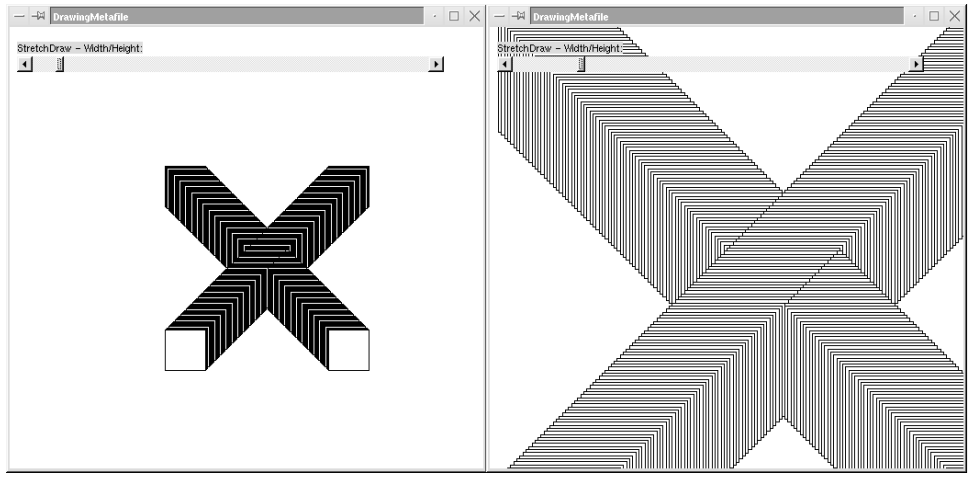


Abbildung 4.4: Ein TDrawing-Objekt spielt gespeicherte Grafikausgabebefehle wieder ab.

Nach der Erzeugung des *TDrawing*-Objekts wird zunächst die Größe der Zeichenfläche (600x600 Einheiten) eingestellt und dann die Grafik erstellt, indem ganz normale *TCanvas*-Methoden für die Zeichenfläche *Drawing.Canvas* aufgerufen werden. Wichtig ist hier, dass die Grafikausgabe durch Aufrufe von *TCanvas.Start* und *TCanvas.Stop* eingerahmt wird, denn ohne *Start* wird das Qt-Objekt, welches die Metadatei verwaltet, nicht richtig initialisiert.

Beim Zeichnen wird zunächst der gesamte Formularinhalt mit *FillRect* gelöscht und dann die in *Drawing* gespeicherte Grafik mit *StretchDraw* ausgegeben. Parameter von *StretchDraw* ist neben der Grafik ein Rechteck, in das die Ausgabe erfolgen soll. Auf diese Weise kann die Grafik nahezu beliebig skaliert werden:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.FillRect(ClientRect);
  Canvas.StretchDraw(Rect((ClientWidth-Scrollbar1.Position*2) div 2,
                          (ClientHeight-Scrollbar1.Position*2) div 2,
                          (ClientWidth+Scrollbar1.Position*2) div 2,
                          (ClientHeight+Scrollbar1.Position*2) div 2),
                     Drawing);
end;
```

Im Beispielprogramm ist die Skalierung von einem ScrollBar abhängig, bei dessen Bewegung die Grafik natürlich aktualisiert werden muss:

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Invalidate;
end;
```

Da das *Drawing*-Objekt in *OnCreate* selbst erzeugt wurde und keine Komponente darstellt, die von einer anderen Komponente automatisch gelöscht wird, ist das Beispielprogramm selbst dafür verantwortlich, das Objekt wieder freizugeben. Dies geschieht beim *OnDestroy*-Ereignis des Formulars:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Drawing.Free;
end;
```

4.6 Menüs

Obwohl sich Menüs ganz einfach visuell zusammensetzen und mit Methoden verbinden lassen, gibt es viele Gründe, sich auch zur Laufzeit im Programmcode etwas mit den Menüs zu beschäftigen, zum Beispiel:

- Das einfachste Ziel ist, Menüpunkte zeitweise zu deaktivieren, da sie gerade nicht anwendbar sind, oder die Menüpunkte mit einer Markierung zu versehen (wie in der Kylix-IDE z.B. die Menüpunkte unter ANSICHT | SYMBOLLEISTEN). Für diese einfachen Aufgaben müssen Sie lediglich die Properties *Checked* und *Enabled* der entsprechenden *TMenuItem-* oder *TAction*-Komponente zur Laufzeit beeinflussen. Beispiele dafür finden Sie in Kapitel 6.2.4.

- Weniger einfach ist es, die Menüstruktur zur Laufzeit zu ändern. Zwar können Sie durch Austausch des Properties *TForm.Menu* gleich das gesamte Menü auswechseln, oft sollen aber nur kleine Erweiterungen vorgenommen werden, beispielsweise mit einer Fensterliste oder einer Dateiliste. Je nach Anwendung sind auch andere Möglichkeiten denkbar, bei denen eine solche Liste innerhalb eines Menüs erheblich benutzerfreundlicher ist als eine eigene Dialogbox und außerdem platzsparender als eine (aufklappbare) Liste innerhalb des Fensters.
- Eine weitere Möglichkeit ist die Veränderung der Tastenkürzel zur Laufzeit, die normalerweise nur auf Wunsch des Anwenders stattfindet. Mit Hilfe der CLX ist dieses Problem wieder sehr schnell zu bewältigen.
- Etwas aus der Reihe fällt das besitzergesteuerte Zeichnen eines Menüs, bei dem Sie die Bildschirmdarstellung des Menüs durch Ihren eigenen Code bestimmen. Dies läuft so ab wie das besitzergesteuerte Zeichnen anderer Komponenten, für welches Sie in den Kapiteln 4.4.4 und 4.5.4 einige Beispiele finden.

Die Kapitel 4.6.2 und 4.6.3 beschreiben, wie Sie die beiden zuletzt genannten Aufgaben mit der CLX bewältigen können. Davor erläutert Kapitel 4.6.1 die allgemeinen Grundlagen, die in der CLX-Unit *QMenus* gelegt werden. Kapitel 4.6.4 befasst sich schließlich mit den Befehlslisten alias *TActionList*-Komponenten.

Popup-Menüs sind ein weiteres Menü-Thema, bei dem Sie es normalerweise jedoch weniger mit der Unit *Menus* zu tun bekommen (siehe beispielsweise Kapitel 6.2.4 und 6.8.3).

4.6.1 Die Unit Menus

Menüs unterscheiden sich insofern stark von den anderen visuellen Komponenten, als die Menüklassen nicht von *TControl*, sondern direkt von *THandleComponent* abgeleitet sind. Die beiden Menütypen – Hauptmenü und Popup-Menü (*TMainMenu* und *TPopupMenu*) – haben die gemeinsame Basisklasse *TMenu*. Die Klasse für die einzelnen Menüpunkte, *TMenuItem*, ist unabhängig davon und direkt abgeleitet von *THandleComponent*.

Ein wichtiger Funktionsbereich der Unit *Menus* befindet sich außerhalb dieser Klassen in Form einer Funktionssammlung, die das dynamische Konstruieren von Menüs und die Definition von Tastenkürzeln noch weiter vereinfacht, als das mit den Menüklassen allein möglich wäre (so ersetzt ein einziger Funktionsaufruf einen Konstruktoraufruf und mehrere Zuweisungen an Properties).

TMenuItem

Bei der Arbeit mit Objektinspektor und Menü-Designer sieht eine *MenuItem*-Komponente noch sehr harmlos aus: Der Objektinspektor zeigt nur die Properties, die für die

Gestaltung und die Funktion des einzelnen Menüpunkts notwendig sind. Diese Properties von *TMenuItem* sind bereits in Kapitel 1.4.8 erklärt, so dass wir hier gleich unter diese Oberfläche gehen können.

Dort zeigt sich, dass die Klasse *TMenuItem* weit mehr ist als nur ein Container für die einzelnen Menüattribute wie Text, Aktivierung, Markierung etc. Alle *MenuItems*, die zu einer Menükomponente gehören, sind miteinander in einer baumartigen Struktur verknüpft (siehe Abbildung 4.5).

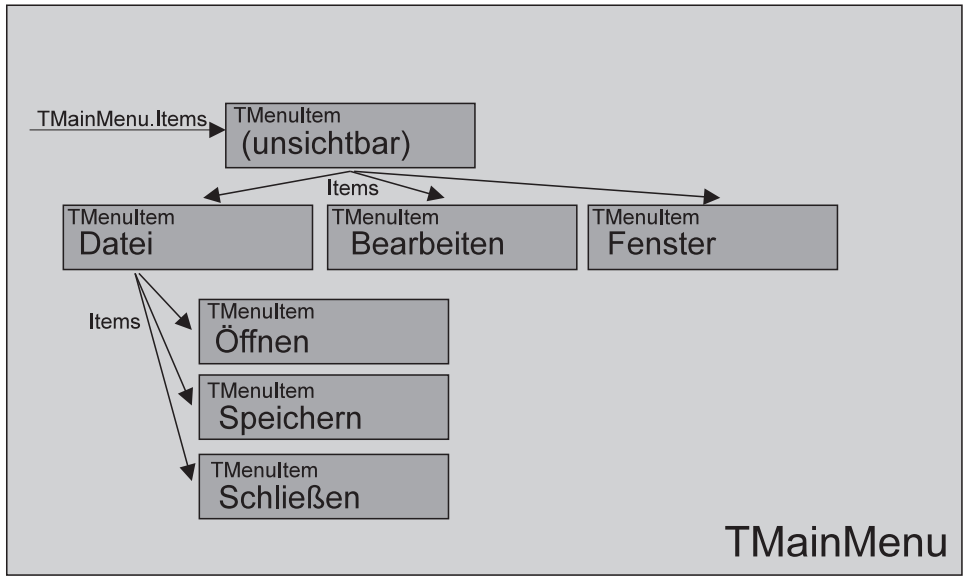


Abbildung 4.5: Aufbau der Menüstruktur mit Komponenten

Die Liste der Menüpunkte

In einem von 0 an indizierten Property namens *Items* speichert ein *TMenuItem*-Objekt alle Menüpunkte (wieder als *TMenuItem*-Objekte), die ihm untergeordnet sind. So enthält beispielsweise die *TMenuItem*-Komponente für den Hauptmenüpunkt DATEI einer normalen Anwendung die Menüpunkt-Komponenten für ÖFFNEN, SPEICHERN, SPEICHERN UNTER... usw. Wenn diese Menüpunkte ihrerseits keine weiteren Unterpunkte besitzen, dann bleibt ihr *Items*-Property leer.

In umgekehrter Richtung sind alle *MenuItems* über das Property *Parent* verknüpft. Es weist zu jedem Menüpunkt den übergeordneten Menüpunkt aus. Die Wurzel des Menübaumes wird von einem einzelnen *TMenuItem*-Objekt gebildet, dessen *Parent*-Property *nil* ist. Es beinhaltet in seinem *Items*-Property alle Menüpunkte der obersten Ebene, die der Benutzer am Bildschirm sehen kann, ist aber selbst für den Benutzer unsichtbar.

Viele weitere Methoden und ein Property helfen bei der Verwaltung der *Items*-Liste. Sie stimmen in Namen und Funktion mit den entsprechenden Methoden der Klasse *TList* überein:

Element	Bedeutung
Property Count	Zahl der Menüpunkte
Insert(Index, Item)	fügt *Item* an Position *Index* in die Liste ein.
Delete(Index)	löscht den Menüpunkt an Position *Index*.
IndexOf(Item)	stellt fest, ob sich *Item* bereits in der Liste befindet, und gibt die Position zurück.
Add(Item)	hängt *Item* an das Ende der Liste an.
Remove(Item)	löscht *Item* aus der Liste.

Diese Funktionen können Sie beliebig zur Laufzeit des Programms aufrufen, um die Struktur des Menüs anzupassen. Wir kommen in Kapitel 4.6.2 auf diese Funktionen zurück, wenn es um die dynamische Erzeugung eines Menüs der zuletzt geöffneten Dateien geht.

TMenu, TPopupMenu und TMainMenu

Dadurch, dass die gesamte Menüstruktur bereits in einem *TMenuItem*-Baum gespeichert ist, wird die Klasse *TMenu* stark entlastet: Sie benötigt lediglich ein Property *Items*, das die unsichtbare Wurzel des Menübaums als *TMenuItem*-Komponente enthält (zur Erinnerung: im Objektinspektor können Sie über dieses Property den Menü-Designer aufrufen). Die beiden Klassen für Haupt- und Popup-Menüs fügen dem noch einige weitere Funktionen hinzu: *TMainMenu* erweitert *TMenu* um Funktionalität zum Verschmelzen von Menüs (siehe Kapitel 6.7.3); *TPopupMenu* enthält einige Properties, die das Verhalten des Popup-Menüs steuern (z.B. das Property *AutoPopup*, siehe Kapitel 6.2.4).

Als Beispiel für die Verwendung des *Items*-Properties kann hier der Tastenkürzel-Manager aus Kapitel 4.6.3 dienen, der die Punkte eines Menüs in einer ListBox aufführt. Die folgenden Zeilen hängen den Text (*Caption*) aller Menüpunkte einer *TMainMenu*-Komponente an die Liste einer *TListBox*-Komponente an. Die *TMainMenu*-Komponente ist in *FMenu* angegeben:

```
for i := 0 to FMenu.Items.Count-1 do
  ListBox1.Items.Add(FMenu.Items[i].Caption);
```

Wie in diesem Beispiel benötigen Sie die *TMainMenu*-Komponente oft nur, um an das *TMenuItem*-Objekt *Items* zu kommen. Insofern sind diese drei Zeilen eher Beispiel für die Verwendung von *TMenuItem* als für den Umgang mit *TMainMenu*.

Die Hilfsfunktionen *R26*

Wenn Sie ein Menü mit Hilfe der von *TMenuItem* bereitgestellten Methoden und Properties dynamisch zur Laufzeit aufbauen wollten, müssten Sie in etwa die folgenden Schritte unternehmen:

```
var
  MenuItem, MainMenu: TMenuItem;
begin
  MainMenu := TMenuItem.Create(...);
  MenuItem := TMenuItem.Create(...);
  MenuItem.Caption := 'Menutext';
  MainMenu.Add(MenuItem);
  ...
```

Dies wäre letztlich komplizierter, als wenn Sie das Menü mit den entsprechenden Qt-Funktionen zusammengestellt hätten. Einige Hilfsfunktionen der CLX vereinfachen die Sache jedoch deutlich, wenn ihre große Parameterzahl auch etwas unübersichtlich ist. Die folgende Tabelle sortiert diese Funktion nach dem Objekt, das sie erzeugen:

Objekt	**hergestellt von**	**Parameter**
TMainMenu	NewMenu	Owner: TComponent; Name: String; Items: array of TMenuItem
TPopupMenu	NewPopupMenu	Owner, Name, Alignment, AutoPopup, Items
TMenuItem mit Unterpunkten	NewSubMenu	Caption: String; helpCtx: Word; Name: String; Items: array of TMenuItem
TMenuItem ohne Unterpunkte	NewItem	Caption, ShortCut, Checked, Enabled, OnClick, helpCtx, Name
TMenuItem als Trennlinie	NewLine	(keine)

Die Bedeutung der Parameter stimmt mit denen der Properties von *TMenuItem* überein. Zugegebenermaßen sieht diese Tabelle auf den ersten Blick noch nicht nach einer ereinfachung aus. Erst in einem Beispielprogramm zeigt sich der Nutzen dieser Funktionen.

iele Funktionen erwarten als letzten Parameter ein offenes Array von Menüpunkten, das Sie in Object Pascal direkt aus geschachtelten Funktionsaufrufen angeben können. Beispielsweise können Sie einem Formular wie folgt ein neues Hauptmenü zuweisen (der Code ist dem Beispielprojekt *HLTest* (Demo für *THistoryList*) entnommen):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { Unsichtbare Wurzel des Menüs: }
  Menu := NewMenu(self, 'unsichtbarer Text', [
    { Liste der Hauptmenüpunkte (hier nur der Punkt "Demo"): }
    NewSubMenu('&Demo', 0, 'NoName', [
```

```
        { Liste der Menüpunkte des "Demo"-Menüs: }
        NewItem('&Tastenkürzel...', 0, False, True,
                P1Click, 0, ''),
        NewItem('&Zum Symbol verkleinern', 0, False, True,
                P2Click, 0, ''),
        NewItem('&Farbe wechseln', 0, False, True,
                P3Click, 0, '')
      ]) { Ende von NewSubMenu "Datei" }
      { hier könnten weitere Hauptmenüpunkte folgen... }
    ]); { Ende von NewMenu "Hauptmenü" }
  end;
```

Der Text des Hauptmenüs ist am Bildschirm nicht sichtbar, wohl aber der einzige Menüpunkt des Hauptmenüs (DEMO) und seine drei Unterpunkte. Durch den dynamischen Aufbau der Menüpunkt-Arrays mit der Object-Pascal-Schreibweise *[...]* (siehe *Offene Arrays* in Kapitel 2.5.3) genügt für jeden Menüpunkt eine einzige Anweisung.

Die Struktur des Menüs kontrollieren Sie mit der Position dieser Aufrufe. Wenn Sie beispielsweise den Menüpunkt FARBE WECHSELN als Hauptmenüpunkt neben DEMO sehen wollen, verschieben Sie den entsprechenden *NewItem*-Aufruf einfach eine Schachtelungsebene nach oben hinter die beiden nächsten geschlossenen Klammern (eine runde und eine eckige) und passen die Kommasetzung entsprechend an. Um Trennlinien einzufügen, verwenden Sie statt *NewItem* einfach *NewLine*.

Das obige Beispiel verknüpft jeden Menüpunkt mit einer Methode des Formulars (*P1Click*, *P2Click*, *P3Click*). Falls Sie eine solche nicht schon vom Objektinspektor aus erstellt haben, müssen Sie sie vollständig von Hand erstellen. Die folgende Methode gehört zum obigen Beispiel und dem Menüpunkt ZUM SYMBOL VERKLEINERN (für Weiteres siehe *HLTest*).

```
procedure TForm1.P2Click(Sender: TObject);
begin
  Application.Minimize;
end;
```

Anders als in diesem Beispiel werden bei einer dynamischen Menüpunktliste oft alle Menüpunkte durch eine einzige Methode bearbeitet. Entweder muss diese Methode die verschiedenen Punkte über das *Tag*-Property unterscheiden, oder sie richtet sich nach dem Inhalt des *Caption*-Properties. Letzteres ist im Beispiel von Kapitel 4.6.2 der Fall.

Tastenkürzel

Ein besonderes Highlight der Unit *QMenus* ist das Management der Tastenkürzel. Zwar können Sie auch diese bereits bequem im Objektinspektor festlegen, jedoch werden Tastaturkürzel noch häufiger zur Laufzeit angepasst als Menüs. Auch hier gibt die Kylix-IDE selbst wieder ein gutes Beispiel: Sie bietet im Umgebungsdialog vier verschiedene Tastaturbelegungen an. Wenn Sie hier eine andere Belegung auswählen, werden die geänderten Tastenkürzel automatisch auch im Menü angezeigt.

Damit reizt Kylix jedoch die Fähigkeiten der eigenen VCL (die Kylix-IDE ist ja noch mit der VCL von Windows programmiert) noch keineswegs aus. Und auch hier stimmen CLX und VCL überein: Auch die CLX zwingt Sie nicht dazu, die gesamte Tastaturbelegung auf einmal auswechseln, sondern erlaubt Ihnen, das Tastenkürzel jedes einzelnen Menüpunkts zu ändern. Dazu benötigen Sie noch nicht einmal Tastencodes, sondern nur einen String, der diesen Code angibt, z.B. *'Strg+F1'*.

Das Property für das Tastenkürzel ist ja bereits im Objektinspektor ersichtlich: Jede *TMenuItem*-Komponente verfügt über ein *ShortCut*-Property, das Sie auch zur Laufzeit verändern können. Die CLX sorgt dann automatisch dafür, dass das angegebene Kürzel im Menü erscheint und dass es als Tastenkürzel funktioniert.

ShortCut hat jedoch den Typ *TShortCut*, bei dem es sich um einen Zahlencode handelt. Durch die Funktionen *TextToShortCut* können Sie diesen Code aus einem einfach zu verstehenden String berechnen lassen. Die Funktion *ShortCut* generiert den *TShortCut*-Wert aus einem Tastencode und einem *TShiftState*-Parameter, der den Status verschiedener umschaltbarer Tasten angibt. Die folgende Tabelle erklärt die Parameter und listet die beiden Umkehrfunktionen auf, die einen *TShortCut*-Code wieder zurück in einen lesbaren String oder in Tastencode und *ShiftState*-Angabe umwandeln:

Funktion	Beschreibung
function ShortCut(Key: Word; Shift: TShiftState): TShortCut;	liefert den Code für die in *Key* angegebene Taste, wenn diese mit den in *Shift* angegebenen Zusatztasten gedrückt wird.
ocedure ShortCutToKey(ShortCut; var Key; var Shift);	liefert den Code, der dem *Text* entspricht. Gültige Parameter sind beispielsweise: 'F1', 'Strg+A' oder 'Alt+0'.
function ShortCutToText(ShortCut: TShortCut): String;	wandelt einen Tastencode in einen String um.
function TextToShortCut(Text: String): TShortCut;	wandelt einen Tastencode zurück in die Parameter der Funktion *ShortCut*.

4.6.2 Dynamische Menüerweiterungen

Dieses Kapitel gibt ein praktisches Anwendungsbeispiel für die im letzten Kapitel beschriebenen Hilfsfunktionen und für die Menüpunkteliste in *TMenuItem*. Dabei geht es um die dynamische Menüerweiterung der Klasse *THistoryList*, die Sie in der Unit *HList* auf der CD-ROM finden.

Die dynamische Erweiterung von Menüs ist wahrscheinlich am häufigsten notwendig, um Listen von Dateien oder Fenstern an bestehende Menüs anzuhängen. So finden Sie im Datei-Menü der Kylix-IDE eine Liste der zuletzt geladenen Projekte und im Fenstermenü einer MDI-Anwendung eine Liste der offenen Fenster.

Erweitern des Dateimenüs

Um ein Dateimenü zu erweitern, benötigen Sie zuerst das *TMenuItem*-Objekt, das für das Dateimenü steht. Solange dieses Dateimenü sich immer an derselben Stelle befindet, können Sie einen festen Ausdruck angeben, um das gewünschte *TMenuItem*-Objekt zu erhalten. Falls sich das Dateimenü z. B. ganz links befindet, lautet dieser: *Menu.Items.Items[0]*. *Menu* ist dabei das Hauptmenü-Property des Formulars, *Items* ist die unsichtbare Wurzel des Menübaumes und in *Items.Items* sind schließlich die Hauptmenüpunkte gespeichert. Da die Liste *Items* in der Klasse *TMenuItem* als Standard-Array-Property deklariert ist, erhalten Sie das Dateimenü auch mit *Menu.Items[0]*. Den so erhaltenen Menüpunkt können Sie nun um die Dateinamen erweitern, indem Sie mit seiner *Add*-Methode neue *TMenuItem*-Objekte anfügen. Die dynamische Dateiliste des TreeDesigners sehen Sie in Abbildung 4.6.

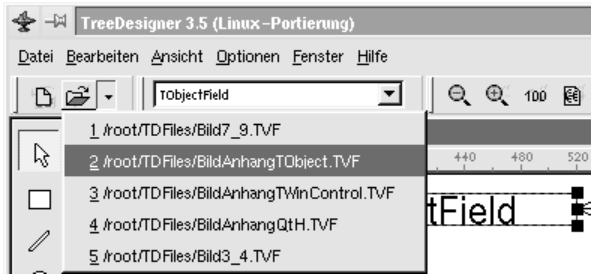


Abbildung 4.6: Die Datei-History-Liste des TreeDesigners kann über das Datei-Menü und über den Öffnen-Schalter der Symbolleiste aufgerufen werden.

Dateilisten im TreeDesigner

Beim TreeDesigner genügt es nicht, die Dateiliste an das Dateimenü des Hauptfensters anzuhängen, weil dieses beim Verschmelzen des Menüs mit einem Kindfenster-Menü (siehe Kapitel 6.7.3) durch das Dateimenü dieses Kindfensters ersetzt wird. Der TreeDesigner muss daher die Dateiliste jedes Mal, wenn ein anderes Kindfenster aktiviert wird, mit dessen Dateimenü verknüpfen. Erst wenn einmal alle Kindfenster geschlossen sind, hängt er die Liste wieder an das Dateimenü des Hauptfensters an. Zuständig hierfür sind die Methoden *TDocumentForm.MdiActivate* und *TDocumentForm.FormClose*, im Folgenden sind diese Details jedoch ohne Bedeutung.

THistoryList

Die Klasse *THistoryList* aus Kapitel 4.1.2 verfügt über die Fähigkeit, die komplette von ihr verwaltete Stringliste an ein Menü anzuhängen. Wie schon erwähnt, ist dies meistens nur für die Liste der zuletzt geladenen Dateien notwendig. Bei der Verwendung einer *HistoryList* müssen Sie im Wesentlichen nur das *MenuItem*-Property dieser Liste

setzen, die *THistoryList* hängt dann automatisch alle Strings an den angebenenen Menüeintrag an und aktualisiert das Menü bei jeder Änderung der History-Liste (auf die Details dieser Verwendung kommen wir später noch einmal zurück).

Da viele Anwendungen eine Dateiliste nicht nur im Hauptmenü, sondern auch als Popup-Menü für einen Schalter einer Symbolleiste anzeigen, verfügt *THistoryList* auch über das Property *PopupMenu*. Wenn es gesetzt ist, schreibt *THistoryList* die Dateinamen zusätzlich in dieses Popup-Menü (dieses muss vorher allerdings leer sein, da *THistoryList* es komplett neu aufbaut).

Zum Verständnis der Menümethoden benötigen Sie die folgenden Informationen über *THistoryList*:

```
type
  { Event, das bei Anwahl eines der automatisch zum Menü
    hinzugefügten Einträge ausgelöst wird: }
  TListMenuEvent = procedure(Sender: TObject; MenuText: string)
                   of object;

  THistoryList = class(TStringList)
  { Stringliste, die sich selbst an ein Menü anhängen kann und
    sich in die Registry/eine INI-Datei speichern und von dort
    wieder lesen kann }
  private
    FMenuItem: TMenuItem;
    FPopupMenu: TPopupMenu;
    MenuSizeBefore: Integer; { Speichert die vorherige
                  Menügröße zur späteren Rekonstruktion }
    FOnAutoItemClick: TListMenuEvent;

    procedure SetMenuItem(MenuItem: TMenuItem);
    procedure SetPopupMenu(PopupMenu : TPopupMenu);
    procedure AddMenuItems; // ruft AddToMenuItem und AddToPopupMenu auf
      procedure AddToMenuItem; // Liste nur an den Hauptmenüpunkt anhängen
      procedure AddToPopupMenu; // Liste nur in das Popup-Menü schreiben
    procedure RemoveMenuItems; // ruft beide Remove...-Methoden auf
      procedure RemoveFromMenuItem; // entfernt die Liste aus FMenuItem
      procedure RemoveFromPopupMenu; // entfernt die Liste aus FPopupMenu
    procedure DoAutoItemClick(Sender: TObject);
       { empfängt die OnClick-Ereignisse }
  public
    { Funktionsbereich Menü: sobald "Menu" einmal gesetzt ist,
      passt die Liste das Menü bei jeder Veränderung an. }
    property MenuItem: TMenuItem read FMenuItem write SetMenuItem;
    property PopupMenu : TPopupMenu read FPopupMenu write SetPopupMenu;
    property OnAutoItemClick: TListMenuEvent read FOnAutoItemClick
                                          write FOnAutoItemClick;
    ...
```

Bei der Menüerweiterung spielen vier Datenelemente dieser Klasse eine wichtige Rolle:

- *Strings*, die von *TStringList* geerbte Stringliste, die die Dateinamen enthält (oder die anderen Strings, die an das Menü angehängt werden sollen). Wie diese Liste verwaltet wird, lesen Sie in Kapitel 4.1.2; in diesem Kapitel können wir diese Stringliste als fest vorgegeben betrachten.
- Das Menü, an das die Strings dieser Liste angehängt werden sollen, speichert *THistoryList* im Datenfeld *FMenuItem*.
- Für das gleichzeitig verwendbare Popup-Menü gibt es entsprechend eine *FPopupMenu*-Variable.
- *MenuSizeBefore* speichert die Zahl der Menüpunkte, bevor die Liste an *FMenuItem* angehängt wird, damit die Liste später wieder entfernt und aktualisiert werden kann. Die Zahl der Menüpunkte des Popup-Menüs wird nicht gespeichert, da dieses Menü wie erwähnt immer neu aufgebaut wird, also gar keine anderen Punkte enthalten soll.

Wie Sie der obigen Deklaration entnehmen können, wird *FMenuItem* von außen über das Property *MenuItem* angesprochen. Jedes Mal, wenn *MenuItem* von außen gesetzt wird, wird die Methode *SetMenuItem* aufgerufen. *SetMenuItem* löscht zuerst mit *RemoveFromMenuItem* alle zuletzt angefügten Menüpunkte, speichert die neue Zahl der Menüpunkte in *MenuSizeBefore* und fügt die derzeitige Liste mit *AddToMenuItem* zum neuen Menü hinzu:

```
procedure THistoryList.SetMenuItem(MenuItem: TMenuItem);
begin
  { ein eventuell schon verändertes Menü in seinen alten
    Zustand zurückversetzen: }
  if Assigned(FMenuItem) then
    RemoveFromMenuItem;
  FMenuItem := MenuItem; { Property-zugehörige Variable setzen }
  MenuSizeBefore := MenuItem.Count; { bisherige Menügröße speichern }
  AddToMenuItem; { ab sofort hält THistoryList dieses Menü aktuell }
end;
```

Menüpunkte hinzufügen und löschen *R67*

Wir sehen uns nun die von *SetMenu* aufgerufenen Methoden *AddToMenuItem* und *RemoveFromMenuItem* an. *AddToMenuItem* kann davon ausgehen, dass das Menü noch in seinem Ursprungszustand ist, dass also alle eventuell früher schon angefügten Menüpunkte mit *RemoveFromMenuItem* wieder entfernt worden sind.

Hinweis: Dies wird dadurch sichergestellt, dass alle Aufrufer von *AddToMenuItem* oder *AddMenuItems* vorher *RemoveFromMenuItem* aufrufen. Die Aufrufer dieser Methoden sind *SetMenu* (siehe oben) und *AddString*(Kapitel 4.1.2), von außen können diese Methoden nicht aufgerufen werden, da sie privat sind.

AddToMenuItem hängt zuerst eine Trennlinie, erzeugt durch die Funktion *NewLine*, an die vorhandenen Menüpunkte an. Danach durchläuft sie in einer Schleife alle Strings der Liste und erzeugt mit der Funktion *NewItem* für jeden String einen neuen Menüpunkt. Zum Anfügen der Trennlinie und der Dateinamen verwendet sie die *TMenuItem*-Methode *Add*:

```
procedure THistoryList.AddToMenuItem;
var
  i: Integer;
begin
  if Assigned(FMenuItem) then begin
    FMenuItem.Add(NewLine); { mit einem Separator abtrennen }
    for i := 0 to Count-1 do
      { Der Menütext wird mit der komfortablen Format-Funktion
        zusammengesetzt: }
      FMenuItem.Add(NewItem(Format('&%d %s', [i+1, Strings[i]]), 0,
                            False, True, DoAutoItemClick, 0, ''));
  end;
end;
```

AddToMenuItem nummeriert die neuen Menüpunkte durch, wobei es mit der Ziffer 1 beginnt. Sie übergibt der Methode *NewItem* einen Menütext, der sich zusammensetzt aus dem Zeichen »&« (für die Unterstreichung), aus der Nummer des Eintrags, aus einem Leerzeichen und schließlich aus dem Dateinamen (zur Funktion *Format* siehe Kapitel 2.8.4).

RemoveItems löscht die Einträge wieder. Sie löscht so lange den letzten Eintrag der *TMenuItem*-Liste, bis die Liste dadurch so klein geworden ist wie beim ursprünglichen Menü (*MenuSizeBefore*).

```
procedure THistoryList.RemoveFromMenuItem;
begin
  { alle Menüeinträge ab der gespeicherten Position
    MenuSizeBefore wieder entfernen: }
  if Assigned(FMenuItem) then
    while FMenuItem.Count > MenuSizeBefore do
      FMenuItem.Delete(FMenuItem.Count-1);
end;
```

Diese beiden Methoden werden jedes Mal aufgerufen, wenn die History-Liste verändert wird (siehe Methode *THistoryList.AddString* in Kapitel 4.1.2).

Hinweis: Die Methoden *AddToPopupMenu* und *RemoveFromPopupMenu* funktionieren ähnlich und sind daher nicht abgedruckt.

Verknüpfen der OnClick-Ereignisse

NewItem erwartet im vierten Parameter eine Methode, die bei jedem *OnClick*-Ereignis des neuen Menüpunkts aufgerufen werden soll. Die obige Methode *AddToMenuItem* übergibt in diesem Parameter jedes Mal die Methode *DoAutoItemClick*. Diese wird also jedes Mal aufgerufen, wenn zur Laufzeit einer der hinzugefügten Strings bzw. Dateinamen aus dem Menü ausgewählt wird.

Wie in der allgemeinen Beschreibung schon erwähnt, steht diese Methode nun vor dem Problem, zwischen den einzelnen Menüpunkten zu unterscheiden. Sie benötigt dazu lediglich das *Caption*-Property des Menüpunkts, der im Parameter *Sender* übergeben wird:

```
procedure THistoryList.DoAutoItemClick(Sender: TObject);
var
  Text: string;
  Index: Integer;
begin
  if Assigned (FOnAutoItemClick) then begin
    { Caption beinhaltet den Menüpunkt-Text: }
    Text := (Sender as TMenuItem).Caption;
    { Text sieht so aus:"&12 MenuText"
      das "&" muss übersprungen werden, dann folgt bis
      zum Leerzeichen der Index, der gewählt wurde. }
    Index := StrToInt(Copy(Text, 2, Pos(' ', Text)-2));
    { Index - 1 ergibt den Index in der internen Stringliste: }
    if Index > 0 then
      FOnAutoItemClick(FMenuItem, Strings[Index-1]);
  end;
end;
```

DoAutoItemClick findet zuerst die Nummer des Menüpunkts heraus, die von *AddToMenuItem* vor den Text geschrieben wurde (mit Hilfe der Standardfunktionen *Copy, Pos* und *StrToInt*). Mit Hilfe dieser Nummer findet sie in ihrer eigenen Liste schließlich den zugehörigen String (bzw. Dateinamen).

Hinweis: Natürlich hätte die Methode den Dateinamen auch direkt im Property *Caption* finden können, allerdings wurde hier zuerst der Index gesucht in der Annahme, dass er sich vielleicht später einmal sinnvoll verwerten lässt.

Was schließlich mit dem angeklickten String oder Dateinamen geschehen soll, kann das *THistoryList*-Objekt nicht wissen, dafür muss der Benutzer dieses Objekts sorgen. *THistoryList* stellt diesem dafür das Event *OnAutoItemClick* zur Verfügung. Innerhalb

der Methode *DoAutoItemClick* wird dieses Menü über den Methodenzeiger *FOnAuto ItemClick* ausgelöst; mehr über das Auslösen von eigenen Events finden Sie in Kapitel 8.3.3.

erwendung von THistoryList *R49*

Wenn Sie *THistoryList* in eigenen Programmen verwenden wollen, müssen Sie die folgenden vier hier beschriebenen Schritte nachvollziehen:

1. Initialisierung und Laden der History-Liste, z. B. beim *OnCreate*-Event
2. Schreiben einer Methode für die automatischen Menüpunkte, also für das Ereignis *OnAutoItemClick*
3. Erweitern der History-Liste an den geeigneten Stellen
4. Speichern der History-Liste, z. B. im *OnDestroy*-Event

Der TreeDesigner aus Kapitel 6 demonstriert die Anwendung von *THistoryList* zur Speicherung einer Liste der zuletzt geladenen Dateien. Die genannten vier Schritte lassen sich in der Unit für das MDI-Hauptformular *MdiForm* wiederfinden und sind im Folgenden einzeln besprochen.

Initialisieren und Laden der History-Liste

In der *OnCreate*-Methode des TreeDesigner-Hauptformulars geschehen die folgenden unter Punkt 1 erwähnten Dinge:

```
{ Auszug aus MainFormCreate: }
FileList := THistoryList.Create;
FileList.LoadFromIni(IniPath+'TREEDSGN.INI', 'FILES');
FileList.PopupMenu := RecentFilesPopup;
FileList.MenuItem := TLM_File;
FileList.OnAutoItemClick := AutoItemClick;
```

FileList, ein *THistoryList*-Objekt, wird wie üblich per Konstruktoraufruf erzeugt. Dann werden die Namen der zuletzt benutzten Dateien, die beim letzten Schließen des Programms gespeichert wurden, wieder geladen. Die Methode *LoadFromIni* erwartet hierzu im ersten Parameter den Namen einer INI-Datei und im zweiten den Namen des Abschnitts, unter dem die Dateinamen gespeichert werden sollen.

Als Nächstes setzt die Methode die Properties, die die zu erweiternden Menüs angeben. *Datei1* ist der Name des *TMenuItem*-Objekts für den Menüpunkt DATEI (um diesen Namen im Objektinspektor zu sehen, müssen Sie den Menü-Editor aufrufen). *RecentFilePopup* ist der Name einer *TPopupMenu*-Komponente.

Die letzte Zeile im obigen Codeausschnitt setzt das Property für das Event *OnAuto ItemClick*, das von der Methode *DoAutoItemClick* aufgerufen wird, jedes Mal, wenn einer der automatisch erzeugten Menüpunkte aufgerufen wird.

Schreiben einer Methode für OnAutoItemClick

Die Methode des Formulars für dieses Ereignis heißt *AutoItemClick*. Wie aus der Deklaration von *TListMenuEvent* auf Seite 472 ersichtlich, ist der erste Parameter der übliche *Sender*, im zweiten Parameter befindet sich der Menütext (ohne die vorangehende Ziffer).

AutoItemClick weiß, dass es sich bei diesem Menütext nur um einen Dateinamen handeln kann, und lädt diesen mit der hier nicht weiter interessanten Methode *LoadFromFile* in ein Fenster, das mit der hier ebenfalls uninteressanten Methode *NewWindow* geöffnet wird. Punkt 2 der obigen Aufzählung sieht damit im TreeDesigner wie folgt aus:

```
procedure TMainForm.AutoItemClick(Sender: TObject; FileName: string);
begin
  NewWindow.LoadFromFile(FileName);
end;
```

Wie die Dateinamen in die Liste kommen

Wir sind nun beim dritten Schritt der obigen Aufzählung angekommen: beim Erweitern der Dateiliste an den geeigneten Stellen. Hierfür stellt *THistoryList* die Methode *AddString* zur Verfügung. Sie können diese aufrufen, wenn eine Datei unter neuem Namen gespeichert wird oder wenn eine neue Datei geöffnet wird. Im TreeDesigner ist die Methode *MDIOpenClick* für das Öffnen der Datei zuständig:

```
procedure TMainForm.MDIOpenClick(Sender: TObject);
var
  NewChild: TDocumentForm;
begin
  if OpenDialog1.Execute then begin
    NewChild := NewWindow;
    NewChild.LoadFromFile(OpenDialog1.FileName);
    { Datei-History-Liste erweitern: }
    FileList.AddString(OpenDialog1.FileName);
  end;
end;
```

AddString erweitert die History-Liste um den neuen Dateinamen und aktualisiert das Menü, indem sie die Methoden *AddMenuItems* und *RemoveMenuItems* aufruft. Weitere Informationen zu *THistoryList* und zur Methode *AddStrings* finden Sie in Kapitel 4.1.2.

Genauso können Sie die History-Liste erweitern, wenn eine Datei unter neuem Namen gespeichert wird (im TreeDesigner wird das Speichern vom Dokumentfenster übernommen).

Speichern in einer INI-Datei

Als letzter Schritt bleibt nun noch das »*Speichern der History-Liste, z.B. im OnDestroy-Event*«, denn ein wesentlicher Vorteil der History-Liste soll ja sein, dass der Anwender auch auf Dateien schnell zugreifen kann, die er in früheren Sitzungen geladen hat. Als Gegenstück zum *LoadFromIni*-Aufruf in *FormCreate* findet daher in der Methode zum *OnDestroy*-Event ein Aufruf von *SaveToIni* statt:

```
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FileList.SaveToIni(IniPath+'TREEDSGN.INI', 'FILES');
  FileList.Free;
end;
```

Der Aufruf von *Free*, der den manuellen *Create*-Aufruf in *FormCreate* wieder aufhebt, sollte nicht fehlen, denn da *FileList* keine Komponente ist, wird sie bekanntlich nicht automatisch von der CLX gelöscht.

4.6.3 Ein dynamischer Tastenkürzeleditor

Mit den in Kapitel 4.6.1 beschriebenen Funktionen können wir leicht einen Tastenkürzeleditor schreiben, der in alle Anwendungen eingebunden werden kann und der dort die Tastenkürzel von beliebigen Menüs modifizieren kann. Ein solcher Editor sollte natürlich in der Lage sein, die Menükonfiguration in einer Datei zu speichern.

Auf der CD-ROM finden Sie den Tastenkürzeleditor als Formular in der Unit *HKeyForm* und als Komponente *THotkeyManager* in der Unit *HKeyMan* (siehe Kapitel 8.7); eine Komponente der Klasse *THotkeyManager* ist im Dokumentformular des *TreeDesigners* aus Kapitel 6 eingebunden, zur Laufzeit rufen Sie den Editor dort über das Menü PTIONEN | TASTENKÜRZEL-EDITOR... auf.

Da die Methoden des Tastenkürzeleditors mit ähnlichen Mitteln arbeiten wie die Methoden zur dynamischen Menüerweiterung aus dem letzten Kapitel, ist hier nur ein kleiner Teil des Quelltextes abgedruckt.

astenkürzel für Aktionslisten

Ebenfalls nicht abgedruckt sind die Programmteile, die den Editor befähigen, auch die Tastenkürzel einer Aktionsliste zum Editieren freizugeben. Ob ein Menü oder eine Aktionsliste editiert wird, entscheidet sich daran, welches Property von *THotkeyEditor* Sie setzen: Bei Zuweisung an das Property *Menu* wird das Property *ActionList* automa-

tisch gelöscht und umgekehrt. Wie die Abbildung zeigt, verwendet der Editor in beiden Fällen denselben Fensteraufbau, lediglich die Beschriftungen sind unterschiedlich.

Ein *ActionList*-Property finden Sie natürlich auch in der Komponente *THotkeyManager*. Wenn Sie alle Aktionen Ihrer Anwendung in einer Aktionsliste verwalten, kann *THotkeyManager* leicht zum Editieren aller Tastenkürzel einer Anwendung dienen. Mit den Aktionslisten wird sich Kapitel 4.6.4 beschäftigen.

Bedienung

Abbildung 4.7 zeigt eine aktive Version des Editors. Seine erste Liste enthält schon nach dem Öffnen des Fensters die Punkte des Hauptmenüs (bzw. die Kategorien der Aktionsliste). Wählen Sie einen Eintrag dieser Liste aus, so füllt der Editor die zweite Liste mit den zugehörigen Untermenüpunkten (bzw. den Aktionen der gewählten Kategorie), wobei er die bereits aktiven Tastenkürzel in Klammern anzeigt. Im Editierfeld unter der ersten Liste geben Sie das Tastenkürzel ein, indem Sie einfach die entsprechenden Tasten drücken.

Das Dateimenü des Editors enthält die beiden Punkte *Laden* und *Speichern*, mit denen Sie manuell verschiedene Dateien mit Tastenkürzeln verwalten können; normalerweise sollte jedoch die in Kapitel 8.7 beschriebene automatische Speicherung der Tastaturbelegung ausreichen.

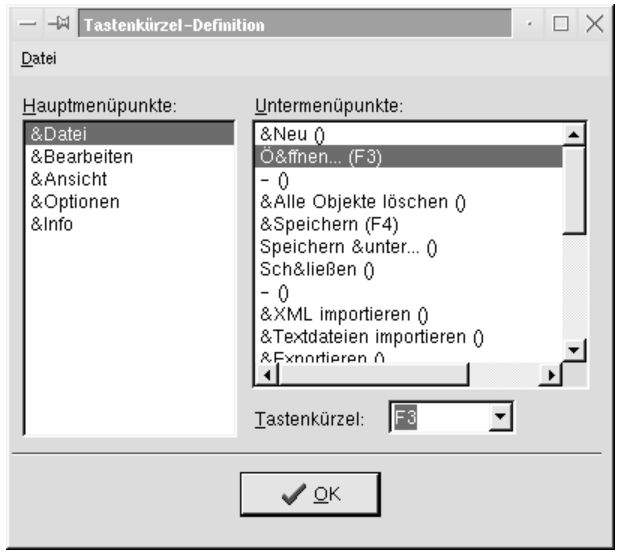


Abbildung 4.7: Der Tastenkürzeleditor in HKeyForm

Einbindung des Tastenkürzeleditors in eigene Programme

Kapitel 8.7 beschreibt, wie Sie den Tastenkürzeleditor über eine Komponente zur Entwurfszeit in das Formular einbinden. Auch ohne diese Hüllkomponente können Sie den Editor leicht in eigenen Programmen verwenden. Nachdem Sie die Unit *HKeyForm* optional in Ihr Projekt (PROJEKT | DEM PROJEKT HINZUFÜGEN) und die Unit in die *uses*-Anwendung der passenden Unit eingefügt haben, können Sie den Editor wie in der folgenden Methode aufrufen.

```
procedure TDocumentForm.Tastenkrzel1Click(Sender: TObject);
begin
  HotkeyEditor.Menu := Menu; { "Menu" ist das Formular-Property }
{ oder, für das Editieren der Aktionen einer Aktionsliste:
    HotkeyEditor.ActionList := ActionList1; }
  HotkeyEditor.ShowModal;
end;
```

Diese Zeilen setzen voraus, dass das Formular des Tastenkürzeleditors in den Projektoptionen als automatisch zu erzeugen angegeben ist. Mit einer dynamischen Erzeugung könnten Sie während der Zeit, in der der Editor nicht angezeigt wird, Ressourcen sparen, siehe dazu Kapitel 3.4.4.

Nach der Ausführung von *ShowModal* werden die neuen Tastenkürzel im Menü (bzw. bei entsprechender Verwendung einer Aktionsliste in allen Menüs) angezeigt und können sofort verwendet werden. Um die Tastenkürzel automatisch zu speichern, müssten Sie nun noch die Methoden *SaveToFile* und *LoadFromFile* an einer geeigneten Stelle aufrufen (z. B. bei *OnCreate* und *OnDestroy* oder direkt vor und hinter dem Aufruf von *ShowModal*, falls Sie den Tastenkürzeleditor auf diese Weise aufrufen). Ein Beispiel für den Aufruf der Speichern- und Lademethoden finden Sie in der Hüllkomponente für diesen Editor in Kapitel 8.7.

Implementierung

Wir wenden uns noch einigen Ausschnitten aus dem Code zu. In diesem kommen nun die in Kapitel 4.6.1 behandelten Tastenkürzelfunktionen zum Einsatz. Beim Aktualisieren der zweiten Liste verwendet *UpdateSubItems* die Funktion *ShortCutToText*, um das Tastenkürzel in eine lesbare Form umzuwandeln (*SubMenu* ist eine Variable des Formulars, sie zeigt immer auf das in der ersten Liste gewählte Menü):

```
procedure THotkeyEditor.UpdateSubItems;
{ Baut "ListBox2" neu auf und füllt sie mit den
  Punkten des aktuellen "SubMenu" }
var
  i: Integer;
  Index: Integer;
begin
  { gewählten Listeneintrag merken: }
```

```
    Index := ListBox2.ItemIndex;
    ListBox2.Clear;
    for i := 0 to SubMenu.Count-1 do begin
      ListBox2.Items.Add(
        SubMenu[i].Caption+' '
         +ShortCutToText(SubMenu[i].ShortCut)+')');
    end;
    { bisher gewählten Listeneintrag wiederherstellen: }
    ListBox2.ItemIndex := Index;
  end;
```

Die Angabe eines Tastenkürzels geschieht im Editor über eine Kombobox, die eine (erweiterungsfähige) Liste von einfachen Tastenkürzeln zur Wahl vorgibt (für VCL-Kenner: Die *THotkey*-Komponente der VCL basiert auf einem speziellen Windows-Control und steht daher in der CLX nicht zur Verfügung).

Wenn der Benutzer nun die aktuelle Auswahl der Kombobox ändert, muss das Programm den neuen gewählten String in einen *TShortCut*-Wert umrechnen. Hier kann sich die CLX-Methode *TextToShortCut* nützlich machen:

```
procedure THotkeyEditor.ComboBox1Change(Sender: TObject);
begin
  if Assigned(Selected) then begin
    Selected.ShortCut := TextToShortCut(ComboBox1.Text);
    UpdateSubItems;
  end;
end;
```

Dabei ist *Selected* eine Variable des Formulars, die immer auf den in der zweiten Liste gewählten Eintrag weist.

Speichern der Tastenkürzel

Wie in Kapitel 4.3 angekündigt, gibt es nun ein Beispiel für die Verwendung der Methoden von *TWriter* und *TReader* im Zusammenhang mit einem *TStream*-Objekt, insbesondere dem Schreiben und Lesen von Listen. Untersucht werden die Methoden des Tastenkürzeleditors, die das Speichern und das Laden der Tastenkürzel durchführen.

Zuerst zur Methode für das Speichern. *SaveToFile* schreibt einfach für jeden Menüpunkt, der ein Tastenkürzel hat, drei Zahlen in die Datei: den Index des Hauptmenüpunkts, zu dem der Menüpunkt gehört, den Index des Menüpunkts innerhalb des Untermenüs und den Zahlencode des Tastenkürzels. Geschachtelte Untermenüs werden also – wie beim Editieren auch – ignoriert. Auch hier müssen wieder alle Punkte eines *Items*-Properties in einer Schleife durchlaufen werden, dieses Mal sogar in zwei geschachtelten Schleifen.

```
procedure THotkeyEditor.SaveToFile(FileName: string);
var
  MainIndex, SubIndex: Integer;
  SubMenu: TMenuItem;
  CurrentItem: TMenuItem;
  S: TStream;
  W: TWriter;
begin
  S := TFileStream.Create(FileName, fmCreate);
  W := TWriter.Create(S, 2000);
  W.WriteListBegin;
  { für alle Hauptmenüpunkte: }
  for MainIndex := 0 to FMenu.Items.Count-1 do begin
    SubMenu := FMenu.Items[MainIndex];
    { für alle Untermenüpunkte: }
    for SubIndex := 0 to SubMenu.Count-1 do begin
      CurrentItem := SubMenu.Items[SubIndex];
      { nur speichern, wenn ein Tastenkürzel existiert: }
      if CurrentItem.ShortCut <> 0 then begin
        W.WriteInteger(MainIndex);
        W.WriteInteger(SubIndex);
        W.WriteInteger(CurrentItem.ShortCut);
      end;
    end;
  end;
  W.WriteListEnd;
  W.Free;
  S.Free;
end;
```

Da die *TWriter*-Methoden bereits in Kapitel 4.3.4 erläutert wurden, können wir uns sofort dem Laden der Datei zuwenden.

Lesen der Tastenkürzeldatei

Die Methode zum Lesen kann keine interessanten neuen Aktionen in Sachen Menüs mehr durchführen, dient hier also in erster Linie als Fortsetzung des *TReader/TWriter*-Beispiels und wird hier nur soweit dargestellt, wie es die Arbeit des *TReader*-Objekts betrifft. Dieses muss die Daten genauso aus der Datei lesen, wie sie in *SaveFile* vom *Writer* gespeichert wurden.

Eine Besonderheit ist, dass der Benutzer aus Versehen eine Datei laden kann, die für ein ganz anderes Menü gedacht war. Hierauf bereitet sich *LoadFromFile* mit einer Exception-Abfrage vor. Anstatt jeden aus der Datei geladenen Menüindex (*MainIndex* und *SubIndex*) daraufhin zu überprüfen, ob es ihn wirklich gibt, probiert die Methode einfach aus, ob die Verwendung der Menüpunkt-Indizes eine Exception hervorruft (das Schlüsselwort *try* ist hier also wörtlich zu nehmen):

```
// Auszug aus THotkeyEditor.LoadFromFile:
// [von SaveToFile abweichende Variablen:
//    R: TReader; Errors: Boolean; ShortCut: Integer;]
R := TReader.Create(S, 2000);
R.ReadListBegin;
while not R.EndOfList do begin
  MainIndex := R.ReadInteger;
  SubIndex := R.ReadInteger;
  ShortCut := R.ReadInteger;
  try
    SubMenu := FMenu.Items[MainIndex];
    CurrentItem := SubMenu.Items[SubIndex];
    CurrentItem.ShortCut := ShortCut;
  except
    on EListError do Errors := True;
  end;
end;
R.ReadListEnd;
R.Free;
```

Falls es in einem Fall zu einer *EListError*-Exception kommt, setzt *LoadFromFile* das Flag *Error*, damit sie am Schluss eine warnende Dialogbox anzeigen kann. Alle anderen Exceptions werden nicht behandelt und führen zu einem Abbruch der Methode.

Natürlich führt die Verwendung der *TWriter*- und *TReader*-Methoden nicht gerade zu übermäßiger Effizienz; da es aber bei den kleinen Tastenkürzeldateien auch so nicht zu Wartezeiten kommen sollte, ist dieses Problem eher theoretischer Natur.

4.6.4 Befehlslisten mit TActionList

Viele moderne Anwendungen stimmen darin überein, dass sie Menübefehle und Aktionsschalter in Symbolleisten gleich behandeln, insbesondere in beiden dieselben Bildsymbole verwenden und somit dem Benutzer den Menüpunkt, der über einen Schalter aufgerufen werden kann, zusammen mit diesem Schalter als eine Einheit zu präsentieren. Spätestens seit dieser neuen Entwicklung drängt es sich auch für den Entwickler auf, Menüpunkte und zugehörige Schalter gemeinsam zu behandeln. Nun kann man in Kylix zwar einem Schalter-*OnClick*-Ereignis dieselbe Methode zuweisen wie einem Menüpunkt-*OnClick*-Ereignis. Aber damit ist es ja nicht getan, denn folgende Dinge müssen Sie in diesem Fall immer noch getrennt festlegen, einmal für den Schalter und einmal für den Menüpunkt:

- die statischen Daten, die sich zur Laufzeit normalerweise nicht ändern: die Beschriftung (*Caption*), der Bildindex (*ImageIndex*), der Hinweistext (*Hint*) und der Hilfekontext (*HelpContext*)
- die dynamischen Daten, die sich zur Laufzeit in vielen Fällen ändern können: der Markierungszustand (*Checked*), der Aktivierungszustand (*Enabled*) und der Sichtbarkeitszustand (*Visible*), vielleicht auch das Tastenkürzel (*ShortCut*).

Unter Verwendung von Aktionslisten brauchen Sie all diese Daten jeweils nur ein einziges Mal festzulegen, und – was noch besser ist – wenn Sie diese Daten zur Laufzeit ndern, wirkt sich diese Änderung sowohl auf den Schalter als auch auf den Menüpunkt aus.

TAction

Das zugrunde liegende Konzept besteht darin, dass die Aktion, die von beiden gleichermaßen ausgeführt werden soll, in einem neuen Objekt, einem Aktionsobjekt der Klasse *TAction* zusammengefasst wird.

TAction verfügt über alle in obiger Aufzählung genannten Properties und natürlich auch über ein *OnExecute*-Ereignis, das mit der Methode verknüpft wird, die die Aktion ausführt. Wenn Sie ein *TAction*-Objekt mit einem Menüpunkt oder einem Schalter verknüpfen, übernehmen Menüpunkt bzw. Schalter diese Daten einfach. Um die Verknüpfung herzustellen, brauchen Sie lediglich das *Action*-Property des Schalters bzw. des Menüpunkts auf das *TAction*-Objekt zu setzen.

Die Komponente, die zum Auslösen einer Aktion verwendet wird, also z. B. ein Menüpunkt, wird als *Client* der Aktion bezeichnet. Welche Properties von *TAction* in den Client der Aktion übernommen werden, bestimmen die *TActionLink*-Klassen der CLX, um die Sie sich normalerweise nicht zu kümmern brauchen. Es kann jedenfalls sein, dass ein Property von *TAction* in ein verschieden benanntes Property des Clients kopiert wird (so übernimmt *TToolButton.Down* beispielsweise den Wert von *TAction.Checked*).

ActionList

Die Komponentenklasse *TActionList* hat drei wichtige Aufgaben:

- Offensichtlich liegt der organisatorische Zweck darin, die häufig in großen Scharen auftretenden Aktionen von Programmen in einer gemeinsamen Liste zu verwalten. Einen Editor für diese Liste erhalten Sie mit einem Doppelklick auf eine *TActionList*-Komponente (Abbildung 4.8). Zur besseren Übersicht können Sie Aktionen auch in Kategorien aufteilen. Jedes *TAction*-Objekt verfügt hierzu über ein *Category*-Property, das der Aktionslisten-Editor verwendet, um die Aktionen in Kategorien zu sortieren.
- *TActionList* bietet auch den einzigen Weg, ein *TAction*-Objekt zur Entwurfszeit zu erzeugen, denn *TAction* selbst ist nicht in der Komponentenpalette vertreten. Im Aktionslisten-Editor können Sie mit Einfg bzw. mit dem entsprechenden Menüpunkt oder Schalter neue Aktionsobjekte erzeugen, deren Properties Sie dann über den Objektinspektor einstellen können.

- Schließlich speichert eine *TActionList* im Property *Images* einen Verweis auf die Bilderliste, auf die sich die *ImageIndex*-Angaben aller in ihr enthaltenen *TAction*-Objekte beziehen. Dies ist auch schon das einzige Property dieser Klasse – abgesehen von den obligatorischen *Name*- und *Tag*-Angaben.

Es ist übrigens nicht notwendig, die *TMainMenu*- oder *TPopupMenu*-Komponenten mit den *TActionList*-Objekten zu verknüpfen, sondern es genügt, wenn die einzelnen Menüpunkte mit *TAction*-Objekten verknüpft sind. Diese dürfen theoretisch auch aus verschiedenen Aktionslisten stammen.

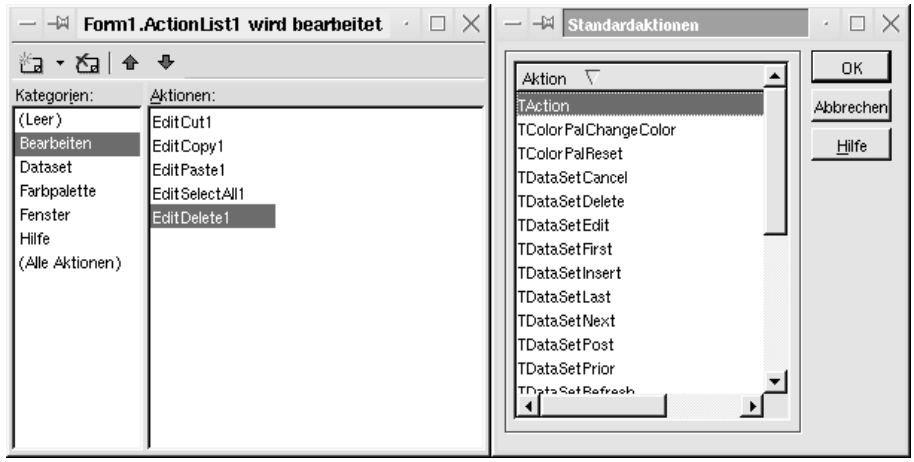


Abbildung 4.8: Editieren von Aktionslisten. Die Dialogbox zeigt die mit der CLX mitgelieferten Standardaktionen und die beiden in der Komponentenbibliothek dieses Buchs definierten Aktionen für die Farbpalette (Kategorie »Farbpalette«).

Praktischer Einsatz *R12*

In der Praxis sieht die Verwendung der Aktionslisten so aus, dass Sie für alle Befehle, die der Benutzer Ihrer Anwendung ausführen soll, eine eigene Aktion in einer *TActionList*-Komponente definieren. Bei umfangreichen Anwendungen bietet es sich an, diesen Aktionen verschiedene Kategorien (*Category*-Property) zuzuweisen. Statt die *OnClick*-Ereignisse von Schaltern und Menüpunkten zu bearbeiten, schreiben Sie eine Methode für das *OnExecute*-Ereignis der Aktion. Auch alle schon genannten weiteren Parameter wie etwa Tastenkürzel und Bildsymbol-Index legen Sie in der Aktion fest. Bei Schaltern und Menüpunkten müssen Sie dann als Einziges noch das *Action*-Property setzen.

Beispiele zur grundsätzliche Anwendung dieses Aktionskonzepts finden Sie sowohl im TreeDesigner aus Kapitel 6 (für die Befehle des Ansichtsmenüs) als auch in der von Kylix bereitgestellten Projektschablone für eine *MDI-Anwendung* (DATEI | NEU / PROJEKTE / MDI-ANWENDUNG).

Ereignisse von TAction R24

Die Klasse *TAction* verfügt noch über zwei interessante Ereignisse, die Sie im Objektinspektor mit Methoden verknüpfen können:

- *OnUpdate* wird während des Wartezustandes der Anwendung aufgerufen, um Ihnen die Gelegenheit zu geben, die Aktivierbarkeit der Aktion zu überprüfen und das *Enabled*-Property entsprechend zu setzen (ggf. auch das *Checked*-Property). Bei Aktionen, die sich auf ein Editierfeld beziehen, können Sie z. B. die Aktivierbarkeit der Aktion von der Frage abhängig machen, ob im Editierfeld Text selektiert ist:

  ```
  procedure TForm1.EditieraktionUpdate(Sender: TObject);
  begin
    Editieraktion.Enabled := Edit1.SelLength>0;
  end;
  ```

  Mit »Wartezustand« ist übrigens derselbe Zustand gemeint, in dem auch die *OnIdle*-Ereignisse auftreten und der in diesem Zusammenhang in Kapitel 4.8.1 untersucht wird.

- *OnHint* wird aufgerufen, bevor ein Hinweistext der Aktion angezeigt wird. Sie können darin bestimmen, ob und welcher Hinweis angezeigt werden soll (standardmäßig wird der im *Hint*-Property angegebene Hinweis angezeigt).

Standardaktionen

Das Konzept der Aktionen ist aber mit den bisher beschriebenen Möglichkeiten noch nicht ausgereizt, sondern es erlaubt auch die Definition von vordefinierten Aktionen. Das sind kurz gesagt Aktionen, zu denen Sie keine *OnExecute*-Methode mehr schreiben müssen, da sie »sich selbst ausführen« können. Ein praktisches Beispiel dafür gibt es in Kapitel 4.7.1, in dem für ein Memo-Feld ein funktionsfähiges Editiermenü mit allen üblichen Zwischenablageoperationen erzeugt wird, ohne eine Zeile Code zu schreiben.

Zur technischen Sicht der Standardaktionen kommen wir in Kapitel 8.6.7, wo erläutert wird, wie Sie eigene Standardaktionen definieren und sogar an andere Benutzer weitergeben können. An dieser Stelle soll es genügen, auf die mit Kylix mitgelieferten Standardaktionen hinzuweisen. Rufen Sie aus dem Kontextmenü des Aktionslisten-Editors den Befehl NEUE STANDARD-AKTION auf, um eine Liste der verfügbaren Aktionen zu erhalten und daraus zu wählen. Kylix wartet mit drei Kategorien von einfachen Standardaktionen auf: Fensteraktionen für MDI-Anwendungen (siehe Kapitel 6.7.4), die typischen Zwischenablageoperationen von Editierfeldern (diese werden immer auf das Editierfeld angewendet, das gerade den Tastaturfokus besitzt) sowie die von *TDBNavigator* gewohnten Navigationsbefehle für Datenbanken (in den Properties dieser Aktionen können Sie eine *TDataSource*-Komponente, auf die sich die Aktionen beziehen sollen, fest vorgeben).

Zielkomponenten

Bei den Standardaktionen ist auffällig, dass sie sich meistens auf eine andere Komponente beziehen: So benötigen beispielsweise Editieroperationen ein Editierfeld und Datenbankaktionen eine Datenquelle (*TDataSource*). Auch selbst geschriebene Aktionen sind oft von solchen *Zielkomponenten* abhängig, jedoch sind diese leicht ausfindig zu machen, da Sie sie einfach im Programmcode für das *OnExecute*-Ereignis mit Namen ansprechen können. Da die Aktionsobjekte der CLX aber in einem *beliebigen* Formular eingesetzt werden können, muss die CLX sich ihre Zielkomponenten auf eine kompliziertere Weise suchen. Der Ablauf dieser Suche ist wie folgt:

- Zuerst stellt die CLX fest, auf welches Formular sich die Aktion bezieht: falls vorhanden, wird das aktive Formular verwendet (angegeben im *ActiveForm*-Property des globalen *Screen*-Objekts), ansonsten das Hauptformular der Anwendung (*Application.MainForm*).
- Innerhalb dieses Formulars testet sie zuerst, ob die fokussierte Komponente ein passendes Ziel für die Aktion ist (*TForm.ActiveControl*).
- Ist das nicht der Fall, probiert sie, ob sich das Formular selbst als Zielkomponente eignet.
- Wenn auch dies nicht zutrifft, probiert sie alle anderen Komponenten des Formulars (*TForm.Components*) schrittweise durch, bis sich eventuell eine davon als passendes Ziel erweist.

Die Überprüfung, ob eine Komponente ein passendes Ziel für eine Aktion ist, wird vom Aktions-Objekt selbst vorgenommen. Die Standard-Editieraktionen akzeptieren beispielsweise nur fokussierte Editierfelder, so dass die Suche nach einer Zielkomponente hier entweder im zweiten Schritt oder gar nicht erfolgreich beendet werden kann. In Kapitel 8.6.7 wird gezeigt, wie dies in selbst definierten Standardaktionen in einer *HandlesTarget*-Methode geschieht.

Hinweis: Wie schon bei der Nachrichtenverarbeitung haben Sie auch beim Aufruf von Aktionen die Möglichkeit, auf globaler Ebene in die Verarbeitung einzugreifen. Noch bevor die eigentliche Aktion ausgeführt wird, erzeugt die CLX nämlich ein *OnExecute*-Ereignis der Aktionsliste und, soweit die zugehörige Bearbeitungsmethode es nicht unterbindet, danach noch ein *OnActionExecute*-Ereignis des Applikationsobjekts.

4.7 Die Zwischenablage

Die am weitesten verbreitete Methoden zum Datenaustausch zwischen X Window-Anwendungen funktioniert so, dass der Benutzer ein Datenelement (z. B. einen Textbereich) in einem Fenster markiert und es dann in einem anderen Fenster per Tastenkürzel, Menübefehl oder andere Eingaben einfügt. Ein AUSSCHNEIDEN-Befehl wie unter Windows ist normalerweise nicht erforderlich.

Die Daten werden dabei nicht direkt von der einen in die andere Anwendung kopiert, sondern vorher in einem Zwischenspeicher gelagert. Dieser Zwischenspeicher wird in diesem Kapitel als *Zwischenablage* oder *Clipboard* bezeichnet (in Übereinstimmung mit der Terminologie von Kylix und Qt, jedoch in Abweichung vom Sprachgebrauch in X11 selbst, wo das *Clipboard* nur ein Spezialfall von »Selections« ist).

Zu der erwähnten Vorgehensweise gibt es unter X Window auch noch Alternativen – typischerweise läuft also auch bei der Zwischenablage nicht alles so normiert ab wie etwa unter Windows. Die Kylix-IDE selbst ist ein Beispiel dafür, denn der Text, der in ihrem Editor markiert wird, steht nicht sofort anderen Anwendungen zum Einfügen zur Verfügung, sondern muss erst mit einer Ausschneiden- oder Kopieren-Operation in die Zwischenablage befördert werden. Dies mag jedoch daran liegen, dass die IDE in der ersten Version intern noch eine Windows-Kompatibilitäts-Schicht mit sich herumträgt, denn die CLX-Komponenten reagieren X-typisch und stellen markierten Text sofort in die Zwischenablage.

Aus technischer Sicht ist die Zwischenablage ein Speicherplatz, der vom X Server verwaltet wird. Auf Desktop-Installationen von Linux mit einem einzelnen PC ist der X Server natürlich derselbe PC, auf dem die Anwendungen laufen. In einem Netzwerk können jedoch Anwendungen von verschiedenen Rechnern auf dem Bildschirm eines gemeinsamen X Servers in Erscheinung treten und dieser X Server ist daher auch für den Datenaustausch verantwortlich. Zwischenablagedaten sind öffentlich zugänglich und können von jeder anderen Anwendung gelesen (und auch überschrieben) werden.

4.7.1 Vordefinierte Zwischenablage-Operationen

Die CLX definiert eine Klasse *TClipboard*, die den programmtechnischen Umgang mit der Zwischenablage stark vereinfacht. In vielen Fällen kommen Sie ohne diese Klasse aus, da einige CLX-Komponenten schon von selbst mit der Zwischenablage zusammenarbeiten, so etwa die Editierkomponenten *TEdit* und *TMemo*. Sie verfügen bereits über ein eingebautes Popup-Menü, über das Sie Text einfügen, kopieren und ausschneiden können.

Zwischenablage-Methoden und Aktionen R73

Sie können die genannten Operationen auch manuell über die Methoden *PasteFrom Clipboard*, *CopyToClipboard* und *CutToClipboard* starten; diese Methoden stehen Ihnen bei ein- und mehrzeiligen Editierfeldern aller Art zur Verfügung: *TEdit*, *TMemo*, *TMaskEdit*, *TDBEdit* und *TDBMemo*.

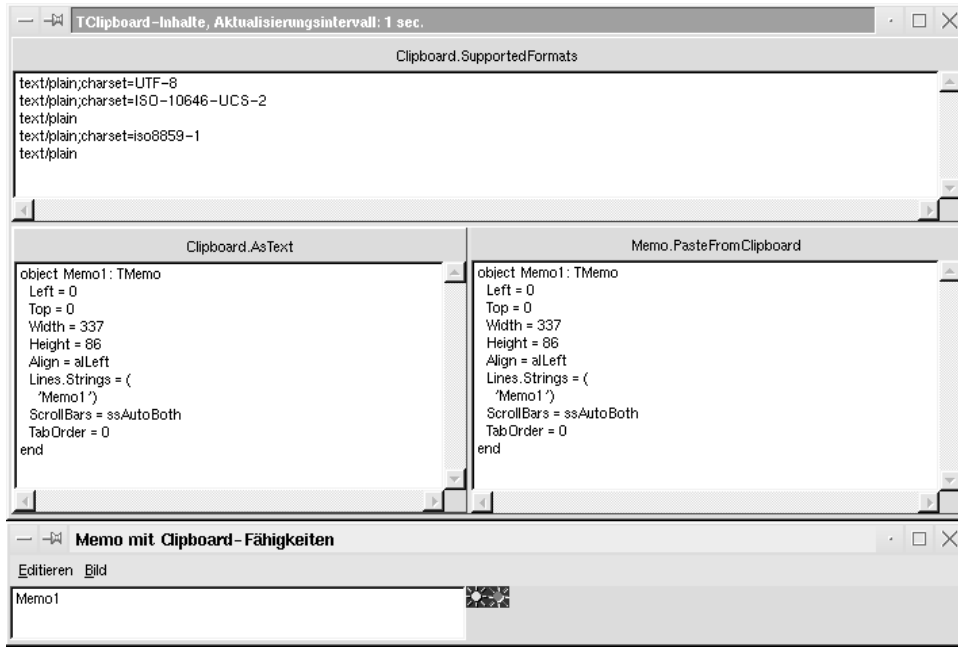


Abbildung 4.9: Das Beispielprogramm aktualisiert jede Sekunde seine Informationsfelder über den Zwischenablageinhalt und enthält in einem zweiten Fenster eine Memo- und eine Image-Komponente, mit denen sich Zwischenablageoperationen ausprobieren lassen.

Doch selbst wenn Sie die Zwischenablage-Operationen in Ihrer Anwendung über ein Menü zur Verfügung stellen wollen, haben Sie gute Chancen, um das Schreiben von Code herumzukommen, denn die CLX verfügt bereits über Standardaktionen, mit deren Hilfe Sie sich das Menü »zusammenklicken« können. Grundlage hierfür ist das in Kapitel 4.6.4 beschriebene Konzept von Aktionen und Aktionslisten. Im Folgenden soll ohne Erläuterung der Hintergründe nur kurz zusammengefasst werden, wie Sie ein Editiermenü mit Aktionen aufbauen:

- Fügen Sie eine Komponente der Klasse *TActionList* in Ihr Formular ein und rufen Sie mit einem Doppelklick darauf den Aktionslisten-Editor auf.

- Wählen Sie aus dessen Popup-Menü den Befehl NEU STANDARDAKTION und dann aus der erscheinenden Liste die fünf Standardaktionen mit dem Namensbeginn *TEdit...* . Schließen Sie den Dialog, um die Aktionen in die Befehlsliste einzufügen.
- Erstellen Sie ein neues Menü, etwa ein *TMainMenu* mit einem Hauptmenüpunkt BEARBEITEN.
- Für jeden der einzufügenden Zwischenablage-Menüpunkte müssen Sie jetzt im Objektinspektor nur noch die gewünschte Standardaktion für das *Action*-Property auswählen, alle weiteren Menüpunkt-Properties werden aus der Standardaktion übernommen. Auch die Bearbeitung des Menüpunkts geschieht automatisch, Sie brauchen also keine Ereignisse zu bearbeiten. Sogar der Aktivierungszustand der Punkte wird automatisch eingestellt, d.h. also, dass beispielsweise der KOPIEREN-Menüpunkt inaktiv dargestellt wird, wenn im aktuellen Editierfeld gar kein Text gewählt ist.

Auf der CD finden Sie ein Projekt, in dem diese Aktionen durchgeführt wurden, unter dem Namen *ClipboardViewer*. Dieses Projekt enthält zwei Formulare, im Formular *ClipboardTestForm* befindet sich das besagte Menü.

4.7.2 TClipboard

Mit reinem Textkopieren sind jedoch die Zwischenablage-Funktionen der CLX noch nicht erschöpft. Auch die Komponenten zur Anzeige von Bildern – *TImage* und *TDBImage* – können ihren Inhalt an die Zwischenablage weitergeben oder von dort beziehen. Allerdings bedarf es dazu schon vereinzelter Zeilen Programmcode und auch das erwähnte *TClipboard*-Objekt kommt jetzt ins Spiel.

TClipboard ist in der Unit *QClipbrd* definiert und ist keine Komponente, sondern direkt von *TPersistent* abgeleitet. Da Sie demnach keine Clipboard-Objekte zur Entwurfszeit in das Formular einfügen können, müssen Sie die Unit *QClipbrd* in jedem Fall selbst zur *uses*-Klausel Ihrer Unit hinzufügen.

Die wichtigsten Elemente von *TClipboard* sind:

- Zum Setzen des Zwischenablageinhalts das Property *AsText* und die Methoden *Assign* und *SetFormat*.
- Zum Auslesen des Inhalts dienen die Methode *GetFormat* und wieder das Property *AsText*; das Gegenstück zu *TClipboard.Assign* befindet sich außerhalb von *TClipboard* und ist die *Assign*-Methode von *TPicture*.
- Zur Bestimmung der verfügbaren Formate die Funktionen *Provides* und *SupportedFormats*.

Einfache Operationen

Clipbrd enthält nicht nur die Klasse, sondern auch ein globales Objekt davon, das wie zu erwarten *Clipboard* heißt und auf das Sie direkt zugreifen können. Um beispielsweise Text in die Zwischenablage zu kopieren, der sofort allen Anwendungen zugänglich ist, genügt eine einfache Zuweisung:

```
Clipboard.AsText := 'Hallo Zwischenablage!';
```

Ebenso können Sie *AsText* zum Auslesen des Textes verwenden.

Für die von *TClipboard* unterstützten Grafikformate gibt es keine Properties von der Art *AsPicture*, sondern die Methode *Assign*. Übergeben Sie dieser Methode ein *TBitmap*-, *TDrawing*- oder *TPicture*-Objekt, um dessen Inhalt in die Zwischenablage zu kopieren, z. B.:

```
Clipboard.Assign(Image1.Picture.Graphic); // Image1: TImage
```

Um umgekehrt ein Bild der Zwischenablage in eine Grafik zu übernehmen, rufen Sie die *Assign*-Methode dieser Grafik auf wie in den folgenden zwei Beispielen:

```
Image1.Picture.Graphic.Assign(Clipboard);
Bitmap.Assign(Clipboard);
```

Hierbei ist jedoch zu beachten, dass der Grafiktyp in der Zwischenablage mit dem Typ des Objekts, dem Sie die Grafik zuweisen, übereinstimmen muss. Dies gilt auch für die Grafik des Picture-Objekts. Angenommen, die Zwischenablage enthält eine Grafik des Typs *TDrawing*, die grundsätzlich in einem *TPicture* gespeichert werden kann. Wenn jedoch *TPicture.Graphic* bereits ein *TBitmap*-Objekt enthält, können Sie erst dann ein *Drawing* aus der Zwischenablage zuweisen, wenn Sie *Graphic* auf ein Objekt des passenden Typs umgestellt haben:

```
Image1.Picture.Graphic := TDrawing.Create;
Image1.Picture.Graphic.Assign(Clipboard);
```

Hierzu müssen Sie jedoch vor der Zuweisung wissen, welches Format überhaupt von der Zwischenablage zur Verfügung gestellt wird. Hiermit befasst sich der nächste Abschnitt.

Formate

Eine wichtige Eigenschaft der Zwischenablage ist, dass sie zur selben Zeit nur ein primäres Datenobjekt aufnehmen kann, welches jedoch in mehreren Formaten vorliegen kann. Wenn Sie beispielsweise Text aus dem mit Kylix mitgelieferten Hilfesystem markieren, wird dieser Text in mehreren Kodierungen in die Zwischenablage geschrieben. Andere Anwendungen, die vielleicht nur eine Kodierung kennen, haben dann eine höhere Chance, dass sie die Daten in einem ihnen genehmen Format laden können.

Wenn Sie an einem bestimmten Format interessiert sind, dann können Sie mit *Clipboard.Supports(Formatname)* abfragen, ob dieses gerade verfügbar ist. Eine Liste aller gegenwärtig abrufbaren Formate können Sie von der *SupportedFormats*-Methode erfragen. Diese erwartet eine fertig initialisierte Stringliste als Parameter und schreibt die Namen der Formate hinein. Das bereits erwähnte Programm *ClipboardViewer* etwa listet diese Formate in einem Memo auf, das im Sekundentakt aktualisiert wird. Zur Aktualisierung wird lediglich die folgende Anweisung ausgeführt:

```
Clipboard.SupportedFormats(Memo1.Lines);
```

Die in der ersten Kylix-Version unterstützten Formate haben die folgenden Namen:

- *text/plain* für ganz normalen Text aus Editierfeldern. Der Text wird neben diesem Format auch noch in weiteren Kodierungen bereitgestellt, die beispielsweise den Clipboard-Formatnamen »text/plain;charset=UTF-8« aufweisen.
- *image/delphi.bitmap* stellt eine vom *TBitmap*-Objekt der CLX in die Zwischenablage kopierte Bitmap dar.
- *image/delphi.drawing* steht für ein *TDrawing*-Objekt.

Der Name der beiden letztgenannten Formate deutet bereits an, dass es sich um spezielle Delphi-Formate handelt, die nicht mit anderen Bildformaten, etwa *image/ppm*, wie es von XPaint bereitgestellt wird, kompatibel sind. Sie können daher diese Formate kaum mehr als in andere Kylix-Anwendungen kopieren oder von dort übernehmen.

Wenn Sie das Vorhandensein eines dieser Formate abfragen wollen, brauchen Sie die Namen nicht wörtlich zu zitieren, sondern können die Konstanten *SDelphiBitmap* und *SDelphiDrawing* aus der Unit *QConsts* angeben, etwa:

```
if Clipboard.Provides(SDelphiBitmap) then begin
  Image.Picture.Graphic := TBitmap.Create;
  Image.Picture.Graphic.Assign(Clipboard);
end;
```

Hinweis: Die Formatnamen haben eine über die Zwischenablage weit hinausgehende allgemeine Bedeutung, und zwar handelt es sich um Mime-Typen, mit denen beispielsweise auch beim Verschicken von EMails oder beim Anzeigen von Dateiinhalten in der Vorschaufunktion des KDE-Konquerors gearbeitet wird.

ormatabfrage für Menüs *R25*

Wenn Sie die Zwischenablageoperationen in einem Menü angeben, kann es interessant sein, die derzeit in der Ablage befindlichen Formate zu überprüfen. So können Sie beim Öffnen des Menüs die Menüpunkte deaktivieren, die sich auf ein nicht vorhandenes Zwischenablageformat beziehen.

Hierzu genügt es nicht, die üblichen *OnClick*-Ereignisse zu bearbeiten, Sie benötigen ein Ereignis, das schon beim Öffnen des Menüs erzeugt wird. Beim Öffnen des Menüs kommt es entweder

- zu einem *OnClick*-Ereignis des (Haupt-)Menüpunkts, dessen Untermenü gerade geöffnet wird,
- oder zu einem *OnPopup*-Ereignis, falls es sich beim geöffneten Menü um eine einzelne *TPopupMenu*-Komponente handelt.

Bei diesem Ereignis können Sie das *Checked*-Property aller Menüpunkte neu setzen. Um das Format abzufragen, verwenden Sie im einfachsten Fall die *TClipboard*-Funktion *Supports*, der Sie das gefragte Format übergeben. Wenn etwa im TreeDesigner das BEARBEITEN-Menü aufgeklappt wird, stellt die folgende Methode fest, ob das spezielle TreeDesigner-Datenformat (siehe Kapitel 4.7.3) verfügbar ist, und deaktiviert den EINFÜGEN-Menüpunkt, falls das Format nicht gefunden wird:

```
procedure TForm1.BearbeitenMenuClick(Sender: TObject);
begin
  Einfgen1Click.Enabled := Clipboard.Supports('image/TreeDesigner');
end;
```

Hinweis: Falls Sie Aktionslisten verwenden, bringen Sie die Format-Abfrage stattdessen im *OnUpdate*-Ereignis der jeweiligen Aktion unter.

4.7.3 Eigene Ablageformate

Besonders nützlich kann sich die Zwischenablage auch erweisen, wenn Sie Daten in einem anwendungsspezifischen Format zwischen verschiedenen Fenstern der Anwendung kopieren wollen (diese Fenster können etwa von zwei verschiedenen, gleichzeitig laufenden Instanzen der Anwendung stammen oder zwei Dokumentfenster innerhalb einer MDI-Anwendung sein) oder wenn Sie Daten innerhalb eines Fensters duplizieren wollen.

Für VCL-Kenner gibt es die erfreuliche Nachricht zu vermelden, dass das Kopieren eigener Ablageformate mit der CLX noch erheblich einfacher ist als mit der VCL.

Kopieren anwendungsspezifischer Daten *R75*

Im TreeDesigner können Sie beispielsweise mit BEARBEITEN | KOPIEREN den markierten Teil des aktuellen Dokuments in die Zwischenablage kopieren (oder mit AUSSCHNEIDEN dahin verschieben) und mit BEARBEITEN | EINFÜGEN wieder einfügen – entweder in dasselbe Dokument, um ein Duplikat zu erhalten, oder in das Dokument eines anderen Dokumentfensters.

ür diese Funktionalität bietet es sich oft an, den Code zum Schreiben der Daten in einen Stream wiederzuverwenden, und zwar indem er auf einen Memory-Stream angewendet wird. Dieser kann dann leicht über die Zwischenablage kopiert werden.

Die folgende Methode reagiert im TreeDesigner auf das *OnClick*-Ereignis des KOPIEREN-Menüpunkts. Im TreeDesigner heißt die Methode, die alle Dokumentdaten in einen Stream schreibt, *TGraphicDocument.SaveToStream*. Ihr zweiter Parameter gibt an, ob nur die markierten Teile des Dokuments geschrieben werden sollen. Wenn der Parameter *False* ist, wird das gesamte Dokument geschrieben. Hier wird der Parameter immer dann auf *True* gesetzt, wenn Grafikelemente markiert sind. Es wird dann also nur die aktuelle Auswahl in die Zwischenablage kopiert:

```
procedure TDocumentForm.Kopieren2Click(Sender: TObject);
var
  Stream: TMemoryStream;
  MemHandle: THandle;
  MemBlock: pointer;
begin
  Stream := TMemoryStream.Create;
  // Wenn MarkedElementCount = 0 werden alle Elemente kopiert.
  Document.SaveToStream(Stream, Document.MarkedElementCount>0);
  Stream.Seek(0, soFromBeginning); // Stream zurück auf Startposition
  Clipboard.SetFormat('image/TreeDesigner', Stream);
  Stream.Free;
end;
```

Der entscheidende Aufruf in dieser Methode ist natürlich *Clipboard.SetFormat*. Der erste Parameter gibt einen Namen für das selbst definierte Format an, der zweite den Stream mit den Daten. Da *SetFormat* eine Kopie seines Inhalts anfertigt, kann der Stream schon am Ende der Methode wieder gelöscht werden, während die Kopie der Daten auf Abruf in der Zwischenablage bereitsteht.

Das Einfügen der Daten verläuft nahezu spiegelbildlich: Wieder wird ein neues *TMemoryStream*-Objekt erzeugt, dann wird statt *SetFormat* einfach *GetFormat* aufgerufen. Durch diese Methode wird der Memory-Stream mit Daten gefüllt. Beginnend vom Anfang des Streams werden nun mit der Methode *LoadFromStream*, die ursprünglich ür das Lesen von Dateien gedacht war, die Daten aus dem Stream gelesen.

```
procedure TDocumentForm.Einfgen1Click(Sender: TObject);
var
  Stream: TMemoryStream;
  MemHandle: THandle;
  MemBlock: pointer;
  Size : Integer;
begin
  if Clipboard.Provides('image/TreeDesigner') then begin
    Stream := TMemoryStream.Create;
    try
```

```
        Clipboard.GetFormat('image/TreeDesigner', Stream);
        Stream.Seek(0, soFromBeginning); // Zurück zum Anfang
        Document.LoadFromStream(Stream);
        Invalidate; // Fensterinhalt muss aktualisiert werden.
      finally
        Stream.Free;
      end;
    end;
  end;
```

Der Vollständigkeit halber sei nun auch noch die Methode für den Menüpunkt AUSSCHNEIDEN abgedruckt. Sie lässt sich auf die Kopieren-Methode und auf eine Löschen-Methode zurückführen, die jedoch nichts mit der Zwischenablage zu tun hat und daher hier nicht gezeigt wird:

```
procedure TDocumentForm.Ausschneiden1Click(Sender: TObject);
begin
  Kopieren2Click(Sender);
  Loeschen1Click(Sender);
end;
```

4.8 Threads

Threads sind Programmabschnitte, die innerhalb einer Anwendung parallel ablaufen können. Dieses Kapitel zeigt anhand eines interaktiven Beispielprogramms die Unterschiede zwischen zwei verschiedenen Multitasking-Typen, wichtige Eigenschaften der Threads und wie diese synchronisiert werden, so dass beispielsweise mehrere Threads auf dieselben Variablen zugreifen können. Weitere Beispielprogramme gibt es zur Synchronisation von Threads und zur Verwendung einer dynamischen Anzahl von Threads.

4.8.1 Multitasking-Typen

In einem Betriebssystem, das lediglich kooperatives Multitasking beherrscht, dürfen Anwendungen nicht ununterbrochen arbeiten, sondern müssen hin und wieder die Kontrolle an das Betriebssystem zurückgeben, so dass dieses weitere Eingaben des Benutzers an die anderen Anwendungen weiterleiten und dort bearbeiten lassen kann.

Beim preemptiven Multitasking kann das Betriebssystem die gerade laufende Anwendung jederzeit unterbrechen, was dem Anwendungsprogrammierer einen erheblichen Vorteil bringt, da er sich nicht mehr um die Aufteilung der Rechenzeit kümmern muss. Auch der Anwender kann großen Nutzen aus dieser Form des Multitasking ziehen, da seine Arbeitsoberfläche durch einzelne fehlgeleitete Programme nicht mehr ohne weiteres blockiert werden kann.

Ziel des ersten Beispielprogramms *ThreadDemo1* (Abbildung 4.10) ist ein Vergleich zwischen preemptivem Multitasking durch Threads und kooperativem Multitasking durch die Nutzung von *TApplication.OnIdle* oder *TApplication.ProcessMessages*.

ooperatives Multitasking

Wie oft eine Anwendung beim kooperativen Multitasking die Kontrolle an das Betriebssystem zurückgeben muss, damit der Benutzer vernünftig weiterarbeiten kann, können Sie z. B. beim Eingeben von Texten daran ablesen, wie schnell Sie erwarten, dass die Zeichen auf dem Bildschirm erscheinen, gemessen vom Zeitpunkt, an dem Sie die Taste gedrückt haben. Eine Viertelsekunde Reaktionszeit ist hier für einen Computer schon ziemlich träge, setzt aber voraus, dass eine eventuell im Hintergrund arbeitende Anwendung ihre Berechnung mindestens vier Mal pro Sekunde unterbricht.

Im einfachsten Fall wird das kooperative Multitasking dadurch sichergestellt, dass eine Anwendung jede Eingabe des Benutzers in Sekundenbruchteilen vollständig bearbeiten kann. Handelt es sich z. B. um eine Kylix-Anwendung, bei der die Bearbeitung innerhalb einer Ereignisbearbeitungsmethode stattfindet, gelangt die Kontrolle automatisch an das Betriebssystem zurück.

Wenn es einmal etwas länger dauert, verwandeln viele Anwendungen den Mauszeiger ür kurze Zeit in eine Sanduhr, die dem Benutzer signalisiert, dass er sich etwas gedulden muss. Hat die Anwendung die Aufgabe erledigt, erhält der Anwender seinen gewohnten Mauszeiger zurück und kann sich neue Herausforderungen für seine Software ausdenken. Das kooperative Multitasking ist in diesem Fall darauf beschränkt, dass das System so lange beliebig zwischen den Anwendungen wechseln kann, wie keine neuen Ereignisse zu bearbeiten sind.

Damit eine Anwendung auch bei kooperativem Multitasking länger dauernde Aufgaben erledigen kann, ohne das System zu blockieren, bieten sich mit der CLX zwei Möglichkeiten an:

- stückweises Ausführen der Aufgabe bei bestimmten regelmäßigen Ereignissen, z. B. bei *OnIdle*- oder *OnTimer*-Ereignissen oder
- komplettes Ausführen der Aufgabe mit freiwilliger Unterbrechung und Rückgabe der Kontrolle an das Betriebssystem durch Aufruf von *Application.ProcessMessages*.

Application.OnIdle

Eine Anwendung kann die »Zeit zwischen den Ereignissen« gezielt nutzen, um kleinere Arbeiten zu verrichten. Die CLX definiert für eine solche ereignislose Phase das Ereignis *TApplication.OnIdle*, das Sie nicht über den Objektinspektor, sondern nur im

Programmcode mit einer Methode verknüpfen können. Das Ereignis *OnIdle* wird immer dann von der CLX erzeugt, wenn diese feststellt, dass die Nachrichtenwarteschlange der Anwendung leer ist.

Da auch eine Methode für *OnIdle* nur wenig Zeit in Anspruch nehmen sollte (damit neue Eingaben des Benutzers sofort bearbeitet werden können), wird *OnIdle* meistens dazu verwendet, zeitlich unkritische Routineaufgaben zu erfüllen, wie z.B. die Anzeige einer Uhrzeit in der Statuszeile zu aktualisieren. Eine *Idle*-Methode kann jedoch auch länger dauernde Aufgaben durchführen, wenn sie bei jedem Durchlauf, also bei jedem *OnIdle*-Event, nur ein kleines Stück weiter rechnet und sich zwischen den Ereignissen den bisherigen Fortschritt merkt.

TTimer als Alternative

Eine Alternative zum Abfangen des *OnIdle*-Ereignisses ist in vielen Fällen die Verwendung einer *TTimer*-Komponente, die *OnTimer*-Ereignisse erzeugt. Diese *OnTimer*-Ereignisse sind erheblich regelmäßiger als *OnIdle*-Ereignisse. Ein Timer hat außerdem den Vorteil, dass er automatisch ausgelöst wird und die Anwendung nicht ständig abfragen muss, ob Nachrichten zur Bearbeitung anliegen (diese Abfrage wird von der CLX durchgeführt, bevor diese ein *OnIdle*-Ereignis erzeugt).

Zur Verwendung eines Timers nehmen Sie sich eine *TTimer*-Komponente von der Palettenseite *System*, stellen ihr *Intervall*-Property auf die Zeit in Millisekunden, die zwischen zwei *OnTimer*-Ereignissen vergehen soll, und schreiben eine Methode für das *OnTimer*-Ereignis. Um den Timer zeitweise zu deaktivieren, verwenden Sie das Property *Enabled*. Ein Beispiel für die Verwendung von *TTimer* geben die Beispielprogramme aus Kapitel 1.8 und das Programm *SyncDemo* aus Kapitel 4.8.3.

Das Multitasking-Demoprogramm

Wenn Sie das Beispielprogramm *ThreadDemo1* aus Abbildung 4.10 ablaufen lassen, erhalten Sie einen Eindruck davon, wie regelmäßig *OnIdle*-Ereignisse auftreten und wie im Vergleich dazu ein Thread mit der Berechnung vorankommen würde.

Das Fenster des Programms wird durch *TPanel*-Komponenten in vier Bereiche geteilt, die jeweils durch eine *TPaintBox*-Komponente abgedeckt werden. Der linke obere Bereich ist für das *OnIdle*-Ereignis reserviert: Die *OnIdle*-Methode zeichnet bei jedem Aufruf einen farbig gefüllten Kreis, wobei dieser – über mehrere Ereignisse betrachtet – auf einer Ellipsenbahn über die Fläche des Panels bewegt wird. Die Zahl in der Mitte dieser Ellipsenbahn gibt die für die letzte »Runde« ungefähr benötigte Zeit an.

Das Panel rechts oben wird auf eine später beschriebene Weise gefüllt, die beiden unteren Panels werden durch Threads gezeichnet.

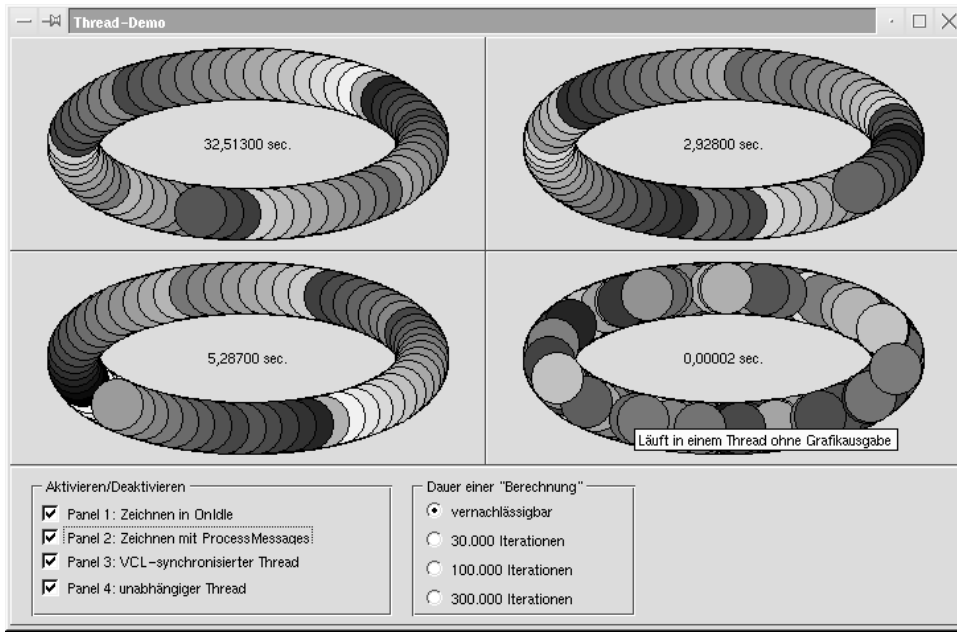


Abbildung 4.10: Die vier Bereiche dieses Fensters ermöglichen einen Vergleich zwischen kooperativem und eemptivem Multitasking

Unwichtige Arbeit wird in einer Hilfsunit versteckt

Das Beispielprogramm wurde so entworfen, dass der Code zum Thread-Management streng von allem restlichen Code getrennt ist. Der »restliche Code« ist dafür zuständig,

- eine Berechnung für die verschiedenen Threads (bzw. für *OnIdle*) zu simulieren,
- den Fortschritt der Berechnung durch die Ausgabe der Grafik am Bildschirm sichtbar zu machen
- und die Geschwindigkeit des Fortschritts zu messen sowie am Bildschirm auszugeben.

Damit wir uns hier ganz auf das Thread-Management konzentrieren können, wurde alles, was mit den obigen Aufgaben zusammenhängt, in die Unit *GlobalProcs* verlagert. Zum Verständnis des Programms brauchen Sie nur das Interface dieser Unit zu kennen:

```
unit GlobalProcs;
interface
uses SysUtils, QExtCtrls, QGraphics;
type
  TProgress = object
    // keine öffentlichen Variablen
```

```
  end;

procedure InitProgress(var Progress: TProgress);
procedure SimulateWork(var Progress: TProgress);
procedure ShowProgress(PaintBox: TPaintBox; const Progress: TProgress);

var
// zwei Optionen, die zur Laufzeit angepasst werden können
// (s. Abbildung 4.10, "Dauer einer Berechnung"
//  und "Synchronize verwenden"):
  Iterations: integer;
  UseSynchronize: Boolean=False;

implementation
```

Für jeden seiner vier Fensterbereiche wird das Beispielprogramm ein Objekt des oben deklarierten Typs *TProgress* erstellen, das vom jeweiligen Thread bzw. von *OnIdle* verwaltet wird. Um die Berechnung zu simulieren und die Grafik auszugeben, rufen *OnIdle* und die Threads einfach nur *SimulateWork* und *ShowProgress* auf und übergeben ihnen ihr *Progress*-Objekt, in dem alle intern wichtigen Daten enthalten sind.

Hinweis: Für *TProgress* wurde das Schlüsselwort *object* verwendet, damit *TProgress* von außen so leicht zu verwenden ist wie ein *record* (es sind keine Konstruktor- oder Destruktoraufrufe erforderlich), aber gleichzeitig den Zugriffsschutz einer *class* genießt (alle Elemente von *TProgress* sind als *private* deklariert und werden nur innerhalb der Unit *GlobalProcs* angesprochen).

Implementation von OnIdle *R166*

In der Methode für das *OnCreate*-Event des Formulars wird das *OnIdle*-Ereignis des globalen Anwendungsobjekts der CLX wie folgt mit einer Methode verknüpft:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Variablen initialisieren, die in FormIdle benutzt werden:
  InitProgress(IdleProgress);
  Application.OnIdle := FormIdle; // Ereignis und Methode verbinden
```

Um den bisherigen »Fortschritt« zu speichern, ist in der Formularklasse die Formularvariable *IdleProgress* deklariert. Sofern die *OnIdle*-Bearbeitung nicht über den Markierungsschalter *Panel1Active* abgestellt wurde, wird *IdleProgress* wie angekündigt einfach nur an die Prozeduren der Unit *GlobalProcs* weitergegeben:

```
procedure TForm1.FormIdle(Sender: TObject; var Done: Boolean);
begin
  // nur, wenn der Markierungsschalter nicht ausgeschaltet ist:
  if Panel1Active.Checked then begin
    GlobalProcs.SimulateWork(IdleProgress);
```

```
      GlobalProcs.ShowProgress(PaintBox1, IdleProgress);
      Done := False;
    end;
    // else Done := True; :: Done ist per Voreinstellung True.
  end;
```

OnIdle in der Praxis

Wenn Ihr Rechner durch andere Prozesse nicht ausgelastet ist, wird die *OnIdle*-Methode sehr schnell mit dem Zeichnen der Kreise vorankommen, und zwar zunächst einmal erheblich schneller als der im linken unteren Bereich arbeitende Thread. Allerdings kann es bei der Bewegung des Kreises sehr schnell zu Stockungen und Pausen kommen, nämlich immer dann, wenn es in der Nachrichtenschlange der Anwendung zu einem kleinen Stau kommt, die Bearbeitung einer einzelnen Nachricht etwas länger dauert oder das System aus anderem Grund für eine Weile vollständig ausgelastet ist.

Hinweis: In der ersten Version der CLX wird *OnIdle* nicht kontinuierlich ausgelöst, auch wenn Sie in der *Idle*-Methode *Done* auf *False* setzen. Der linke obere Kreis des Beispielprogramms bewegt sich daher immer nur einen Schritt weiter, wenn das Programm gerade Nachrichten zu bearbeiten hat und dann in den Zustand »Idle« übertritt, also beispielsweise nach jeder *OnMouseMove*-Nachricht, während Sie die Maus über dem Fenster bewegen. Der Grund für dieses von der VCL abweichende Verhalten liegt darin, dass die CLX die *OnIdle*-Nachrichten mit Hilfe eines Timers nur simuliert und diese Simulation offenbar noch nicht perfekt ist. Ob die *OnIdle*-Erzeugung in nachfolgenden Versionen der CLX an die VCL angepasst werden soll, ist nicht bekannt.

Application.ProcessMessages

Während die Methode für *OnIdle* davon abhängig ist, dass nicht zu viele andere Ereignisse anfallen (bzw. davon, dass überhaupt Ereignisse auftreten), verhält es sich bei der zweiten Möglichkeit des kooperativen Multitasking umgekehrt: Mit einem Aufruf von *Application.ProcessMessages* haben Sie es in der Hand, wann Ereignisse bearbeitet werden dürfen; wenn Sie dies zu selten zulassen, blockieren Sie Ihre eigene Anwendung – die anderen Anwendungen sind aufgrund des preemptiven Multitaskings von Linux hiervon unabhängig.

Ein Beispiel für einen sinnvollen Einsatz von *ProcessMessages* finden Sie im Beispielprogramm *FileDB* in Kapitel 7.3.7. In *ThreadDemo1* wird *ProcessMessages* innerhalb der Methode *Panel2Proc* aufgerufen. *Panel2Proc* tut dasselbe wie die eben gezeigte Methode *FormIdle*, kehrt aber nicht nach jedem einzelnen Schritt zum Aufrufer zurück, sondern ruft nach jedem Arbeitsschritt *ProcessMessages* auf:

```
procedure TForm1.Panel2Proc;
var
  Progress: Integer;
begin
  InitProgress(Progress);
  // Wiederholen, solange der Schalter Panel2Active eingeschaltet ist:
  while Panel2Active.Checked do begin
    Application.ProcessMessages;
    GlobalProcs.SimulateWork(Progress);
    GlobalProcs.ShowProgress(PaintBox2, Progress);
  end;
end;

// Aufruf von Panel2Proc beim Anklicken des Markierungsschalters:
procedure TForm1.Panel2ActiveClick(Sender: TObject);
begin
  if Panel2Active.Checked then
    Panel2Proc;
end;
```

Interessant ist hierbei, dass *Panel2Proc* erst dann beendet wird, wenn Sie den Markierungsschalter ausschalten und damit *Panel2Active.Checked* wieder auf *False* setzen. Die Ereignisbearbeitungsmethode *Panel2ActiveClick* kehrt also zunächst einmal überhaupt nicht zu ihrem Aufrufer zurück, und trotzdem können Sie dem Programm andere Eingaben senden, die es bereitwillig verarbeitet. Dies liegt am Aufruf von *Application.ProcessMessages*, zu dessen Laufzeit alle zwischenzeitlich angefallenen Nachrichten bearbeitet werden, also quasi geschachtelt in die Bearbeitung von *Panel2ActiveClick*.

Ein Aspekt ist hier jedoch wichtig: Das Fenster kann auf normalem Wege erst dann geschlossen werden, wenn alle Nachrichten abgearbeitet sind. Daher muss die Prozedur *Panel2Proc* gestoppt werden, wenn der Benutzer versucht, das Fenster zu schließen. Im Falle eines *OnClose*-Ereignisses wird daher folgende Aktion durchgeführt:

```
procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  Panel2Active.Checked := False; // Simuliert das Ausschalten
    // des Schalters, so dass eine eventuell aktive Panel2Proc
    // und somit auch eine eventuell noch nicht abgeschlossene
    // Bearbeitung des OnClick-Ereignisses für den Schalter
    // Panel2Active beendet wird.
end;
```

ProcessMessages in der Praxis

Die Grafik im Fensterviertel rechts oben wird kontinuierlich erneuert, solange nicht viele Nachrichten in die Warteschlange der Anwendung eingehen, denn *ProcessMessages* arbeitet so lange, bis keine Nachricht mehr anliegt. Wenn Sie nun beispielsweise die Maus über dem Formular bewegen, treten ständig neue *OnMouseMove*-Nachrich-

ten auf und *ProcessMessages* wird erst bei stillstehender Maus beendet – so lange muss dann die »Berechnung« im rechten oberen Fensterviertel pausieren. (Diese Situation trat zumindest auf einem 300 MHz schnellen Rechner des Autors auf. Es ist denkbar, dass bei unendlich hoher Rechnergeschwindigkeit die *OnMouseMove*-Nachrichten schneller bearbeitet werden können, als sie – abhängig von der Auflösung der Maus – von der Maus generiert werden können. In diesem Fall würde das *Panel 2* auch während einer Mausbewegung Rechenfortschritte vermelden.)

Eine interessante Beobachtung ist darüber hinaus, dass sich entweder der linke obere oder der rechte obere Kreis bewegt – je nach Situation in der Nachrichtenwarteschlange kommt also entweder ständig *OnIdle* zum Zug (solange kontinuierlich neue Nachrichten eintreffen) oder *Panel2Proc* (wenn einmal Pause im Nachrichtenbetrieb eintritt). Dieses Verhalten ist gewissermaßen paradox (Die »Untätigkeits-Nachrichten« *OnIdle* treten dann auf, wenn die Anwendung gerade nicht untätig ist) und entspricht auch nicht dem Verhalten des Programms mit der VCL unter Windows. Dennoch eignet sich die Verwendung von *ProcessMessages* auch unter Linux für die gleichen Zwecke wie unter Windows, denn unter der Annahme, dass der Benutzer die Maus die meiste Zeit *nicht* bewegt, wird sie häufig genug und auch regelmäßig zum Zug kommen, um eine Berechnung durchführen zu können.

Unter Linux sind weder *ProcessMessages* noch *OnIdle* notwendig, um andere Anwenungen zum Zuge kommen zu lassen, jedoch können Sie auf beide Arten erreichen, dass Ihre eigene Anwendung gleichzeitig rechnen und weitere Nachrichten bearbeiten kann, ohne dass Sie dazu einen neuen Thread starten müssen.

Preemptives Multitasking

Mussten wir uns bisher darum kümmern, eine längere Berechnung ständig zu unterbrechen, sei es durch *Application.ProcessMessages* oder durch ein möglichst schnelles erlassen der *OnIdle*-Bearbeitung, nimmt uns beim preemptiven Multitasking das Betriebssystem diese Aufgabe vollständig ab. Es kann das Programm zwar genau genommen nur bei einem Interrupt unterbrechen, aber schon der alte PC-Timer-Interupt wird ca. 18-mal pro Sekunde ausgelöst (theoretisch könnte das Betriebssystem eine Anwendung natürlich auch nach jeder einzelnen Anweisung unterbrechen, indem es wie ein Debugger Haltepunkte setzt).

ür Sie als Entwickler bedeutet das, dass Sie sich weitaus weniger Gedanken über die Beteiligung anderer Programme an der Rechenzeit machen müssen. Preemptives Multitasking bedeutet also nicht nur mehr Komfort für den Anwender, sondern auch für den Softwareentwickler. Die Tatsache, dass das Betriebssystem die gesamte Verantwortung über die Rechenzeit übernimmt, braucht jedoch keinen Programmierer in die Verantwortungslosigkeit zu treiben, denn auch Threads können kompliziert werden, wenn sie zu mehreren innerhalb eines Prozesses (einer Anwendung) auftreten, auf

gemeinsame Daten zugreifen und synchronisiert werden wollen. Hier stellt sich dann eventuell eine Vielzahl an neuen Herausforderungen, in die man die durch das preemptive Multitasking eingesparte Entwicklungszeit wieder neu investieren kann.

4.8.2 Threads in der CLX

Wenn Sie Ihre Anwendung in mehrere Threads aufteilen, kann sie gleichzeitig Benutzereingaben verarbeiten und längere Berechnungen durchführen. Nach dem Start der Anwendung läuft diese in einem *primären Thread* (oder Haupt-Thread), und nur dieser primäre Thread kann die Nachrichten des Anwenders bearbeiten. Um eine längere Berechnung durchführen zu können, ohne dass die Anwendung blockiert wird, kann der primäre Thread weitere Threads starten. Ein solcher zusätzlicher Thread genügt in vielen Fällen schon.

Die Klasse TThread

Nach der langen Vorrede des letzten Abschnitts kommen wir nun zur Sache und erstellen ein *TThread*-Objekt. Dafür machen wir zunächst vom Angebot der Kylix-IDE Gebrauch, eine Gerüst-Unit für einen neuen Thread anzulegen. Sie können Thread-Klassen natürlich auch selbst anlegen und müssen diese nicht jeweils in einer eigenen Unit unterbringen.

Wenn Sie also mit DATEI | NEU die Objektablage aufrufen, das mit »Thread-Objekt« beschriftete Symbol wählen und in der folgenden Dialogbox einen Namen für die neue Thread-Klasse eingeben, erhalten Sie eine neue Seite im Editor mit einer neuen Unit, die wiederum eine neue Klasse enthält. Für das Beispielprogramm *ThreadDemo1* ist der Klassenname *TDemoThread*; das Gerüst sieht wie folgt aus:

```
unit Unit2;
interface
uses
  Classes;

type
  TDemoThread = class(TThread)
  private
    { Private-Deklarationen }
  protected
    procedure Execute; override;
  end;

implementation

... Kommentar zu "SYNCHRONIZE" wurde hier gekürzt ...
```

```
{ TDemoThread }

procedure TDemoThread.Execute;
begin
  { Place thread code here }
end;

end.
```

Die Methode Execute

Im Implementationsteil der neuen Unit befindet sich ein hier nicht abgedruckter Hinweis zum Aufruf der CLX aus diesem Thread (hierzu kommen wir später) und der bis auf einen weiteren Hinweis leere Rumpf der Methode *Execute*.

Diese Methode ist auch bereits das Wichtigste, was ein Thread braucht; sie enthält sozusagen das »Hauptprogramm« des Threads: Der Thread beginnt mit der ersten Anweisung in *Execute* parallel zum bisher laufenden Programm zu arbeiten und wird beendet, sobald die Ausführung dieser Methode abgeschlossen ist.

Um nun wie im letzten Abschnitt die kreisende Scheibe zu zeichnen, kann die *Execute*-Methode wie folgt implementiert werden:

```
procedure TDemoThread.Execute;
begin
  while not Terminated do begin
    GlobalProcs.SimulateWork(Progress);
    Synchronize(Update);
  end;
end;
```

Hierbei fallen zwei Dinge auf: Viele Threads, die ein bestimmtes Ziel zu erreichen haben, enden »freiwillig«, wenn die *Execute*-Methode abgelaufen ist (z.B. die Threads aus Kapitel 4.8.4). Der oben gezeigte Thread kann theoretisch unendlich lange laufen, bis er von außen durch den Aufruf von *Terminate* beendet wird. In diesem Fall wird das *erminated*-Property gesetzt, das oben nach jedem Rechenschritt überprüft wird.

Außerdem wurde der bisher bekannte *ShowProgress*-Aufruf durch den Aufruf von *Synchronize(Update)* ersetzt, zu dem wir gleich kommen werden.

Hinweis: Tatsächlich besteht der durch *TThread* definierte Thread nicht aus der *Execute*-Methode, sondern aus der CLX-Funktion *ThreadProc*, die aber sofort *Execute* aufruft. Nachdem diese abgearbeitet ist, erzeugt *ThreadProc* das Ereignis *OnTerminate* des Threads und gibt das *TThread*-Objekt frei, falls Sie dessen Property *FreeOnTerminate* auf *True* gesetzt haben.

Synchronize

Der Aufruf von *Synchronize* ist erforderlich, da die bereits erwähnte Methode *ShowProgress* intern CLX-Objekte benutzt, um Grafik in der PaintBox auszugeben. Die CLX kann jedoch immer nur in einem Thread gleichzeitig aktiv sein: Wenn beispielsweise der Thread *TDemoThread* versuchen würde, in die PaintBox zu zeichnen, während gleichzeitig der Haupt-Thread diese PaintBox gerade selbst verwendet, könnte es zu unvorhersehbaren Zwischenfällen kommen.

Um derartige Konflikte einfach vermeiden zu können, definiert *TThread* die Methode *Synchronize*. Diese wartet ab, bis sich die CLX eventueller anderer Aufgaben entledigt hat, und führt erst dann die Methode aus, die Sie ihr als Parameter übergeben. Sie brauchen nun, um die CLX aus einem Thread heraus aufzurufen, nur alle zusammengehörenden CLX-Aufrufe in eine parameterlose Methode zu packen und diese an *Synchronize* weiterzugeben. Die oben an *Synchronize* übergebene Methode *Update* enthält nur den schon bekannten *ShowProgress*-Aufruf:

```
procedure TDemoThread.Update;
begin
  GlobalProcs.ShowProgress(PaintBox, Progress, Color);
end;
```

Der TThread-Konstruktor

Es wird nun Zeit, den restlichen Quelltext für *TDemoThread* nachzuliefern; es handelt sich um die Erweiterung des automatisch erstellten Klassengerüsts, das die Variablen enthält, die in den oben abgedruckten Methoden bereits verwendet wurden, sowie um den Konstruktor *Create*:

```
type
  TDemoThread = class(TThread)
  protected
    PaintBox: TPaintBox;
    Progress: Integer;

    procedure Execute; override;
    procedure Update;
  public
    constructor Create(PB: TPaintBox);
  end;

constructor TDemoThread.Create(PB: TPaintBox);
begin
  inherited Create(True);
  PaintBox := PB;
  InitProgress(Progress);
  Resume;
end;
```

Create muss natürlich als *public* deklariert sein, damit er von außen aufgerufen werden kann. Seine Aufgabe ist es in diesem Fall, eine *TPaintBox*-Komponente entgegenzunehmen, in die der Thread später seine Grafik zeichnen soll. *Create* hebt diese PaintBox-Referenz in der Variablen *PaintBox* auf, die von der oben gezeigten Methode *Update* verwendet wird.

Der von *TThread* vordefinierte Konstruktor erwartet einen booleschen Parameter, der besagt, ob der Thread zunächst einmal angehalten werden soll (im Falle von *True*). *TDemoThread.Create* ruft diesen geerbten Konstruktor immer mit dem Parameter *True* auf, was bedeutet, dass der Thread nicht sofort gestartet wird, sondern erst, wenn die Methode *Resume* aufgerufen wird. *Resume* sollte in diesem Fall erst nach abgeschlossener Initialisierung aufgerufen werden.

Starten eines Threads

Sehen wir uns nun an, wie man ein *TThread*-Objekt vom Haupt-Thread aus steuern kann. Als Erstes muss das *TThread*-Objekt wie jedes andere Objekt, das nicht schon im Formularentwurf eingebaut wurde oder anderweitig vordefiniert ist, über den Konstruktoraufruf konstruiert werden. Das Beispielprogramm tut dies in der Methode für das *OnCreate*-Ereignis:

```
// Auszug aus der Klassendeklaration von TThreadDemo1Form:
  public
    Thread1: TDemoThread;

procedure TThreadDemo1Form.FormCreate(Sender: TObject);
begin
  ... // Initialisierung für OnIdle, siehe Kapitel 4.8.1
  // der Thread soll in der PaintBox3 (links unten) zeichnen:
  Thread1 := TDemoThread.Create(PaintBox3);
```

Da *TDemoThread* den *Create*-Konstruktor von *TThread* überschreibt, entfällt hier der boolesche Parameter, der normalerweise angibt, ob der *Thread* nicht sofort gestartet werden soll.

Das Laufzeitverhalten im Vergleich zu OnIdle/ProcessMessages

Wie schon im Abschnitt 4.8.1 angedeutet, ist das Verhalten des Threads im Beispielprogramm gegenüber den beiden kooperativen »Konkurrenten« durch eine höhere Gleichmäßigkeit charakterisiert. Wenn es bei der Mausbewegung über dem Formular otzdem noch zu Stockungen im Thread kommt, so liegt dies an dem Aufruf von *Synchronize*, durch den der Thread häufig darauf warten muss, die Erlaubnis zur Ausgabe der Grafik zu erhalten. Auch die Geschwindigkeit des Threads kann aus diesem Grund nicht die Geschwindigkeit erreichen, die *OnIdle* hat, während die Maus über dem Fenster bewegt wird und zahlreiche *OnIdle*-Ereignisse auftreten.

Der Thread im rechten unteren Fensterviertel beweist dies, indem er die gleichen »Berechnungen« durchführt, aber nicht auf die Ausgabe von Grafik wartet: Seine Geschwindigkeit ist um ein Vielfaches höher, solange Sie nicht die »Dauer einer Berechnung« erhöhen und damit für den Thread links unten das Verhältnis zwischen Rechenaufwand und Grafikausgabe stark verändern.

Manipulation eines Threads

Sobald der Thread mit *Create* erzeugt ist, können Sie ihn auf verschiedene Weise manipulieren. Wie üblich stellt die Klasse *TThread* zu diesem Zweck Methoden, Properties und Events, in diesem Fall allerdings nur ein einziges, zur Verfügung. Diese Klassenelemente sind in der folgenden Tabelle zunächst einmal zusammengefasst:

TThread-Element	**Aufgabe**
constructor Create (CreateSuspended: Boolean);	erzeugt das Objekt; falls *CreateSuspended True* ist, wird der Thread nicht sofort gestartet – dies kann z. B. mit *Resume* nachgeholt werden.
function WaitFor: Integer;	wartet, bis der Thread beendet ist, und liefert den Rückgabewert des Threads als Ergebnis zurück. Das Warten des aktuellen Threads besteht darin, dass seine Ausführung angehalten wird.
procedure Resume;	nimmt die Ausführung des Threads wieder auf, falls der Thread noch nicht gestartet oder mit *Suspend* oder *Suspended:=True* angehalten wurde. Der Thread wird allerdings erst dann fortgesetzt, wenn *Resume* genauso oft aufgerufen wurde wie *Suspend*.
procedure Suspend;	hält die Ausführung des Threads an und erhöht einen internen Zähler um eins, so dass mehrere Aufrufe von *Suspend* mehrere Aufrufe von *Resume* erfordern, bis der Thread fortgesetzt wird.
procedure Terminate;	setzt das interne *TThread*-Property *Terminated*. Damit der Thread tatsächlich beendet wird, müssen Sie dieses Flag in der *Execute*-Methode abfragen und die Methode verlassen, wenn es den Wert *True* erhält.
property FreeOnTerminate: Boolean;	Wenn Sie dieses (per Voreinstellung auf *False* gesetzte) Flag auf *True* setzen, gibt die CLX das Thread-Objekt automatisch mit einem Aufruf von *Free* frei, sobald der Thread beendet ist. (Vorsicht: Eventuell auf das Thread-Objekt weisende Zeiger werden natürlich nicht auf *nil* gesetzt.)
property Suspended: Boolean;	Setzen des Wertes *True* bewirkt den Aufruf von *Suspend*, falls der Thread noch nicht angehalten wurde. Umgekehrt führt ein Setzen des Wertes *False* zum Aufruf der Methode *Resume*, falls der Thread nicht bereits läuft.
property ThreadID: THandle;	Kennzahl des Threads auf Betriebssystem-Ebene.

TThread-Element	Aufgabe
operty OnTerminate: TNotifyEvent;	Dies ist das einzige Event dieser Klasse; es wird ausgelöst, sobald die Methode *Execute* beendet wurde und kurz bevor das Objekt gelöscht wird (falls *FreeOnTerminate=True*). Die Bearbeitungsmethode für dieses Event hat das einfache Format *»xxxTerminate(Sender: TObject);«*.

Unterschiede zwischen Threads in CLX und VCL

Aufgrund von Unterschieden zwischen Linux- und Windows-Threads steht nicht die gesamte Funktionalität der VCL-Threads auch unter Linux zur Verfügung. Dies betrifft hauptsächlich die Verwendung von Prioritäten und das Anhalten von Threads:

- Prioritäten sind zwar in der Thread-Schnittstelle von Linux als Parameter vorgesehen, jedoch noch nicht implementiert. Der Versuch, die Standardpriorität zu ändern, wird daher mit einem Fehlerergebnis des Linux-Kernels und daraufhin mit einer Exception der CLX quittiert. Im weiteren Verlauf der Linux-Entwicklung ist es jedoch möglich, dass Sie dann auch die *TThread*-Properties *Priority* und *Policy* setzen können. Im Gegensatz zur Priorität von Windows-Threads wird jedoch die Priorität eines Linux-Threads einfach als Integer-Zahl angegeben und nicht als Konstante vom Typ *TThreadPriority*.
- Das Anhalten von Threads (Methode *Suspend* oder Property *Suspended*) ist dagegen vom Linux-Thread-System überhaupt nicht vorgesehen. Es wird von der CLX simuliert, jedoch unter Nutzung einer »Lücke« in der Linux-Implementierung, in der diese vom POSIX-Standard abweicht. Auch dies kann sich in Zukunft ändern, mit im Gegensatz zu *Priority/Policy* umgekehrten Auswirkungen auf eine Kylix-Anwendung: Die Verwendung von *Suspend/Resume* funktioniert dann *nicht* mehr. (Lediglich der erste Aufruf von *Resume* zum Starten eines Threads, der nicht schon im Konstruktoraufruf gestartet wurde, sollte dann noch möglich sein.)

Aufgrund dieser Umstände verzichtet das Beispielprogramm ganz auf die Verwendung von Prioritäten und die *Suspend*-Methode. Trotzdem bietet es eine Möglichkeit, Threads anzuhalten, und zwar mit einer zukunftssicheren Lösung, die auch Änderungen in der Linux-Implementierung überdauern sollte. Weil sie von Synchronisationsobjekten Gebrauch macht, wird sie erst am Ende von Kapitel 4.8.3 beschrieben.

4.8.3 Mehrere Threads und deren Synchronisation

Wenn eine Anwendung aus mehreren Threads besteht, so müssen sich diese viele Ressourcen der Anwendung teilen, wodurch Konflikte entstehen können. Dieses Kapitel beschreibt ein zweites Beispielprogramm, das die Häufigkeit dieser auf den ersten Blick selten auftretenden Konflikte demonstriert und zeigt, wie derartige Konflikte

durch die Verwendung von Synchronisationsobjekten vermieden werden können. Wir beginnen jedoch mit einem einfachen Beispiel für ein Programm mit zwei zusätzlichen Threads, bei dem noch keine weitere Synchronisation erforderlich ist.

Verbesserung von ThreadDemo1 durch einen zweiten Thread *R102*

Und zwar geht es ein letztes Mal um das Beispielprogramm des letzten Kapitels, dessen Fenster noch ein leer stehendes Viertel zu beklagen hat. Im letzten Kapitels haben wir festgestellt, dass der Thread durch das ständige Warten auf die Synchronisation der Grafikausgabe stark gebremst wurde und unter Umständen sogar noch erheblich langsamer als die *OnIdle*-Methode war. Es gibt mehrere Möglichkeiten, diese Bremse zu lösen:

- Sie könnten im Thread gänzlich darauf verzichten, die Bildschirmanzeige unter Verwendung der CLX zu aktualisieren. Der Thread müsste dann kein *Synchronize* aufrufen und könnte immer so schnell arbeiten, wie es der Prozessor und die anderen Threads erlauben. Um dennoch eine Rückmeldung zu erhalten, wie schnell der Thread vorankommt, könnte der Haupt-Thread in gewissen Abständen – z. B. im *OnIdle*- oder in einem Timer-Event – einen Fortschrittszähler (in diesem Fall die Variable *TDemoThread.Progress*) abfragen und den Fortschritt in einer der Komponenten oder Zeichenflächen anzeigen.
- Das Beispielprogramm geht einen ähnlichen Weg: Es lagert die Aktualisierung des Bildschirms für das rechte untere Fensterviertel in einen weiteren Hilfs-Thread aus. Dieser muss dann zwar mit *Synchronize* auf die CLX warten, hält damit aber den zweiten Thread nicht auf. Während der Hilfs-Thread also noch darauf wartet, den letzten Fortschritt am Bildschirm ausgeben zu dürfen, kann der eigentliche Rechen-Thread in der Zwischenzeit weiterrechnen.

Die Unit *Thread2* enthält die beiden genannten Threads, wobei der Rechen-Thread *TDemoThread2* und der Hilfs-Thread *TUpdateThread* heißt. Gegenüber dem im letzten Kapitel gezeigten Thread *TDemoThread* wird die *while*-Anweisung der Methode *Execute* des neuen Threads um den bremsenden Aufruf von *Synchronize(Update)* verkürzt, so dass nur noch übrig bleibt:

```
procedure TDemoThread2.Execute;
begin
  while not Terminated do
    GlobalProcs.SimulateWork(Progress);
end;
```

Ein weiterer Thread wiederholt nun ununterbrochen den *Synchronize*-Aufruf, der aus dem ersten Thread entfernt wurde; die Variable *ThreadToUpdate* wurde zuvor auf diesen ersten Thread gesetzt:

```
procedure TUpdateThread.Execute;
begin
  while not Terminated do
    Synchronize(Update);
end;

procedure TUpdateThread.Update;
begin
  GlobalProcs.ShowProgress(ThreadToUpdate.PaintBox,
                           ThreadToUpdate.Progress);
end;
```

Was bleibt, ist das Starten beider Threads in der *OnCreate*-Methode des Formulars. Der *Thread2* erhält nun die vierte PaintBox als Arbeitsgebiet zugewiesen (allerdings wird diese PaintBox nur im Thread *UpdateThread* benutzt):

```
procedure TThreadDemo1Form.OnCreate(Sender: TObject);
begin
  ...
  Thread2 := TDemoThread2.Create(PaintBox4);
  UpdateThread := TUpdateThread.Create(Thread2);
```

Ein Programmstart zeigt, dass *Thread2* bei gleicher Priorität tatsächlich erheblich schneller vorankommt als der erste Thread und auch die beiden kooperativen Methoden weit übertrifft (dies liegt zu einem guten Teil natürlich wieder daran, dass die eigentliche Berechnung kaum ins Gewicht fällt, wodurch die kooperativen Methoden natürlich erheblich benachteiligt werden). Allerdings ist das Gespann aus den beiden neuen Threads weniger gut für optische Effekte geeignet, da das Aktualisieren der Bildschirmanzeige in *UpdateThread* zu selten stattfindet. *TDemoThread2* macht zwischen zwei Zeichenvorgängen so große Fortschritte, dass keine zusammenhängende Ellipse mehr gezeichnet wird, sondern die einzelnen Kreise vielmehr an zufällig wirkenden Positionen erscheinen. (Auch hier lohnt es sich wieder, den Parameter »Dauer der Berechnung« zu erhöhen.)

TThread.WaitFor

Bevor wir zu einem Beispiel für den gleichzeitigen Datenzugriff in mehreren Threads kommen, sei hier auf die einfachste Art der Thread-Synchronisation hingewiesen, für die nur eine Methode aufgerufen werden braucht: Mit *TThread.WaitFor* synchronisieren Sie den Thread, aus dem Sie *WaitFor* aufrufen, auf das Ende des Threads, dessen *WaitFor*-Methode aufgerufen wird. *WaitFor* besteht nicht etwa aus einer Warteschleife, sondern bewirkt, dass der aufrufende Thread so lange angehalten wird, bis der andere Thread beendet ist.

WaitFor ist damit eine besonders einfache Möglichkeit sicherzustellen, dass nur einer von zwei Threads gleichzeitig aktiv ist, allerdings bewirkt sie, dass die Threads nacheinander ablaufen, was eigentlich nicht das Ziel mehrerer Threads sein sollte.

Gleichzeitige Datenzugriffe

Manchmal müssen mehrere Threads auf gemeinsame Daten zugreifen. Da die Variablen der Anwendung allen Threads innerhalb der Anwendung zugänglich sind, ist dies technisch kein Problem. Für die Funktion der Anwendung bedeutet ein gleichzeitiger mehrfacher Zugriff jedoch eine Gefahr:

- Wenn mehrere Threads auf die gleichen Daten schreibend zugreifen, kann es sehr schnell zur Zerstörung der Daten kommen. So kann es zum Beispiel kaum funktionieren, dass zwei Threads die gleiche Liste gleichzeitig in aufsteigender und in absteigender Reihenfolge sortieren.
- Auch wenn nur einer von mehreren Threads die Daten verändert, kann es die anderen Threads ins Chaos stürzen, wenn sie die Daten während einer solchen Veränderung lesen und die Veränderung gerade in einem Zwischenstadium befindlich ist, in dem die Daten keinen oder einen falschen Sinn ergeben. (Ein solches Zwischenstadium ist nur bei Variablen denkbar, die vom Prozessor nicht in einem Schritt gesetzt werden. Es kommt natürlich auch darauf an, dass der Compiler die entsprechenden Anweisungen erzeugt und ein *Word* nicht beispielsweise als zwei Einzelbytes beschreibt.)

Wie man dieses Problem auf einfache Weise vermeiden kann, demonstriert die CLX: Man verbietet einfach, dass mehrere Threads, die auf die gleichen Daten zugreifen wollen, gleichzeitig laufen. Nichts anderes ist die Aufforderung zur Verwendung der *Synchronize*-Methode, die sicherstellt, dass die CLX immer nur einmal gleichzeitig aktiv ist und folglich auch immer nur einmal auf ihre Daten zugreifen kann.

Hinweis: Wenn eine Funktion oder eine ganze Bibliothek mit sich selbst in Konflikt gerät, falls sie in mehreren Threads gleichzeitig aktiv ist, bezeichnet man diese Funktion bzw. Bibliothek als *nicht wiedereintrittsfähig* (non reentrant). Die Wiedereintrittsfähigkeit einer Funktion lässt sich beispielsweise dadurch erreichen, dass die Funktion nur lokale Variablen verwendet und nur Funktionen aufruft, die ihrerseits wiedereintrittsfähig sind.

Ein Beispielprogramm mit gleichzeitigem Datenzugriff

Das Beispielprogramm *SyncDemo* aus Abbildung 4.11 startet zwei Paare von Threads, wobei die Threads innerhalb eines Paares auf dieselben Daten zugreifen müssen und nur das zweite Paar richtig synchronisiert ist. Im Memofeld des Fensters wird angezeigt, wie häufig es in den beiden Thread-Paaren zu Zugriffskonflikten kommt.

Sehen wir uns zunächst die Aufgaben der Threads an:

- Jeder Thread soll in ständiger Wiederholung das gesamte Array *Bytes* aus der unten gezeigten Datenstruktur bearbeiten.

- Dabei füllt es die einzelnen Array-Elemente mit Zahlen und setzt danach alle Elemente des Arrays wieder auf Null.
- Nur wenn alle Array-Elemente auf Null gesetzt sind, darf ein anderer Thread auf das Array zugreifen. Diese frei erfundene Regel soll für das Beispielprogramm alle Zustände des Arrays, in denen nicht alle Elemente Null sind, als »Zwischenzustände« definieren, während derer nur ein Thread Zugriff auf das Array haben darf.

Das Daten-Array ist im falsch synchronisierten Thread wie folgt definiert:

```
const
  DataCount = 100;
type
  TThreadData = record
    InUseFlag: Boolean;
    Bytes: array[1..DataCount] of Byte;
  end;
```

Das *InUseFlag* soll speichern, ob das Array gerade von einem der beiden Threads verwendet wird. Bevor ein Thread das Array *Bytes* verändert, soll er warten, bis *InUseFlag* auf *False* gesetzt wird, bis also der andere Thread seine Bearbeitung des Arrays beendet hat. Bevor er das Array selbst verändert, setzt er seinerseits *InUseFlag* auf *False*, damit der zweite Thread warten muss.

Der eigentliche Fehler in der Synchronisation besteht nun nicht darin, dass beide Threads gleichzeitig auf *Bytes* zugreifen, sondern dass sie beide gleichzeitig das *InUseFlag* modifizieren (als Folge davon greifen sie dann auch gleichzeitig auf das Array zu):

```
procedure TSyncThread.Falsch; // einer von vielen falschen Wegen
var
  CollisionDetected: Boolean;
  i: Integer;
begin
  // Data weist hier auf eine von beiden Threads benutzte
  // TThreadData-Struktur
  repeat
    // Warten, bis der Datenblock nicht mehr verändert wird:
    while Data^.InUseFlag do ;
    // Datenblock reservieren:
    Data^.InUseFlag := True;
    inc(IterationCount); // nur für die Statistik im Fenster
    // Überprüfen, ob sich der Datenblock tatsächlich im
    // Ursprungszustand (gefüllt mit 0-Bytes) befindet:
    CollisionDetected := False;
    for i := 1 to DataCount do
      if Data^.Bytes[i] <> 0 then
        CollisionDetected := True;
    if CollisionDetected then // Kollision
      inc(CollisionCount); // für die Statistik im Fenster
    // Die eigentliche Operation, die immer nur ein
```

```
      // Thread gleichzeitig ausführen soll, und zwar
      // den Datenblock zu verändern, folgt hier:
      for i := 1 to DataCount do
        Data^.Bytes[i] := i;
      // Datenblock in den definierten Ausgangszustand
      // zurückversetzen:
      for i := 1 to DataCount do
        Data^.Bytes[i] := 0;
      // Datenblock freigeben:
      Data^.InUseFlag := False;
    until False;
  end;
```

Bevor diese Methode den eigentlichen Datenzugriff durchführt, überprüft sie, ob die Synchronisation über das *InUseFlag* funktioniert hat. Dazu testet sie, ob alle Elemente des Arrays Null sind (dieser Test findet zwar nicht alle Fälle, in denen beide Threads gleichzeitig auf das Array zugreifen, aber doch so viele, dass es für eine Demonstration ausreicht). Falls die Methode feststellt, dass das Array nicht vorschriftsmäßig mit Nullen gefüllt ist, erhöht sie den Zähler *CollisionCount*.

Hinweis: Der Zähler *CollisionCount* wird durch Methoden, die hier nicht abgedruckt sind, im Programmfenster angezeigt, siehe Abbildung 4.11. Die Aktualisierung des Fensters findet übrigens zur Abwechslung einmal mit Hilfe einer *Timer*-Komponente statt.

Bei Ablauf des Programms zeigt sich, dass diese Art der Synchronisation tatsächlich nicht immer funktioniert: In seltenen Fällen stellen beide Threads fest, dass sie auf ein Array zugreifen, das nicht vollständig mit Nullen gefüllt ist, und erhöhen daher den Zähler *CollisionCount*. Zwar tritt dieser Fall weniger als einmal alle 10000 Durchläufe auf; da moderne Prozessoren jedoch mit diesen Durchläufen sehr schnell sind, treten die Fehler immer noch schneller auf, als der Anwender bei einer echten Anwendung Fehlermeldungsfenster schließen könnte.

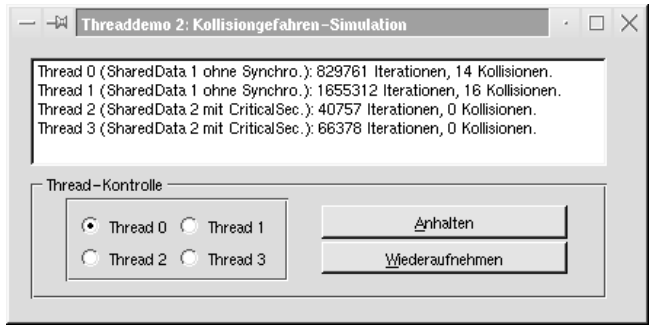


Abbildung 4.11: Zugriffskonflikte im Beispielprogramm SyncDemo

Die Kollisionsursache

Die Ursache des Fehlers liegt darin, dass das Betriebssystem die beiden Threads auch zwischen den beiden folgenden Zeilen unterbrechen kann:

```
while Data^.InUseFlag do ;
Data^.InUseFlag := True;
```

Angenommen, Thread 1 stellt fest, dass *InUseFlag* = *False*. Er beendet die *while*-Schleife und will das Flag nun setzen, wird aber vorher vom System unterbrochen, das nun zum Thread 2 wechselt. Auch dieser führt die obige *while*-Anweisung aus und stellt fest, dass das Flag gelöscht ist (da Thread 1 es noch nicht setzen konnte). Nun setzt Thread 2 das Flag, und wenn das Betriebssystem wieder zu Thread 1 wechselt, tut dieser dasselbe. Beide Threads führen nun den Code aus, der eigentlich nur einmal gleichzeitig aktiv sein sollte.

Es genügt also nicht, dass der Prozessor das *InUseFlag* in jeweils einem Schritt setzen oder lesen kann, er müsste sogar in der Lage sein, Lese- und Schreibzugriff zusammen in einem nicht unterbrechbaren Schritt auszuführen, damit es nicht zu einem Konflikt kommen könnte.

Interlocked-Funktionen

Die *SysUtils*-Unit bietet Ihnen vier verschiedene Funktionen mit dem Namensanfang Interlocked« an, die zwei dieser oder ähnlicher Elementaroperationen zu einer nicht unterbrechbaren Einheit zusammenfassen. Der obige Programmauszug könnte beispielsweise durch die kollisionssichere Schleife *while InterlockedExchange (Data^.InUseFlag, 1)=1 do;* ersetzt werden (*InUseFlag* müsste dazu als *Integer* deklariert und zur Freigabe auf 0 statt auf *False* gesetzt werden).

Es gibt jedoch wesentlich elegantere Möglichkeiten, bei der der wartende Prozess die Prozessorzeit nicht in einer Schleife verschwenden muss, sondern ganz unterbrochen wird, bis das erwartete Ereignis eintritt. Die Lösung mit *InterlockedExchange* könnte außerdem nicht garantieren, dass ein einmal laufender Thread jemals die Kontrolle an den wartenden Thread abgibt. Auch diese missliche Situation kann bei der Verwendung von Synchronisationsobjekten vermieden werden.

Synchronisation mit TCriticalSection R150

Dieses Beispiel sollte ausreichend Motivation liefern, die von der Kylix-Laufzeitbibliothek bereitgestellten Synchronisationsobjekte zu verwenden. Dazu ersetzen wir zunächst das gescheiterte InUseFlag durch die Variable CritSec, die grob gesehen dieselbe Funktion hat wie das InUseFlag:

```
uses SyncObjs, ... ; // << SyncObjs = Unit von TCriticalSection

type
  TThreadData = record
    CritSec: TCriticalSection;
    Bytes: array[1..DataCount] of byte;
  end;
```

Bei der Initialisierung der Threads (im *OnCreate*-Event des Formulars) wird diese Variable initialisiert (genauere Erläuterungen folgen später):

```
SharedData2.CritSec := TCriticalSection.Create;
```

Der bisherige direkte Zugriff auf das *InUseFlag* wird durch zwei Methodenaufrufe von *CritSec* ersetzt:

- Ein Aufruf von *Enter* ersetzt die Abfrage des *InUseFlag* und das anschließende Setzen dieses Flags.
- Der Aufruf von *Leave* tritt an die Stelle der Zuweisung *InUseFlag := False*.

Die geänderten Teile der Methode sehen nun wie folgt aus:

```
procedure TSyncThread.Richtig; // einer von vielen richtigen Wegen
var
  CollisionDetected: Boolean;
  i: Integer;
begin
  repeat
    // Warten, bis der Datenblock nicht mehr verändert wird
    // und gleichzeitig reservieren:
    Data^.CritSec.Enter;
    inc(IterationCount);
    ... wie oben ...
    for i := 1 to DataCount do
      Data^.Bytes[i] := 0;
    // Sperre freigeben:
    Data^.CritSec.Leave;
  until False;
end;
```

Zur Funktionsweise von TCriticalSection

TCriticalSection symbolisiert, wie der Name schon andeutet, einen kritischen Programmabschnitt; kritisch deshalb, weil er nur einmal gleichzeitig aktiv sein darf. Die Methoden *Enter* und *Leave* symbolisieren das Betreten und das Verlassen dieses kritischen Bereichs. Die Verantwortung von *TCriticalSection* liegt nun darin, jeweils nur einem Aufrufer von *Enter* gleichzeitig den Zugang zu diesem Programmabschnitt zu gestatten. Wenn also zwei Programmteile nahezu gleichzeitig *Enter* aufrufen, dann erhält der Teil, der zuerst zum Aufruf gekommen ist, den Zuschlag. Die *Enter*-Methode kehrt

zum Aufrufer zurück und dieser kann nun den kritischen Programmteil abarbeiten. Der zweite Aufrufer von *Enter* muss warten, und zwar dadurch, dass Enter *nicht* zurückkehrt, bis der kritische Abschnitt wieder frei ist, bis also der erste Programmteil die *Leave*-Methode aufgerufen hat.

TCriticalSection ist damit quasi ein Türsteher, der darauf achtet, dass sich immer nur eine Person in einem bestimmten Raum aufhält. Die Überprüfung, ob der Raum bereits belegt ist, obliegt dabei ausschließlich dem Türsteher. Die Besucher kommen einfach mit der Aufforderung »Enter« und bekommen sie mal sofort erfüllt, mal erst nach einer Wartezeit.

Zu Anfang wurde die Klasse *TCriticalSection* als »symbolisch« beschrieben. Damit war gemeint, dass ein Objekt dieser Klasse nicht selbst der kritische Programmabschnitt ist, dass es also keine wirkliche Kontrolle über diesen Programmabschnitt hat. Die tatsächliche Verantwortung liegt nämlich beim Programmierer, der die Methoden *Enter* und *Leave* korrekt aufrufen muss. Vergisst er etwa den Aufruf von *Enter*, wird *TCriticalSection* machtlos. Wichtig ist auch, dass pro kritischem Programmabschnitt nur ein *TCriticalSection*-Objekt angelegt wird. Für das Beispielprogramm bedeutet das: ein *CriticalSection*-Objekt pro Datenstruktur, auf die nur einmal gleichzeitig zugegriffen werden darf. Falsch wäre es dagegen, das *CriticalSection*-Objekt als Variable der Thread-Klasse anzulegen, denn dann gäbe es ja ein *CriticalSection*-Objekt pro Thread und die *Enter*- und *Leave*-Aufrufe wären vollkommen belanglos.

Es kann auch ohne weiteres sein, dass mehrere Progammabschnitte sich eine *CriticalSection* teilen. Das Beispielprogramm etwa besitzt einen Schalter ANHALTEN, mit dem der Benutzer einen der beiden Threads anhalten kann. Hier ist es sehr wichtig, dass der Thread nicht angehalten wird, während er sich im kritischen Bereich befindet. Denn dann würde die *Leave*-Methode nicht aufgerufen und auch der andere Thread könnte den kritischen Bereich nicht betreten, er wäre durch das Anhalten des ersten Threads blockiert.

Dies kann dadurch vermieden werden, dass die Bearbeitungsmethode für den ANHALTEN-Schalter vor dem Anhalten des Threads selbst die *Enter*-Methode aufruft. Nach dem *Enter*-Aufruf kann sie sicher sein, dass sich keiner der beiden Threads im kritischen Bereich befindet. In dieser Situation kann nun der vom Benutzer gewählte hread angehalten werden. Danach wird natürlich sofort *Leave* aufgerufen, denn mittlerweile wartet vielleicht schon der nicht angehaltene Thread vor der Schwelle zum kritischen Bereich:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ThreadSelection.ItemIndex>1 then begin
    SharedData2.CritSec.Enter;
    T[ThreadSelection.ItemIndex].Suspend;
```

```
      SharedData2.CritSec.Leave;
    end;
  end;
```

Dieses Listing dient nur als weiteres Beispiel für die Verwendung von *Enter/Leave*. Tatsächlich hält das Beispielprogramm die Threads nicht mit der *Suspend*-Methode an, sondern auf eine im weiteren Kapitelverlauf beschriebene Weise.

TEvent

Eine zweite Synchronisationsklasse der Unit *SyncObjs* ist *TEvent*. Sie kapselt praktisch das Semaphoren-Objekt des Linux-Kernels (während *TCriticalSection* übrigens die Mutex-Funktionen verwendet). Ein *TEvent*-Objekt besitzt zwei Zustände, die in den Methoden als »set« und »reset« bezeichnet werden, also quasi »gesetzt/ungesetzt« oder »signalisiert/nicht signalisiert« bedeuten. *TEvent* wird durch zwei Methoden gesteuert:

- *WaitFor* ähnelt der Methode *TCriticalSection.Enter*, denn beide warten sie auf ein bestimmtes Ereignis. Bei *TEvent* besteht dieses Ereignis darin, dass der Zustand des Objekts auf »gesetzt« geschaltet wird. Sobald dies passiert, kehrt *WaitFor* zum Aufrufer zurück, allerdings schaltet es vorher noch den Zustand wieder auf »ungesetzt«.
- *SetEvent* »setzt« das Event, gibt also einer eventuell mit *WaitFor* wartenden Methode das Signal, das sie jetzt weiterarbeiten darf.

Hinweis: *TEvent* besitzt zwar noch eine Methode *ResetEvent*, die jedoch unter Linux nichts tut, da das Event vor dem Ende der *WaitFor*-Methode automatisch zurückgesetzt wird.

Beim nächsten Thema findet sich auch eine Gelegenheit für die Anwendung von *TEvent*.

Eine neue Suspend-Methode

Wie schon angesprochen wurde, ist die *Suspend*-Methode von *TThread* aus zweierlei Hinsicht eine unsichere Sache:

- Sie simuliert das Anhalten eines Threads lediglich und könnte in zukünftigen Linux-Versionen nicht mehr funktionieren.
- Das simple Anhalten eines Threads kann zum Blockieren anderer Threads führen, nämlich wenn der angehaltene Thread über Systemressourcen verfügt, auf deren Freigabe gerade ein anderer Thread wartet.

Die Fenster der Beispielprogramme *ThreadDemo1* und *SyncDemo* verfügen daher zwar über Schalter, mit denen die Threads angehalten werden können, aber sie verwenden nicht die *Suspend/Resume*-Methoden von *TThread*, sondern die neue Unit *SuspendableThread*, die Sie auf der CD finden.

Die Grundüberlegung dabei ist, dass der zweite Nachteil der *Suspend*-Methode, das drohende Blockieren anderer Threads, nur dadurch vermieden werden kann, dass der angehaltene Thread kooperiert, indem er sich selbst anhält, sobald er keine wichtigen Systemressourcen mehr innehat. Sich selbst anzuhalten, darin besteht nämlich keine Schwierigkeit, und dafür bedarf es auch nicht der vordefinierten Methode *TThread.Suspend*. In den zurückliegenden Abschnitten wurden ja bereits zwei unterschiedliche Konstrukte vorgestellt (*CriticalSection* und *Event*), durch die ein Thread zum Warten verpflichtet wird.

Für das Anhalten von Threads eignen sich die *Events* besonders gut. Denn das Anhalten eines Threads kann verstanden werden als »Warten auf das Fortsetzen-Event«. Jeder Thread, der mit Hilfe eines *TEvent*-Objekts angehalten werden soll, benötigt daher ein eigenes *TEvent*-Objekt. Die Unit *SuspendableThread* definiert daher gleich eine neue Thread-Klasse *TSuspendableThread*, die von *TThread* abgeleitet ist und die private Variable *ResumeEvent* einführt. *TSuspendableThread* besitzt auch eine neue *Resume*-Methode, die das »Fortsetzen-Event« setzt, also quasi die Ampel auf Grün schaltet:

```
procedure TSuspendableThread.Resume2;
begin
  ResumeEvent.SetEvent;
end;
```

Nicht ganz so einfach ist die Implementation der neuen *Suspend*-Methode. Sie soll ja den Thread nicht abrupt anhalten, sondern ihm Gelegenheit lassen, alle wichtigen Systemressourcen freizugeben, auf die andere Threads warten könnten.

Dies wird in der Klasse *TSuspendableThread* nun dadurch realisiert, dass *Suspend2* lediglich eine private Variable namens *FSuspendRequest* setzt, aber den Thread nicht anhält. Der eigentliche Thread-Code (die Methode *Execute* und alle von ihr aufgerufenen Methoden) ist dafür verantwortlich, bei passenden Gelegenheiten *FSuspendRequest* abzufragen und sich selbst anzuhalten, falls *FSuspendRequest* den Wert *True* hat. Zum Anhalten wird die folgende Anweisung verwendet, die so lange wartet, bis die oben gezeigte *Resume2*-Methode aufgerufen wird:

```
ResumeEvent.WaitFor($FFFFFFFF);
```

Das folgende Listing zeigt die komplette Unit *SuspendableThread*, die den bereits gezeigten entscheidenden Anweisungen noch weitere Verwaltungstätigkeiten hinzuzufügen hat, etwa das Initialisieren des Objekts über die Klasse *TSimpleThread* (ebenfalls in *SyncObjs* deklariert und ein Nachfahre von *TEvent*, der besonders einfach zu

verwenden ist), die Freigabe des *TEvent*-Objekts im Destruktor und die Bereitstellung eines Properties *Suspended2* als Alternative zum *TThread*-Property *Suspended*:

```
unit SuspendableThread;

interface

uses Classes, SyncObjs;

type
  TSuspendableThread = class(TThread)
  private
    ResumeEvent: TEvent;
    FSuspended2: Boolean;
    FSuspendRequest: Boolean;
    procedure SetSuspended2(const Value: Boolean);
  public
    property Suspended2: Boolean read FSuspended2 write SetSuspended2;
    procedure CheckSuspendRequest;
    procedure Suspend2;
    procedure Resume2;
    destructor Destroy; override;
    constructor Create(InitSuspended: Boolean);
  end;

implementation

{ TSuspendableThread }

constructor TSuspendableThread.Create(InitSuspended: Boolean);
begin
  inherited Create(InitSuspended);
  ResumeEvent := TSimpleEvent.Create;
end;

procedure TSuspendableThread.CheckSuspendRequest;
begin
  if FSuspendRequest = True then begin
    FSuspendRequest := False;
    FSuspended2 := True;
    ResumeEvent.WaitFor($FFFFFFFF);
    FSuspended2 := False;
  end;
end;

procedure TSuspendableThread.Suspend2;
begin
  FSuspendRequest := True;
end;

destructor TSuspendableThread.Destroy;
```

```
begin
  ResumeEvent.Free;
  inherited;
end;

procedure TSuspendableThread.Resume2;
begin
  ResumeEvent.SetEvent;
end;

procedure TSuspendableThread.SetSuspended2(const Value: Boolean);
begin
  if Value <> FSuspended2 then begin
    if Value then Suspend2 else Resume2;
    FSuspended2 := Value;
  end;
end;

end.
```

Hinweis: Die Klasse *TSuspendableThread* überschreibt absichtlich nicht die geerbten Methoden von *TThread*, denn das geerbte *Resume* wird noch benötigt, um einen Thread überhaupt zu starten, wenn er nicht automatisch im Konstruktor gestartet wird.

erwendung von TSuspendableThread *R164*

Die Verwendung von *TSuspendableThread* ist nun ganz einfach. Sie wird im Folgenden anhand des Beispielprogramms *SyncDemo* gezeigt, die getroffenen Maßnahmen lassen sich aber auch auf beliebige andere Threads übertragen. Zunächst wird die Thread-Klasse nicht mehr von *TThread*, sondern von *TSuspendableThread* abgeleitet:

```
type
  TSyncThread = class(TSuspendableThread)
    ...
```

Die weitere Klassendeklaration bleibt unverändert, da alles Notwendige bereits von *TSuspendableThread* geerbt wird.

Dann wird mindestens eine wiederholt durchlaufene Programmposition gesucht, in der der Thread keine wichtigen Systemressourcen belegt. An dieser Stelle wird die geerbte Methode *CheckSuspendRequest* aufgerufen. In *SyncDemo* ist dies z. B. am Anfang der ständig durchlaufenen Schleife, unmittelbar vor der Reservierung der entscheidenden Systemressource:

```
procedure TSyncThread.Richtig;
var
  CollisionDetected: Boolean;
```

```
    i: integer;
  begin
    repeat
      CheckSuspendRequest;
      Data^.CritSec.Enter;
```

Was bleibt, ist natürlich noch der Aufruf der neuen Methoden *Suspend2* und *Resume2*. In *SyncView* werden diese aufgerufen, wenn der Benutzer die entsprechenden Schalter des Formulars drückt:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  T[ThreadSelection.ItemIndex].Suspend2;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  T[ThreadSelection.ItemIndex].Resume2;
end;
```

TThreadList

Der Schlusshinweis zum Thema der Synchronisierung ist der Klasse *TThreadList* gewidmet. Mit ihr können Sie aus mehreren Threads auf eine *TList*-ähnliche Liste zugreifen. *TThreadList* ist allerdings nicht von *TList* abgeleitet, sondern *enthält* ein *TList*-Objekt, auf das Sie auch direkt zugreifen können – wobei wieder eine *Lock*-Methode ins Spiel kommt: *TThreadList.LockList*. *TThreadList* ist so einfach zu bedienen, dass an dieser Stelle ein Verweis auf die Online-Hilfe sowie auf das Kapitel über *TList* genügen soll.

TThreadList ist eine Liste *für* Threads, nicht zu verwechseln mit der Liste *bestehend aus* Threads, die im Beispielprogramm des nächsten Kapitels vorkommen wird.

4.8.4 Ein Utility mit dynamischer Thread-Anzahl

Dieser Abschnitt zeigt am Beispiel der Anwendung *WaveProcessingThreads*, wie Sie eine sich verändernde Zahl von Threads mit Hilfe eines einfachen *TList*-Objekts verwalten können. Aus Platzgründen können die anderen Aspekte dieser Anwendung hier jedoch nur kurz besprochen werden, zur genaueren Information sei daher auf den Quelltext auf der CD verwiesen.

Als Beispiel für eine rechenintensive Aufgabe dient hier die Bearbeitung von Klangdateien im WAV-Format, wobei die genaue Art der Bearbeitung nicht von Bedeutung ist (das Beispielprogramm kann nur eine Art Chorus-Effekt zum Klang hinzufügen). Damit die Threads auch längere Zeit zum Rechnen haben, sollten Sie große WAV-Dateien verwenden. Für alle Fälle enthält das Programm auch eine Funktion, um sehr

große Klangdateien selbst zu »synthetisieren« (falls es erlaubt ist, bei der Berechnung einer einfachen Sinusschwingung steigender Frequenz schon von »synthetisieren« zu sprechen).

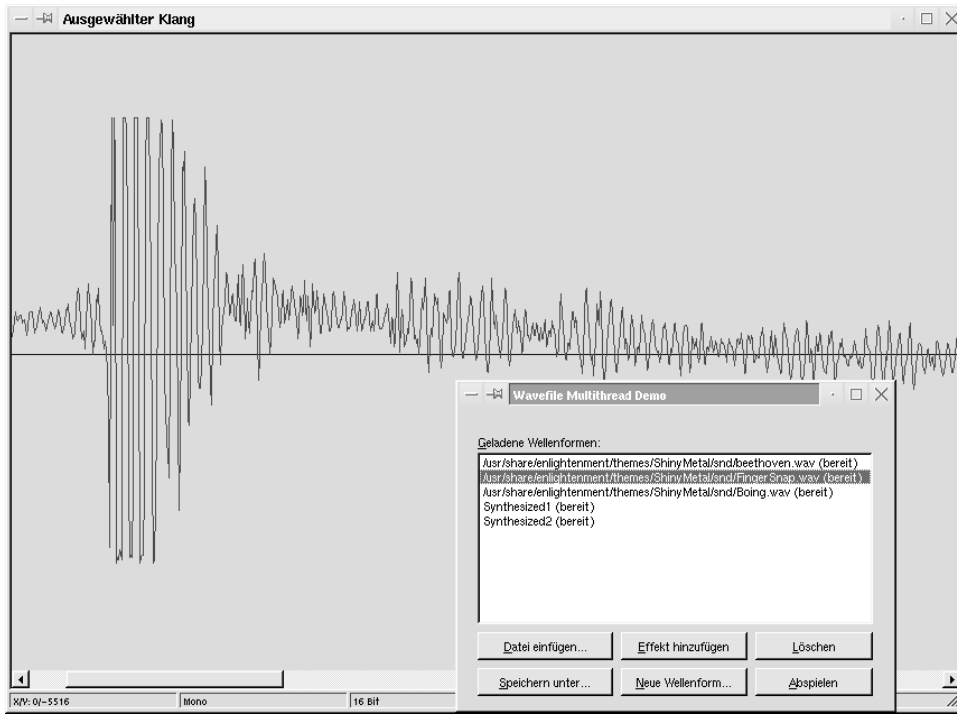


Abbildung 4.12: Die Beispielanwendung WaveProcessingThreads mit einer Liste von Klängen, von denen einige rade bearbeitet werden

Bedienung

WaveProcessingThreads erlaubt es Ihnen, mehrere WAV-Dateien zu laden, deren Namen dann in einer Liste angezeigt werden (Abbildung 4.12). Die Anwendung führt jede längere Bearbeitung einer dieser Dateien in einem eigenen Thread durch. Wenn Sie z. B. den Schalter EFFEKT HINZUFÜGEN drücken, startet die Anwendung einen neuen Thread, der die gewählten WAV-Daten mit einem Effekt versieht. Wie ebenfalls in der Abbildung zu erkennen ist, zeigt es die Anwendung in der Liste an, wenn eine Datei gerade von einem Thread bearbeitet wird.

Während dieser Bearbeitung können Sie einen anderen Klang aus der Liste wählen und auch diesen bearbeiten lassen, oder Sie können neue Klänge erzeugen (Schalter NEUE WELLENFORM...). Es ist natürlich nicht möglich, eine einzige Wellenform von mehreren Threads gleichzeitig bearbeiten zu lassen.

Implementierung der Liste von Klangdaten und Threads *R163*

Wie alle anderen Objekte können Sie natürlich auch *TThread*-Objekte in einer Liste der Klasse *TList* speichern. Während der Laufzeit der Anwendung könnten Sie dieser Liste je nach Bedarf neue Threads hinzufügen oder nicht benötigte Threads löschen. In einer Anwendung wie *WaveProcessingThreads* empfehlen sich jedoch nicht die Threads als Grundelement der Liste, sondern die Daten, mit denen diese Threads arbeiten, in diesem Fall also die Klangdaten.

Hinweis: Das Beispielprogramm benötigt keine der am Ende von Kapitel 4.8.3 erwähnten *ThreadLists*, da sein *TList*-Objekt nur von einem einzigen Thread, dem Haupt-Thread, angesprochen wird. Eine Synchronisation zwischen den verschiedenen Threads (der Daseinszweck von *TThreadList*) ist also nicht notwendig.

WaveProcessingThreads definiert daher für diese Klangdaten eine Struktur namens *TSound*. Jeder einzelne Sound enthält eine eigene *TThread*-Variable. Solange der Klang nicht bearbeitet wird, ist diese Variable auf *nil* gesetzt.

```
type
  TSound = class
    Thread: TSoundProcessThread;
    // Die eigentlichen Klangdaten:
    Data: TBigArray; // Sample-Daten, speichert auch
                     // Auflösung und Zahl der Kanäle
    SampleRate: Integer;
    FromFile: Boolean; // True -> aus Datei geladen
                       // False -> synthetisierter Klang
    // falls aus einer Datei geladen:
    FileName: string;
    Header: TWavHeader; // Header einer evtl. geladenen Datei
    // Benachrichtigung des Formulars, falls sich der
    // Thread-Status ändert (der Thread beendet wird):
    FormNotifyProc: TNotifyEvent;

    // Es gibt zwei Konstruktoren. Die Klangdaten können...
    // ... aus einer Datei geladen werden
    constructor Create(FileName: string);
    // ...oder synthetisiert werden (F kann z.B. die Sinus-Funktion sein):
    constructor CreateFromFunction(F: TSynthFunction; Rate, Size: Integer;
                                   Bytes, Channels: Byte);
    procedure ProcessSound(NotifyProc: TNotifyEvent);
    procedure ThreadTerminateNotify(Sender: TObject);

    // Hilfsfunktionen:
    procedure Play; // Ausgeben über das Multimedia-API
    procedure SaveAs(FN: string); // in Datei speichern
    procedure FillHeaderStruct;  // Datei-Header erstellen, falls
                                 // Klangdaten neu synthetisiert wurden
  end;
```

Mehrere solcher Klangobjekte werden nun in der Liste *Sounds* gespeichert, die in Form einer Formularvariablen vorliegt:

```
type
  TForm1 = class(TForm)
    ...
  public
    Sounds: TList;
```

ür die Erweiterung der Liste ist die bekannte *TList*-Methode *Add* zuständig. Bei jedem Hinzufügen eines neuen Klangobjekts wird außerdem die ListBox neu ausgegeben (zur Methode *UpdateList* siehe CD) und das neue Klangobjekt in dieser ListBox selektiert. Diese drei Aktionen sind in der Methode *AddSound* zusammengefasst:

```
procedure TForm1.AddSound(NewAuftrag: TSound);
begin
  Sounds.Add(NewAuftrag);
  UpdateList;
  ListBox.ItemIndex := ListBox.Items.Count-1;
end;
```

Der Schalter DATEI EINFÜGEN... ist mit der folgenden Methode verknüpft, die *AddSound* aufruft:

```
procedure TForm1.OpenBtnClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    AddSound(TSound.Create(OpenDialog.FileName));
end;
```

Auch der Schalter NEUE WELLENFORM... ruft *AddSound* auf, blendet vorher aber eine Dialogbox ein, die in einem weiteren Formular definiert ist, und verwendet einen zweiten Konstruktor der Klasse *TSound*. All diese Details sind jedoch unabhängig von der Verwaltung der Threads, weswegen hier nicht mehr weiter darauf eingegangen werden kann.

Starten der Threads

WaveProcessingThreads startet einen Thread, wenn Sie mit NEUE WELLENFORM... ein neues Klangobjekt erzeugen oder ein bestehendes mit EFFEKT HINZUFÜGEN verändern. Als Beispiel für die Thread-Verwaltung soll hier der letztgenannte Schalter dienen. Sein *OnClick*-Ereignis ist mit der folgenden Methode verknüpft, die das in der ListBox gewählte *TSound*-Objekt feststellt und dessen *ProcessSound*-Methode aufruft:

```
procedure TForm1.BearbeitenClick(Sender: TObject);
var
  SelectedSound: TSound;
begin
  if ListBox.ItemIndex <> -1 then begin
```

```
      SelectedSound := TSound(Sounds[ListBox.ItemIndex]);
      SelectedSound.ProcessSound(ThreadTerminateNotify);
      UpdateList;
    end;
  end;
```

Da die Thread-Variable Teil der Klasse *TSound* ist, übernimmt mit *ProcessSound* auch eine Methode dieser Klasse die gesamte Verwaltung des Threads. Falls sie aufgerufen wird, obwohl bereits ein Thread aktiv ist, tut sie nichts, außer eine entsprechende Fehlermeldung anzuzeigen. Andernfalls erzeugt sie einen neuen *TSoundProcessThread* (die Parameter des *Create*-Konstruktors spielen hier keine Rolle) und startet ihn mit der Methode *Resume*. Außerdem verknüpft sie eine andere Methode des *TSound*-Objekts mit dem *OnTerminate*-Ereignis des Threads. Der Parameter *NotifyProc* wird später erläutert.

```
procedure TSound.ProcessSound(NotifyProc: TNotifyEvent);
begin
  if not Assigned(Thread) then begin
    FormNotifyProc := NotifyProc;
    Thread := TSoundProcessThread.Create(Data, Header, Effect);
    Thread.OnTerminate := ThreadTerminateNotify;
    Thread.Resume;
  end else
    MessageDlg('Alter Thread arbeitet noch. Kein neuer Thread wurde'
               +'erstellt.', mtInformation, [mbCancel], 0);

end;
```

Die Bedeutung des OnTerminate-Ereignisses *R165*

Warum es erforderlich ist, das *OnTerminate*-Ereignis zu verwenden, können Sie aus der zuletzt gezeigten Methode *ProcessSound* erkennen. Sie stellt sicher, dass immer maximal ein Thread für ein bestimmtes Soundobjekt aktiv ist, indem sie überprüft, ob der Variable *Thread* bereits ein Objekt zugewiesen (*assigned*) ist. Dazu ist es erforderlich, dass *Thread* immer dann auf *nil* gesetzt wird, wenn kein Thread mehr arbeitet.

Wenn der Thread endet, wird *Thread* jedoch nicht automatisch auf *nil* gesetzt. Das Einzige, was beim Ende des Threads automatisch geschehen *kann*, ist, dass das Thread-Objekt gelöscht wird (falls Sie das *TThread*-Property *FreeOnTerminate* auf *True* gesetzt haben).

Das Beispielprogramm muss die Variable *Thread* also selbst auf *nil* setzen, sobald der Thread beendet ist. Es könnte theoretisch mit *Thread.WaitFor* auf diese Beendigung warten, jedoch würde *WaitFor* den Haupt-Thread blockieren und den gesamten Sinn des Threads zunichte machen. Aus diesem Grund stellt *TThread* das Ereignis *OnTerminate* zur Verfügung, das aufgerufen wird, sobald der Thread endet.

TSound verknüpft das Ereignis mit der folgenden Methode, die das aktuelle *Thread*-Objekt freigibt, die Variable auf *nil* setzt und außerdem das Formular vom Ende des Threads benachrichtigt:

```
procedure TSound.ThreadTerminateNotify(Sender: TObject);
begin
  Thread.Free;
  Thread := nil;
  FormNotifyProc(Sender);
end;
```

Dabei ist *FormNotifyProc* eine Methode des Formulars, die dieses bei der Erzeugung des Threads im Parameter *NotifyProc* der Methode *TSound.ProcessSound* angegeben hat. Das Formular möchte nämlich auch vom Ende des Threads informiert werden, da es bei diesem Anlass die ListBox neu ausgeben muss, die ja auch Auskunft über aktive Threads gibt. In der oben gezeigten Methode *BearbeitenClick* wird die folgende Methode *ThreadTerminateNotify* zur Benachrichtigung des Formulars angegeben:

```
procedure TForm1.ThreadTerminateNotify(Sender: TObject);
begin
  UpdateList;
  WavForm.Invalidate; // Fenster mit der Grafikdarstellung aktualisieren
end;
```

Wenn also einer der Threads endet, erzeugt die CLX ein *OnTerminate*-Ereignis und ruft damit *TSound.ThreadTerminateNotify* auf. Diese Methode aktualisiert das *TSound*-Objekt und ruft *TForm1.ThreadTerminateNotify* auf, die schließlich das Formular aktualisiert.

Und auf der CD finden Sie ...

... die größeren Teile des Quelltextes, die hier nicht mehr besprochen werden können. Dazu gehört das Innere der Threads, also z.B. das Berechnen des Klangeffektes (einer Art Chorus-Effekt, der mit Hilfe eines 1000 Samples fassenden Puffers realisiert ist) sowie die Ausgabe der Wellenform in einem Grafikfenster.

5 Kylix-Anwendungen in der Linux-Umgebung

Nachdem sich die Kapitel 3 und 4 mit der CLX beschäftigt haben, die vom Kylix-Linker per Voreinstellung zur Anwendungen hinzugebunden wird und als *Bestandteil* der Kylix-Anwendung betrachtet werden kann, geht es nun um die Umgebung einer Kylix-Anwendung. Hiermit sind grob gesagt die Bibliotheken des Betriebssystems gemeint, mit denen die Anwendung interagieren kann – vom Linux-Kernel bis zur direkt unter der CLX anzusetzenden Qt-Bibliothek. Aber nicht nur spezielle Bibliotheken werden in diesem Kapitel behandelt, sondern auch die allgemeine Vorgehensweise, wie Sie Ihre Kylix-Anwendung mit den (unter Linux meist in C oder C++ geschriebenen) anderen Bibliotheken oder Programmen zusammenarbeiten lassen können.

Da Kernel, Qt und KDE jeweils für sich umfangreiche Bücher füllen, können sie in diesem Kapitel ohnehin nur als Beispiel dienen. Wenn Sie jedoch das allgemeine Muster verstanden haben, wie man die genannten Bibliotheken in Kylix anspricht, können Sie alle weiteren Informationen aus der speziellen Dokumentation dieser Bibliotheken oder aus speziellen Büchern dazu beziehen, auch wenn diese alle für die Sprache C/C++ vorgesehen sind. Gewisse Grenzen der Übertragbarkeit sind allerdings zu beachten, da insbesondere die Verwendung von C++-Klassen aus Kylix heraus so umständlich ist, dass man sie vielleicht lieber mit einem C++-Tool ansprechen wird. Die fehlende Möglichkeit einer *unmittelbaren* Zusammenarbeit mit C++-Code ist vielleicht der einzige wesentliche Nachteil, der Kylix noch vor dem idealen Linux-RAD-Entwicklungssystem trennt, und vielleicht kann Borlands C++-Builder für Linux diesen Nachteil beheben.

Anmerkung: Da in diesem Kapitel alles am Beispiel von Linux erläutert werden wird, heißt es im Titel »Linux-Umgebung«, jedoch sei darauf hingewiesen, dass die Qt-Bibliothek auch unter Windows verfügbar ist und dass bei Verfügbarkeit von Delphi 6 mit CLX und Qt das hier über Qt Geschriebene auch bei Delphi 6 anwendbar sein sollte.

5.1 Der Kernel und andere C-Bibliotheken

Wir beginnen in diesem Abschnitt die Untersuchung der Linux-Umgebung einer Kylix-Anwendung bei ihrem Fundament – beim Linux Kernel. Zugleich begegnen wir dabei der einfachsten Möglichkeit, Funktionen aus externen Bibliotheken in Kylix aufzurufen. Bei dieser Gelegenheit werden einige nützliche Beispielprogramme präsentiert, die zwar sehr klein sind, deren Funktionalität sich aber bei vielerlei Gelegenheiten in anderen Programmen wiederverwenden lässt.

Der Linux-Kernel

Wie bei jedem Betriebssystem hat auch der Linux-Kernel die Aufgabe, den Anwendungen verschiedene Grundfunktionalitäten wie Ein- und Ausgabe, Dateioperationen, Prozessverwaltung usw. zur Verfügung zu stellen. Der Linux-Kernel ist unter anderem dadurch bekannt, dass man ihn selbst kompilieren kann, und zwar mit einem frei erhältlichen C-Compiler. Man kann den Linux-Kernel daher einfach als eine besondere C-Bibliothek ansehen.

Ein besonderes Merkmal dieser C-Kernel-Bibliothek ist, dass sie aufgrund der Betriebssystem-Architektur im geschützten Modus des Prozessors ausgeführt wird, während die Anwendungsprogramme im Benutzermodus laufen, in dem der Prozessor diverse Schutzregeln überprüft, nach denen eine Anwendung z. B. nicht den Speicher einer anderen Anwendung überschreiben kann. Diese Trennung zwischen Kernel- und Benutzermodus verfolgt das Ziel, dass eine Anwendung andere Anwendungen nicht beschädigen oder gar das gesamte System zum Absturz bringen kann.

ernel und Libc

Eine Konsequenz dieser Schutzmaßnahmen ist, dass die Funktionen des Kernels nur ber ein bestimmtes Protokoll aufgerufen werden können. Um dieses Protokoll zu vermeiden und die Kernel-Funktionalität im Programmieralltag wie die einer normalen (ungeschützten) Bibliothek nutzen zu können, wurden die so genannten *Systemaufrufe* (direkte Aufrufe von Kernel-Funktionen) in einer weiteren C-Bibliothek, der *libc*, verpackt. So gibt es beispielsweise im Kernel die Funktion *open* zum Öffnen einer Datei. Um diese nicht direkt (über das umständliche Aufrufprotokoll) aufrufen zu müssen, definiert auch die *libc* eine Funktion namens *open*, die wie eine normale Funktion aufgerufen wird und ihrerseits das Kernel-Aufruf-Protokoll implementiert.

Wenn im Folgenden von der C-Bibliothek die Rede ist, ist damit die erwähnte *libc* gemeint und nicht der eigentliche Kernel. In Kylix ist diese Bibliothek über die Unit *Libc* ansprechbar, wobei es auf das große »L« ankommt, da bei Unit-Namen zwischen Groß- und Kleinschreibung unterschieden wird. Die Unit *Libc* liefert ein ideales Beispiel dafür, wie Sie C-Funktionen in Object Pascal aufrufen können. Sie finden diese Unit unter dem Namen `Libc.pas` im Verzeichnis `kylix/source/rtl/linux`.

5.1.1 Aufruf von Bibliotheks-Funktionen allgemein

Wenn ein Funktion aufgerufen werden soll, die in einer getrennt kompilierten Bibliothek enthalten ist, kann der Compiler keine Sprungadresse in den Maschinencode schreiben, sondern muss sich mit einem Verweis auf die Funktion behelfen, der erst beim Starten des Programms aufgelöst werden kann. Diese Auflösung des Verweises wird auch als späte Bindung bezeichnet und durch den Programmlader von Linux durchgeführt.

Bei der Programmierung müssen Sie sich jedoch nicht um diese Details kümmern, denn dies ist Aufgabe von Compiler, Linker und dem Betriebssystem (genauer: dessen Programmlader, auch Runtime Linker genannt). Damit die externe Funktion gefunden werden kann, besteht Ihre Aufgabe maximal darin,

- Kylix mitzuteilen, in welcher Bibliothek eine bestimmte Funktion zu suchen ist, damit der Compiler die entsprechenden Verweise an den Linker weitergeben kann.
- dem System mitzuteilen, wo er diese Bibliothek finden kann, sofern sie nicht in einem der automatisch eingestellten Suchpfade enthalten ist.

Damit der korrekte Aufruf möglich ist, sind zwei weitere Schritte erforderlich:

- Die Parameter der Funktion müssen deklariert werden, wie bei jeder normalen Kylix-Funktion auch.
- Für das Aufrufformat, mit der die Funktion auf der Maschinen-Ebene aufgerufen wird, gibt es verschiedene Variationen, die sich z.B. darin unterscheiden, ob die Funktionsparameter in der Reihenfolge ihrer Deklaration oder umgekehrt auf den Stack geschrieben werden. Da der Compiler dieses Aufrufformat für externe Funktionen nicht »erraten« kann, müssen Sie es ihm explizit mitteilen.

Finden der Bibliotheken

Die Standardverzeichnisse für Bibliotheken unter Linux sind `/lib` und `/usr/lib`. Viele Libraries werden nicht direkt in diesen Verzeichnissen gespeichert, sondern es wird nur ein symbolischer Link auf die tatsächliche Position der Datei in diesen Verzeichnissen abgelegt. Für Bibliotheken, die nicht von systemweitem Interesse sind, ist es jedoch naheliegender, sie ganz aus `/lib` und `/usr/lib` herauszuhalten (immerhin sind wir hier nicht unter Windows, wo die Unsitte herrscht, dass viele Programme ihre dynamischen Bibliotheken einfach in das Windows-Systemverzeichnis hineinkopieren).

Wenn Sie Ihre Bibliothek an einem individuellen Ort gelagert haben, bleiben Ihnen zwei Möglichkeiten, dem System die Position mitzuteilen: Am schnellsten und wenigsten haltbar ist es, wenn Sie vor dem Programmstart die Umgebungsvariable

LD_LIBRARY_PATH um den Pfad Ihrer Bibliothek erweitern; im folgenden Beispiel wird neben dem Verzeichnis der Kylix-Bibliotheken auch ein eigenes Verzeichnis `mylib` definiert:

```
LD_LIBRARY_PATH=/root/kylix/bin:/root/mylib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
programstart
```

Wenn Sie eine Kylix-Anwendung außerhalb der IDE starten wollen, so müssen Sie zumindest den Pfad der *libqtintf.so*, normalerweise `kylix/bin`, in den Bibliothekssuchpfad aufnehmen.

Diese Pfad-Erweiterung lässt sich natürlich auf vielerlei Arten automatisieren wie z.B. durch eine Skript-Datei oder im System permanent machen wie durch Verankerung der Erweiterung in der Konfigurationsdatei `.bashrc`, die bei jedem Start der Standard-Linux-Shell ausgewertet wird.

Die zweite Methode, den Programmlader mit einem neuen Bibliothekspfad vertraut zu machen, besteht darin, den Pfad in die Datei `etc/ld.so.conf` zu schreiben und danach das Programm `ldconfig` auszuführen. Dieses trägt dann die im neuen Pfad gefundenen Bibliotheken im vom Runtime-Linker verwendeten Cache-Speicher ein (der Datei `/etc/ld.so.cache`). Die Namen der genannten Dateien sind übrigens vom Dateinamen `ld.so` des Runtime-Linkers abgeleitet.

Analyse des Linkvorgangs

Wenn der Programmlader nicht alle Bibliotheken finden kann, die er für den Start eines Programms sucht, erhalten Sie eine entsprechende Meldung unter Angabe der nicht gefundenen Bibliothek. In diesem Fall ist es meist nicht weiter schwierig, die Bibliothek ausfindig zu machen und die Suchpfade auf eine der oben beschriebenen Weisen zu ergänzen.

Wenn Sie jedoch genau wissen wollen, welche Bibliotheken wo bei einem bestimmten Programmaufruf hinzugelinkt werden, können Sie sich des Programms `ldd` bedienen. Sie übergeben ihm den Namen eines Programms und erhalten dafür eine Liste aller zum Starten dieses Programms erforderlichen Bibliotheken mitsamt dem Pfad, an dem eine jede einzelne Bibliothek gefunden wurde (manchmal existieren verschiedene Versionen einer Bibliothek in verschiedenen Pfaden, so dass die Information, welche davon nun verwendet werden würde, sehr hilfreich sein kann).

Das folgende Listing zeigt das `ldd`-Protokoll für den TreeDesigner, nachdem er mit den XML-Funktionen übersetzt wurde:

```
[elmar@localhost]# ldd Treedsgn
/lib/libNoVersion.so.1 => /lib/libNoVersion.so.1 (0x40015000)
libqtxml_wrapper.so => /usr/lib/libqtxml_wrapper.so (0x4001f000)
libqtintf.so => /root/kylix/bin/libqtintf.so (0x40025000)
```

```
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x401a9000)
libpthread.so.0 => /lib/libpthread.so.0 (0x40247000)
libdl.so.2 => /lib/libdl.so.2 (0x4025a000)
libc.so.6 => /lib/libc.so.6 (0x4025e000)
libstdc++-libc6.1-1.so.2 => /usr/lib/libstdc++-libc6.1-1.so.2 (0x40353000)
libm.so.6 => /lib/libm.so.6 (0x40395000)
libqt.so.2 => /usr/local/qt/lib//libqt.so.2 (0x403b2000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x40a32000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x40a3d000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x40a46000)
libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40a5b000)
```

Wenn der TreeDesigner ohne die XML-Funktionen kompiliert wird, erscheint die Bibliothek *libqtxml_wrapper.so*, mit der die vom TreeDesigner benötigten ausgewählten XML-Funktionen von Qt überhaupt erst in Object Pascal genutzt werden können, nicht in der Abhängigkeitsliste.

Wenn der Bibliothekspfad nicht richtig eingestellt ist und *ldd* die von Kylix bereitgestellte *libqtintf.so* nicht finden kann, erscheint eine entsprechende Meldung und eine ganze Reihe der oben aufgelisteten Bibliotheken werden nicht angezeigt:

```
[elmar@localhost]# ldd Treedsgn
/lib/libNoVersion.so.1 => /lib/libNoVersion.so.1 (0x40015000)
libqtintf.so => not found
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x4001e000)
libpthread.so.0 => /lib/libpthread.so.0 (0x400bb000)
libdl.so.2 => /lib/libdl.so.2 (0x400cf000)
libc.so.6 => /lib/libc.so.6 (0x400d3000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Die gegenüber dem ersten Listing fehlenden Bibliotheken sind genau die Bibliotheken, die direkt oder indirekt von *libqtintf.so* verwendet werden, aber von keiner anderen und auch nicht von der Anwendung selbst. Da *ldd* die *libqtintf* nicht finden konnte, kann das Programm auch nicht wissen, welche Bibliotheken von ihr vorausgesetzt werden. So fehlen im zweiten Listing also im Wesentlichen das Qt-Toolkit und die damit zusammenhängenden Bibliotheken.

Hinweis: Die beim Start des Programms erforderlichen Bibliotheken sind nicht gleichzusetzen mit »alle Bibliotheken, die das Programm verwendet«, denn dynamische Bibliotheken können auch noch *nach* dem Programmstart vom Programm geladen werden (dann sind sie sozusagen noch dynamischer als die beim Start geladenen Artgenossen). Wenn ein Programm jedoch eine Bibliothek erst zur Laufzeit lädt, dann geschieht dies meist zur Ausführung einer optionalen Funktion. In diesem Fall sollte ein Programm auch damit zurecht kommen, wenn die Bibliothek nicht vorhanden ist. Ein Beispiel dafür, wie Sie so ein Programm mit Kylix entwickeln können, finden Sie in Kapitel 5.5.2.

5.1.2 Deklarationen von externen Funktionen in Object Pascal

Bevor wir dies an einem eigenen Programm ausprobieren, soll das erwähnte Idealbeispiel herangezogen werden: die *Libc*-Unit von Kylix, die die Funktionen des Linux-Kernels deklariert, welche indirekt über die *libc*-Bibliothek von Linux bereit gestellt werden.

Als Beispiel soll das Erzeugen einer Datei dienen, welches natürlich über einen Systemaufruf des Kernels möglich ist. Die *libc*-Bibliothek enthält eine *creat*-Funktion, die diesen Systemaufruf durchführt und selbst wie eine normale Funktion aufgerufen werden kann.

Hinweis: Normalerweise gilt, dass Sie eine Funktion der Kylix Laufzeitbibliothek (kurz RTL) der *libc*-Funktion vorziehen sollten, sofern die RTL eine Funktion für dieselbe Aufgabe bereitstellt (egal ob unter gleichem oder anderem Namen), denn die Object-Pascal-Funktionen arbeiten mit den originalen Object-Pascal-Stringtypen und sind auch unter Delphi für Windows verfügbar, also portabel. Die *creat*-Funktion dient hier also nur als Beispiel und braucht normalerweise in keiner Kylix-Anwendung aufgerufen zu werden.

Funktionsparameter und Aufrufformat

Wenn Sie auf einer Linux-Konsole *man creat* aufrufen, erhalten Sie Informationen über die Parameter dieser Funktion:

```
int creat(const char *pathname, mode_t mode);
```

Diese C-Deklaration besagt, dass die Funktion einen *Integer* als Rückgabewert liefert und einen Pfadnamen als String-Parameter (*char* *) sowie einen weiteren Parameter des Typs *mode_t* erwartet (zu Details der Übersetzung von C-Typen in Object-Pascal-Typen kommen wir später).

In der Unit *Libc* ist diese Deklaration wie folgt nach Object Pascal übersetzt worden:

```
function creat(FileName: PChar; Mode: __mode_t): Integer; cdecl;
```

Bis auf den Zusatz von *cdecl* für das Aufrufformat auf Maschinenebene und den Bezeichner *__mode_t* ist dies eine normale Object-Pascal-Deklaration. Der Typbezeichner *__mode_t* entspricht zwar nicht exakt dem *mode_t*, welches in der *man*-Dokumentation angegeben ist, jedoch hat Borland den Namen nicht verändert, sondern aus der Linux-Include-Datei *fcntl.h* übernommen. Allerdings gibt es in Object Pascal keinen solchen Typen und daher musste Borland natürlich auch die Deklaration dieses Typs aus den C-Dateien übersetzen. An zwei weiteren Stellen *Libc.pas* finden wir in Typendeklarationsblöcken folgende Gleichsetzungen:

```
__mode_t = __u_int; (* Dies ist eine Übersetzung der C-Deklaration
                       typedef __u_int __mode_t; aus bits/types.h *)
__u_int = Cardinal; (* Dies ist eine Übersetzung der C-Deklaration
                       typedef unsigned int __u_int; aus bits/types.h *)
```

Die obigen Kommentare stehen zwar nicht in *Libc.pas*, aber es ist dort über Deklarationsabschnitten immer wie eine Überschrift angegeben, welche C-Header-Datei übersetzt wurde.

Mit der Übersetzung von Typen beschäftigt sich ein späterer Abschnitt. Wichtig ist hier nur, dass durch die Rückführung von *__mode_t* auf den Object-Pascal-Typen *Cardinal* der Parameter *Mode* dasselbe Format bzw. dieselbe Größe hat wie der entsprechende Parameter in der C-Deklaration (kurz: *unsigned int* von C entspricht *Cardinal* von Object Pascal).

Hinweise: Um Namensverwechslungen mit den traditionellen Funktionen der Delphi-Runtime Library zu vermeiden, hat Borland einige weitere Bezeichner von *libc* durch Voranstellung von zwei Unterstrichen unterscheidbar gemacht. So wurde etwa aus *chdir* der Name *__chdir* und aus *time* *__time*. Sie finden diese Umbenennungen zu Beginn von *Libc.pas* dokumentiert. Die Namen der Parameter wurden weniger streng von C übertragen, was Sie schon am obigen Beispiel sehen können, in dem aus dem kleingeschriebenen *mode* ein großes *Mode* wurde.

External-Deklaration

Mit den bisherigen Deklarationen wurde also der korrekte Aufruf der C-Funktion vorbereitet, aber die grundlegendere Aufgabe besteht darin, dem Compiler mitzuteilen, wo er die Funktion überhaupt finden kann. Der Compiler muss mitgeteilt bekommen, dass die deklarierte Funktion nicht innerhalb dieser Unit zu suchen ist, sondern in einer externen Bibliothek vorliegt. Anstatt einer Implementierung der Funktion finden Sie zu diesem Zweck im Implementationsteil der Unit lediglich die folgende Zeile:

```
function creat; external libcmodulename name 'creat';
```

Die Deklaration von *libcmodulename* finden Sie an einer anderen Stelle der *Libc*-Unit wie folgt:

```
const
  libcmodulename = 'libc.so.6';
```

libcmodulename ist damit lediglich ein »Kürzel« für den tatsächlichen Dateinamen der Bibliothek. Wie in Abschnitt 5.3 genauer erläutert werden wird, kann sich der Dateiname aufgrund einer neuen Bibliotheks-Version leicht ändern. Da dieser Dateiname in *Libc.pas* bei über 1000 externen Funktionen genannt werden muss, bietet es sich an, ihn in der Konstanten *libcmodulename* zu speichern, so dass dann später, wenn sich der Dateiname ändert, nur diese eine Stelle aktualisiert werden muss anstatt alle 1000+x Referenzen auf diesen Namen.

Durch die Angabe von *external libcmodulename* erfährt der Compiler also, dass der (Runtime-)Linker später in dieser Datei nach der Funktion suchen soll. Mit *name 'creat'* wird schließlich noch der Name genannt, unter dem die Funktion in der Bibliothek auftaucht. Es ist also möglich, der Funktion in Object Pascal einen anderen Namen zu geben (wenn die Namen in der Bibliothek und der Object-Pascal-Deklaration übereinstimmen, kann man die *name*-Direktive auch weglassen).

Hinweis: Wenn Sie die *Libc.pas* selbst anschauen, werden Sie als Begleitung der externen Symbole auch noch die Compileranweisung *$EXTERNALSYM* finden. Diese hängt mit der Verbindung von Delphi und Borland C++-Builder zusammen und ist nicht erforderlich, wenn Sie eigene Extern-Deklarationen vornehmen wollen.

5.1.3 Aufruf der externen Funktionen

Um nun die *creat*-Funktion in einer Kylix-Anwendung aufzurufen, genügt es, die Unit *Libc* einzubinden und dann *creat* wie eine normale Funktion aufzurufen, eine für den Dateinamen angegebene Stringkonstante wird dabei automatisch in einen *PChar* umgewandelt, wenn Sie eine Stringvariable angeben, müssen Sie die Umwandlung explizit vornehmen:

```
uses
  Libc, ...;

...
FileHandle := creat('test1.temp', S_IRUSR or S_IWUSR or S_IRGRP);
(* oder mit String-Variabler: creat(PChar(name), S_IRUSR or ...); *)
```

Hierbei gibt die Verknüpfung der diversen *S_Ixxxx*-Konstanten die Zugriffsrechte an, mit der die neue Datei ausgestattet werden soll, jede einzelne Konstante steht für ein einzelnes Lese- (»R« in der Mitte) oder Schreibrecht (»W« in der Mitte), und zwar für den Besitzer der Datei (»USR« am Ende), die Gruppe des Besitzers (»GRP«-Endung) oder alle anderen (»OTH«). Im obigen Beispiel würde der Benutzer der Datei Lese- und Schreibzugriff erhalten, alle anderen nur Lesezugriff (zu Details siehe *man creat*). Das Ergebnis der Funktion, *FileHandle*, könnte nun für andere *Libc*-Dateioperationen verwendet werden, es sei denn, es zeigt durch den Wert »-1« an, dass beim Erzeugen der Datei ein Fehler aufgetreten ist.

Hinweis: Die in der *man*-Dokumentation angegebenen Zahlenkonstanten für die Zugriffsrechte (z. B. *00644* für die oben gezeigte Verknüpfung) lassen sich nicht nach Object Pascal übertragen, weil es sich um Oktalziffern handelt, die erst in Dezimal- oder Hexadezimalziffern umgerechnet werden müssten.

Allgemein müssen auch die für die Parameter verwendeten C-Konstanten wie oben *S_Ixxxx* in Object Pascal neu deklariert werden, damit Sie sie benutzen können. Für die von der *libc* verwendeten Konstanten hat dies Borland natürlich schon erledigt. Wenn Sie einmal eine spezielle C-Bibliothek verwenden wollen, in der andere Konstanten vorkommen, müssen Sie diese Deklaration selbst vornehmen, nach dem Muster von *Libc.pas*:

```
const
  __S_IREAD = $100;   { Read by owner }
  S_IRUSR = __S_IREAD;
```

Fehlerbehandlung

Die komfortable Exception-Behandlung von Object Pascal, bei der die aktuelle Ereignisbearbeitung im Falle eines Fehlers (wie zum Beispiel dem Zugriff auf einen nicht vorhandenen Listbox-Index) automatisch abgebrochen wird und ein Meldungsfenster angezeigt wird, erstreckt sich nur auf den Bereich der in Object Pascal geschriebenen Bibliotheken (z.B. die CLX). Funktionen aus externen Bibliotheken müssen einen anderen Weg nutzen, um Fehlerinformationen an die Anwendung weiterzugeben.

Die C-Bibliothek von Linux (libc) geht hier den nahe liegenden Weg, Fehler über bestimmte Ergebniscodes mitzuteilen. Wie schon erwähnt, zeigt »-1« im Falle von *creat* einen solchen Fehler an. Weitere Informationen müssen in diesem Fall von der Funktion *errno* ermittelt werden, die auch in *Libc* deklariert ist, mitsamt den von ihr zurückgelieferten Fehlerkonstanten wie etwa *EROFS*, welche angibt, dass die Funktion daran gescheitert ist, dass das verwendete Dateisystem nur gelesen werden.

5.1.4 Typübersetzung von C nach Object Pascal

Bei der Übersetzung einer Funktionsdeklaration von C nach Pascal kommt es neben dem Aufrufformat *cdecl* vor allem auf die Typen von Parametern und Rückgabewerten an. Dieses Kapitel zeigt, welche Object-Pascal-Datentypen Sie für die grundlegenden Datentypen von C verwenden können. Wie im Beispiel des letzten Kapitels werden anstatt der grundlegenden Datentypen zwar meist neu definierte Typen verwendet (z.B. *__mode_t* statt *unsigned int*), jedoch lassen sich diese alle auf die grundlegenden Typen zurückführen (im Beispiel auf den Typen *unsigned int*).

Zahlen, Zeichen und Zeiger

Die folgende Tabelle listet zunächst die einfachen Typen beider Sprachen auf, die dritte Spalte nennt Beispiele für Namen, unter denen diese Typen ebenfalls auftreten, sowie die Header-Dateien, in denen diese Namen per *typedef* definiert sind:

C-Typ	Object-Pascal-Typ	Alternative Bezeichnung
unsigned char	Byte	types.h: u_char
unsigned short	Word	types.h: u_short
unsigned int (generisch)	Cardinal (generisch)	types.h: u_int
unsigned long	LongWord	types.h: u_long
unsigned long long int	UInt64	types.h: u_quad_t
signed char	Shortint	types.h: int8_t
signed short int	Smallint	types.h: int16_t
signed int (generisch)	Integer (generisch)	types.h: ptrdiff_t
int (generisch)	Integer	
signed long int	LongInt	types.h: int32_t
signed long long int	Int64	types.h: int64_t
double	Double	
long double	Extended	cdefs.h: __long_double_t
oid *	Pointer	cdefs.h: __ptr_t
char *	PChar	
char **	PPChar	
oid (als Rückgabewert einer Funktion)	- (Prozedur statt Funktion verwenden!)	

Die in Kapitel 2 genauer erläuterte Unterscheidung zwischen generischen und fundamentalen Datentypen gibt es also auch in C, wo der normale (vorzeichenbehaftete) *int*-yp dem generischen, plattformabhängigen *Integer* von Pascal entspricht und der *unsigned int* dem *Cardinal*.

Der in der Tabelle genannte *UInt64* ist kein vordefinierter Typ von Object Pascal, sondern ein in *Libc* von Borland definierter Typ, der einen möglichst guten Ersatz für den ansonsten nicht vorhandenen 64-Bit-Integer ohne Vorzeichen darstellen soll (*UInt64* hat nur eine Kapazität von 63 Bit).

Weitere Informationen zum *PChar*-Typ finden Sie in Kapitel 2.4.4.

Aufzählungstypen

Die Aufzählungstypen von Pascal werden in C durch das Wort *enum* definiert. Die Deklarationssyntax ist der von Pascal sehr ähnlich:

```
enum FocusPolicy {
  NoFocus = 0,
  TabFocus = 0x1,
  ClickFocus = 0x2,
  ...
}
```

Dieser Typ könnte in Object Pascal wie folgt definiert werden:

```
type
  FocusPolicy = (
    NoFocus = 0,
    TabFocus = 1,
    ClickFocus = 2,
    ...
  );
```

Das Beispiel entstammt übrigens der Qt-Bibliothek, wo *FocusPolicy* allerdings ein in eine Klasse geschachtelter Typ ist, der in der Unit *Qt.pas* zwar so ähnlich wie oben gezeigt übersetzt wird, aber unter dem Namen *QWidgetFocusPolicy*.

Das Entscheidende bei der Übergabe von Aufzählungstyp-Parametern ist, dass der Compiler von Kylix eine Variable mit der richtigen Größe und dem richtigen Inhalt (dem richtigen ordinalen Wert) an die C-Funktion übergibt.

- Die Größe eines *enum*-Typs von C beträgt normalerweise 4 Bytes, während in Object Pascal per Voreinstellung nur soweit nötig eins, zwei oder vier Bytes verwendet werden, also z.B. nur ein Byte, wenn der Aufzählungstyp weniger als 256 verschiedene Werte hat. Damit der Compiler *alle* Aufzählungstypen wie in C als 4-Byte-Werte speichert, geben Sie im Quelltext die Direktive *{$MINENUMSIZE 4}* an.
- Mit »richtigem Inhalt« ist gemeint, dass der Compiler im obigen Beispiel für den Wert *TabFocus* eine 1 übergibt. Wenn die verschiedenen Aufzählungswerte in C also nicht mit aufeinander folgenden Zahlen versehen wurden, so ist es wichtig, diese Werte auch in Object Pascal explizit anzugeben (wie im obigen Beispiel geschehen, obwohl die Zahlen dort aufeinander folgend sind).

Die tatsächlich übergebene Zahl muss in Object Pascal nicht unbedingt als Aufzählungstyp deklariert sein. Sie könnten also im obigen Beispiel auf die Deklaration von *FocusPolicy* in Object Pascal verzichten und den Typ *FocusPolicy* einfach als *Integer* übersetzen, da dieser auch 4 Bytes Platz beansprucht. Bei der Übergabe von Werten könnten Sie dann die Zahlenwerte direkt angeben oder Konstanten dafür definieren, z.B.:

```
const
  NoFocus = 1;
  TabFocus = 1;
  ...
```

Dies ist im Übrigen auch der Weg, den Borland für einige *enum*-Typen der *libc* eingeschlagen hat. Diese Typen wurden in *libc* ohne Namen definiert und dienen nur dazu, Konstantenbezeichner zu vereinbaren, z.B. in der Datei *wordexp.h:*

```
enum {
  WRDE_DOOFFS = (1 << 0),
  WRDE_APPEND = (1 << 1),
  WRDE_NOCMD = (1 << 2),
  ...
}
```

Parameter, die diese Werte erhalten können, wurden einfach als *int* deklariert. Folglich wird daraus in Object Pascal ein *Integer*; die Aufzählungswerte werden zu Konstanten (Aufzählungstypen ohne Namen gibt es in Object Pascal nicht):

```
const
  WRDE_DOOFFS = 1 shl 0;
  WRDE_APPEND = 1 shl 1;
  ...
```

Records

Da sich Records in C wie in Object Pascal aus einfacheren Datentypen zusammensetzen, die oben bereits besprochen wurden, braucht über die Übersetzung der einzelnen Record-Elemente hier nichts mehr gesagt zu werden.

Worauf es bei Records zusätzlich ankommt ist, dass die einzelnen Felder in der gleichen Reihenfolge und in den gleichen Abständen im Speicher aufbewahrt werden – schon bei der Übergabe der Parameter an Funktionen haben wir ja festgestellt, dass C hier standardmäßig eine umgekehrte Reihenfolge verwendet als Object Pascal.

Die Reihenfolge der Daten ist jedoch diesmal übereinstimmend, so dass wir nur noch für die richtigen Abstände Sorge tragen müssen. Es gibt verschiedene Arten, auf die ein Compiler die Record-Felder anordnen kann: Er kann sie hintereinander schreiben, so dass etwa ein Byte, ein Word und ein Integer insgesamt 7 Bytes beanspruchen und das Word beim zweiten Byte beginnt (also an der Adresse: Record-Anfang + 1).

Moderne Prozessoren können jedoch viel schneller auf Speicheradressen zugreifen, die durch ein bestimmtes Vielfaches von 2 teilbar sind. Daher richten Compiler die Felder von Records normalerweise an diesen Adressen aus. Eine Ausrichtung von 4 Bytes würde im genannten Beispiel bewirken, dass das Byte an Offset 0, das Word an Offset 4 und der Integer an Offset 8 gespeichert würden. Jedes der drei Felder würde damit tatsächlich 4 Bytes verbrauchen, obwohl alle drei ganz unterschiedliche Nutzgrößen aufweisen.

Auch dieses interne Detail können Sie in Kylix durch eine Compilerdirektive steuern, und zwar durch *$Align*. Die für die Kompatibilität mit der *libc* erforderliche Ausrichtung an 4-Byte-Vielfachen erreichen Sie beispielsweise durch *{$ALIGN 4}*. Wenn Sie vor *cord* das Schlüsselwort *packed* angeben, verzichtet der Compiler für diesen einen Record auf jegliche Ausrichtung und die Felder folgen einander ohne Abstand. Ein Beispiel für die Übergabe eines Records als Parameter gibt der nächste Abschnitt:

Zeiger und Referenzen

Durch die Übersetzung von Funktionsdeklarationen nach Object Pascal kann es sogar sein, dass sich die Handhabung der Funktion vereinfacht. Das ist besonders dann der Fall, wenn eine C-Funktion einen Zeiger als Parameter erwartet. Wie weiter oben in einer Tabelle schon erwähnt, lassen sich C-Zeiger zwar als *Pointer* übersetzen, in manchen Fällen bietet es sich jedoch an, den Parameter in Pascal durch einen *var*-Parameter darzustellen, z. B.:

```
(* void GetFourIntegers(int *a, int *b, int *c, int *d); *)
   procedure GetFourIntegers(var a, b, c, d: Integer); cdecl;
```

Hier übergibt der Object-Pascal-Compiler ebenfalls nicht direkt vier Zahlenwerte, sondern Zeiger darauf, die exakt den *int* *-Zeigern von C entsprechen.

Im nächsten Beispiel wird in C ein Zeiger auf einen Record erwartet. Dieser lässt sich in Object Pascal wieder als *var*-Parameter angeben, mit der Konsequenz, dass man beim Aufrufen der Funktion gar nicht daran zu denken braucht, dass es sich intern um einen Zeiger handelt:

```
(* int statfs(const char *path, struct statfs *buf); *)
   function statfs(path: PChar; var buf: statfs);
```

Arrays

Arrays beginnen in C immer an der Position 0, in der Deklaration wird nur die Länge des Arrays (Zahl der Elemente) angegeben:

```
// int IntSequence[10];
// sieht in Object Pascal wie folgt aus:
   IntSequence: array[0..9] of Integer;
```

Oft werden in C jedoch einfach Zeiger auf Arrays übergeben, vor allem, wenn die Zahl der Array-Elemente offen ist. Der »Zeiger auf ein Array« unterscheidet sich in C allerdings nicht von einem Zeiger auf ein einzelnes Element (ein Zeiger auf einen Integer ist in C also das Gleiche wie ein Zeiger auf ein Integer-Array). Um einen solchen Zeiger wie in C behandeln zu können, sind in Object Pascal allerdings zwei vorbereitende Typendeklarationen erforderlich:

```
// int *GetArray();
// könnte z.B. wie folgt übersetzt werden:
type
  TIntArray: array[0..(MaxLongint div Sizeof(Integer))-1] of Integer;
  PIntArray: ^IntArray;

function GetArray: PIntArray;
```

Dabei wird *TIntArray* als Array maximaler Größe deklariert (*MaxLongint* gibt die größte 32-Bit-Integer-Zahl an, geteilt durch die Größe eines Integers ergibt sich so die maximale Zahl an Integers, die theoretisch in einem Array gespeichert werden können). *PIntArray* ist ein Zeiger auf dieses Array. *GetArray* wird zwar in Wirklichkeit einen Zeiger auf ein kleineres Array zurückgeben (wie dessen Größe definiert ist, muss aus der Dokumentation der jeweiligen C-Funktion hervorgehen), aber Sie können nun auf dieses Array unbeschränkt zugreifen:

```
var
  ArrayFromCFunction: PIntArray;
begin
  ArrayFromCFunction := GetArray;
  ShowMessage(IntToStr( ArrayFromCFunction[1000] ));
```

Die Deklaration von *TIntArray* als maximal großes Array diente also nur dem Zweck, eventuelle Überprüfungen der Array-Indizes durch den Compiler zu übergehen, denn wenn das Array nur als 100-elementig deklariert wäre, würde der Compiler einen Zugriff auf Element [100] nicht zulassen. (Wir sind ja von der Annahme ausgegangen, dass die Größe des Arrays beliebig groß sein kann. Bei einer vorher feststehenden Größe können Sie diese natürlich auch in der Object-Pascal-Deklaration fest angeben.)

Weitere Informationen

In der Object-Pascal-Sprachreferenz, die auch Teil der Online-Hilfe ist, finden Sie im Kapitel *Der Speichermanager* detaillierte Informationen über die Speicherbelegung durch die verschiedenen Datentypen. Dieses Kapitel diente dazu, die Übersetzung der am häufigsten verwendeten Typen zu erleichtern und die Grundprinzipien zu demonstrieren, auf denen eine solche Übersetzung basiert: Übereinstimmung herzustellen in der Größe der Variablen (Zahl der Bytes), im Inhalt (Inhalt der einzelnen Bits) und in der Abfolge im Speicher (*$Align* bei Record-Feldern, *cdecl* für Funktionsparameter).

5.1.5 Ein Mini-Systemmonitor als Beispiel

Dieses Kapitel stellt die kleine Beispielanwendung *MiniSysMon* vor, die zwei Funktionen der *Libc* verwendet, die bereits von Borland übersetzt wurden. Das Programm soll in einstellbaren regelmäßigen Abständen Informationen über den freien Hauptspeicher oder die verfügbare Festplattenkapazität für das Wurzelverzeichnis ausgeben (siehe Abbildung 5.1). Erstere Daten werden durch die Linux-Funktion *sysinfo*, letztere durch *statfs* ermittelt.

Das Programm berechnet außerdem die Änderung des freien Speichers zwischen den einzelnen Abfrageintervallen und stellt diese Änderungen grafisch dar. Für die grafische Darstellung ist die Komponente *TMiniHistoryGraph* zuständig, die der Autor ein-

mal während früher Experimente mit Kylix in seiner Freizeit entwickelt hat und deren Handhabung nicht das Thema dieses Kapitels sein soll (siehe letzten Abschnitt in diesem Kapitel).

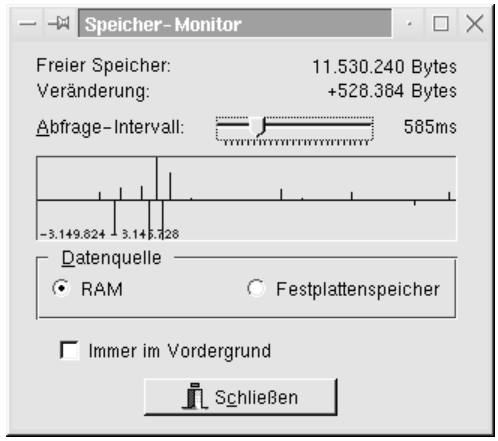


Abbildung 5.1: Ein kleiner mit Libc-Funktionen arbeitender Monitor für freien Speicher.

Eine Record-Struktur in C und Object Pascal

Die Libc-Funktion *statfs* liefert eine ganze Reihe von Informationen über das »Heimat-Dateisystem« einer bestimmten Datei zurück. Diese Informationen werden in einem Record gespeichert und dieser liefert uns zunächst ein Beispiel für die Übersetzung eines C-Records in eine Object-Pascal-Deklaration.

So finden sich in den Dateien `bits/fstat.h` und `sys/fstat.h` (jeweils zu finden im allgemeinen Include-Verzeichnis, normalerweise `/usr/include`) die folgenden Deklarationen, hier um Sonderfälle der bedingten Kompilierung gekürzt:

```
struct statfs
  {
    int f_type;
    int f_bsize;
    __fsblkcnt_t f_blocks;
    __fsblkcnt_t f_bfree;
    __fsblkcnt_t f_bavail;
    __fsfilcnt_t f_files;
    __fsfilcnt_t f_ffree;
    __fsid_t f_fsid;
    int f_namelen;
    int f_spare[6];
  };

extern int statfs ((__const char *__file, struct statfs *__buf));
```

Die Unit *Libc.pas* enthält die Übersetzung, in der die in Kapitel 5.1.4 erläuterten Regeln demonstriert werden: Der Record darf nicht als *packed* deklariert werden (die Unit wird mit *{$ALIGN 4}* übersetzt) und die einzelnen Record-Felder werden der Reihe nach übersetzt wie einzelne Variablen. Dabei lässt sich der kryptische Typ *__fsblkcnt_t* auf den (kaum weniger kryptisch anmutenden) Typen *__u_long* zurückführen, der jedoch aus der Tabelle von Kapitel 5.1.4 als äquivalent zum Object-Pascal-Typ *LongWord* bekannt ist. Damit wir uns in Object Pascal keine Gedanken um Zeiger machen müssen, ist der Record-Parameter der Funktion *statfs* als *var*-Parameter deklariert:

```
// Translated from bits/statfs.h

type
  __fsblkcnt_t = __u_long;
  _statfs = {packed} record
    f_type: Integer;
    f_bsize: Integer;
    f_blocks: __fsblkcnt_t;
    f_bfree: __fsblkcnt_t;
    f_bavail: __fsblkcnt_t;
    f_files: __fsfilcnt_t;
    f_ffree: __fsfilcnt_t;
    f_fsid: __fsid_t;
    f_namelen: Integer;
    f_spare: packed array[0..6-1] of Integer;
  end;
  TStatFs = _statfs;
  PStatFs = ^TStatFs;

// Translated from sys/statfs.h

{ Return information about the filesystem on which FILE resides.  }
function statfs(__file: PChar; var __buf: TStatFs): Integer; cdecl;
```

(Den Kommentar hat Borland praktischerweise auch aus den Header-Dateien übernommen, so dass die *Libc.pas* gar keine so schwierige Lektüre darstellt ...)

Anzeige der verfügbaren Plattenkapazität *R69*

Als ersten Parameter erwartet die Funktion *statfs* den Namen einer Datei, die sich in dem Dateisystem findet, das untersucht werden soll. Hier bietet es sich natürlich an, einfach das Wurzelverzeichnis »/« anzugeben, denn das Programm soll ja nicht voraussetzen, dass außer dem Wurzelverzeichnis noch irgendwelche anderen Dateien vorhanden sind.

Für den freien Speicher der Festplatte gibt es im Ergebnis-Record von *statfs* zwei Felder mit unterschiedlicher Bedeutung:

- *f_bfree* gibt den insgesamt freien Speicher in Blöcken an
- *f_bavail* gibt den Speicher an, der dem aktuellen Benutzer zur Verfügung steht. Falls auf dem System Disk-Quotas eingerichtet sind, ist das weniger als der insgesamt freie Speicher.

Die Blöcke haben im Standard-Linux-Dateisystem *Ext2fs* eine Größe von 1024 Bytes. Das Beispielprogramm funktioniert nur, wenn sich das Wurzelverzeichnisses in einem solchen Dateisystem befindet. Praktischerweise liefert *statfs* im Record-Feld *f_type* gleich auch eine Kennzahl für das Dateisystem mit, die im Programm abgefragt werden kann, um mögliche fehlerhafte Speicherplatzberechnungen zu vermeiden:

```
const
  TypeExt2fs = $EF53;

function TForm1.GetFreeSpace: Real;
var
  fs_info: Tstatfs;
begin
  if RadioGroupDatenQuelle.ItemIndex = 0 then begin
    // Ermittlung des freien Hauptspeichers, siehe unten
  end else begin
    statfs('/', fs_info);
    if fs_info.f_type = TypeExt2fs then
      Result := fs_info.f_bavail * 1024
    else Result := 0; // siehe Fehlermeldung bei Programmstart
  end;
end;
```

Informationen über den freien Speicher

Nun muss noch der im obigen Listing offen gelassene Fall, dass in der Radiogruppe der erste Schalter gewählt ist, gefüllt werden. Dann soll nämlich statt des Festplattenspeichers der Hauptspeicher untersucht werden. Die Vorgehensweise ist hier sehr ähnlich zum bisher behandelten Fall. Die *Libc*-Funktion, die die gewünschten Informationen liefert, heißt *sysinfo*. Sie liefert ebenfalls eine Datenstruktur, diesmal vom Typ *TSysInfo*. Auch *TSysInfo* beinhaltet viele verschiedene Informationen, jedoch interessieren wir uns hier nur für eine einzige davon: *freeram*.

```
var
  S: TSysInfo;
...
  sysinfo(S);
  Result := S.freeram;
```

MiniHistoryGraph

Zum Schluss noch eine kleine Bemerkung zu *TMiniHistoryGraph*. Die Komponente wurde mit einer extrem einfach zu verwendenden Schnittstelle entworfen, von der nur drei Elemente im Beispielprogramm verwendet werden.

- Zunächst wird die Speicherkapazität der Komponente über das Property *HistoryLength* eingestellt. Die darin angegebene Zahl entspricht der Zahl der Änderungsbalken, die von der Komponente angezeigt werden können.
- Wenn nun eine neue Änderung protokolliert werden soll, wird *MiniHistoryGraph.NewValue(Change)* aufgerufen. Der Wert *Change* wird dadurch rechts als Balken dargestellt, alle anderen Werte rücken einen Schritt nach links, der erste fällt ganz heraus.
- Soll die Grafik gelöscht werden (im Beispielprogramm ist das der Fall, wenn Sie zwischen Festplattenspeicher und Hauptspeicher umschalten), wird dies mit der Methode *MiniHistoryGraph.Clear* erledigt.

5.1.6 X Window und die XLib

Unabhängig von verwendeten Desktops oder Window Managern bildet X Window die Grundlage der grafischen Oberfläche unter Linux. Da ein Kylix-Entwickler bereits von den darüber liegenden Bibliotheken Qt und CLX sehr verwöhnt wird, ist es nur in seltenen Fällen nötig, von X Window Gebrauch zu machen. Und selbst wenn er es könnte, wird er wohl eher auf direkte Aufrufe von X Window verzichten, denn diese Aufrufe sind nicht portabel – im Gegensatz zu den Aufrufen der Qt und der CLX. Da X Window jedoch einen ganz zentralen Bestandteil der Umgebung einer Kylix-Anwendung darstellt, soll es hier nicht ignoriert werden.

Bildschirme und Fenster

Besonders die netzwerkfähige Architektur von X Window hat eine Würdigung verdient, denn durch diese »erbt« jede Kylix-Anwendung die Möglichkeit, ihre Benutzeroberfläche auf einem entfernten Display-Server anzuzeigen, ohne dass der Entwickler der Kylix-Anwendung dafür auch nur einen Finger krümmen muss. Ein Display-Server« ist ein Rechner, der einer Anwendung, die auf einem anderen Rechner laufen kann, Bildschirm, Tastatur und Maus zur Verfügung steht (die Summe aus einem oder mehreren Bildschirmen, der Tastatur und der Maus wird in X Window als *Display* bezeichnet).

Damit eine X-Anwendung nicht den Display des aktuellen Rechners verwendet, können Sie entweder die *DISPLAY*-Umgebungsvariable setzen oder beim Start der Anwendung einen Kommandozeilenparameter angeben. Bereits das Qt-Toolkit enthält Code zur Auswertung der Kommandozeilenparameter, so dass etwa der Aufruf

```
MiniSysMon -display Itea:0
```

die Anwendung auf dem Bildschirm 0 des Rechners namens *Itea* erscheinen lässt. Sie können in einer Kylix-Anwendung wie folgt auf einige Informationen zum verwendeten Display zugreifen:

```
Xlib.PDisplay(Application.Display)^.default_screen, // Standard-Bildschirm
Xlib.PDisplay(Application.Display)^.nscreens // Zahl der Bildschirme
Xlib.PDisplay(Application.Display)^.display_name;
```

Auf einem alleine arbeitenden Desktop-PC (der sowohl die Anwendung ausführt als auch den Display-Server bereitstellt) ist die Zahl der Bildschirme meistens »1«, der Standard-Bildschirm trägt die Nummer »0« und der *display_name* lautet »0:0«.

Die Umwandlung von *Application.Display* in den Typ *Xlib.PDisplay* ist übrigens erforderlich, weil der Typ *PDisplay* in der Unit von *TApplication* nur als *Pointer* deklariert und somit nicht praktisch verwendbar ist.

Direkte Xlib-Aufrufe

Was die Unit *Libc* für die Kernel-Funktionalität ist, ist die *Xlib*-Unit für die X Window-Bibliothek. Hier finden Sie die in Object Pascal übersetzten Deklarationen der Funktionen, Typen, Strukturen und Konstanten der XLib, die auch wieder in C geschrieben ist. Auch die Struktur *Display*, auf die im obigen Beispiel über einen *PDisplay*-Zeiger zugegriffen wurde, ist in der *Xlib*-Unit deklariert.

Theoretisch können Sie in Kylix ganz spartanische X Window-Anwendungen schreiben, wie sie unter Unix schon seit über 10 Jahren in C geschrieben werden, ohne Qt und ohne CLX. Die in der X Window-Literatur beschriebenen Aufrufe, die dafür notwendig sind, sollten alle in *Xlib.pas* nach Object Pascal übersetzt worden sein. Die spezielle Schwierigkeit, die es bei direkten Xlib-Aufrufen aus einer »normalen« Kylix-Anwendung (einer Anwendung, die auch Qt und CLX verwendet) zu bewältigen gilt, besteht darin, an die intern von der Qt verwendeten X Window-Handles zu gelangen.

Ein Beispiel wäre etwa, dass die Funktion *XSetInputFocus* einem bestimmten CLX-Formular den Tastaturfokus geben soll. Wenn Sie das Formular einfach mit *Form.Show* anzeigen, beeinflusst das bei vielen Window-Managern nämlich nicht den Tastaturfokus, so dass der Fokus danach eventuell in einem verdeckten Fenster steckt. Nehmen wir an, dass wir versucht haben, den Fokus mit der Methode *Form.SetFocus* doch noch auf das Formular zu bewegen, und dass dies nicht funktioniert hat. (Es funktioniert tatsächlich nicht, wie Sie im Beispielprogramm *XWindowCall* ausprobieren können!)

Mit der Funktion *XSetInputFocus* kann das scheinbare Problem gelöst werden, allerdings benötigt die Funktion zwei Zutaten: Einen Zeiger auf den zu verwendenden X Window-Display und ein Handle auf das zu fokussierende Fenster. Ersteren erhalten wir einfach von der bereits bekannten Variablen *Application.Display*.

Das Window-Handle ist etwas schwieriger zu beschaffen, denn es wird von Qt verwaltet und diese wiederum ist unter der CLX »versteckt«. Wir müssen uns daher zuerst an die CLX wenden, um Zugriff auf die Qt zu erhalten. Das *Handle*-Property aller *TWidgetControl*-Komponenten und damit auch der Formulare stellt eine Referenz auf ein *QWidget* dar. Von diesem *QWidget* können wir nun über die Funktion *winid* die interne X Window-Datenstruktur erhalten:

```
XSetInputFocus(Application.Display,
   QWidget_winid(Screen.ActiveForm.Handle), RevertToNone, CurrentTime);
```

RevertToNone und *CurrentTime* sind vordefinierte Konstanten, deren Bedeutung Sie bei weiterem Interesse aus der Online-Dokumentation (*man XSetInputFocus*) erfahren können.

5.1.7 Shared Libraries mit Kylix

Bisher haben wir andere Funktionen aufgerufen, die unseren Kylix-Programmen durch Shared Libraries zur Verfügung gestellt wurden. Es ist jedoch auch umgekehrt möglich, eine solche Shared Library in Kylix zu erzeugen und diese dann in einem externen Programm, das in C, C++ oder einer anderen Sprache (oder sogar wieder mit Kylix) erzeugt wurde, aufzurufen.

Library-Module und Exports-Klauseln

Hierzu benötigen Sie ein Modul, dass statt durch das Wort *program* wie in der Projektdatei einer normalen Anwendung durch das Wort *library* eröffnet wird. Das Gerüst einer solchen Datei erhalten Sie unter DATEI | NEU, wenn Sie dort das Symbol GEMEINSAMES OBJEKT auswählen.

In diesem Modul oder in den Units, die Sie darin verwenden, können Sie nun Prozeduren und Funktionen schreiben, die von externen Anwendungen genutzt werden können. Lediglich eine Zusatzarbeit ist notwendig, um eine Funktion auch tatsächlich zu exportieren: Ihr Name muss in einer *exports*-Klausel aufgeführt werden, wie im folgenden Beispiel:

```
library KylixPart;

uses
  SysUtils, Classes, ...;

function createPartWidget(Parent: QWidgetH): QWidgetH; cdecl;
begin
  ... Implementierung der Funktion ...
end;

exports
```

```
  createPartWidget; // Exportieren der Funktion

begin
end.
```

Auf diese Weise könnte eine Kylix-Bibliothek etwa ein Widget zur Verfügung stellen, das in den KDE-Konqueror eingebettet werden kann (hierzu bedarf es allerdings wieder einer C-Bibliothek als Schnittstelle zu den C++-Bibliotheken in KDE).

Hinweis: Das Programmieren von Formularen und Dialogen in einer Kylix-Bibliothek, um diese von externen Anwendungen aufzurufen, wie es in Delphi für Windows möglich ist, funktioniert in Kylix nicht, jedenfalls noch nicht in der ersten Version.

Bei der Übersetzung eines solchen *library*-Moduls erzeugt der Compiler eine Datei, deren Name per Voreinstellung in Linux-typischer Weise mit dem Präfix *lib* und mit der Endung *.so* versehen wird. Um diesen Namen zu verändern, etwa ihn um Versionsinformationen zu verlängern, können Sie dies entweder in den Compileroptionen auf der Seite ANWENDUNG oder direkt im Quelltext der Bibliothek über die Compilerdirektiven *$SONAME*, *$SOPREFIX*, *$SOSUFFIX* und *$SOVERSION* einstellen (siehe Online-Hilfe).

Importieren der Object-Pascal-Funktionen in C

Um eine in Object Pascal geschriebene Bibliothek nun beispielsweise in einem C-Programm verwenden zu können, müssen die Deklarationen der Funktionen und Prozeduren wieder übersetzt werden, wie wir es schon von C nach Object Pascal kennen, allerdings jetzt in umgekehrter Richtung. Die Funktion aus obigem Beispiel könnte etwa wie folgt übertragen werden (beachten Sie auch, dass sie in Object Pascal wieder als *cdecl* deklariert wurde!):

```
extern "C" {
  QWidget *createPartWidget(QWidget *parent);
}
```

Beim Linken der C-Anwendung benötigt jetzt noch der Linker Zugriff auf die kompilierte Object-Pascal-Bibliothek. Den direktesten Weg hierzu stellt die Kommandozeilenoption »*-l*« dar, etwa im Falle der oben skizzierten Bibliothek *DelphiPart*:

```
g++ [...andere Parameter zum Linken der Anwendung...] -Ldir -lKylixPart
```

Dabei muss *dir* noch durch das Verzeichnis ersetzt werden, in dem der Linker die kompilierte Object-Pascal-Bibliothek finden kann (angenommen, sie befindet sich nicht in den sowieso schon durchsuchten Verzeichnissen). Die Direktive *-lKylixPart* veranlasst den Linker, nach der Datei *libKylixPart.so* zu suchen.

Re-Importieren in Kylix

Das Verwenden einer in Object Pascal geschriebenen Bibliothek in einer Object-Pascal-Anwendung verläuft grundsätzlich nach den gleichen Regeln wie das Importieren einer C-Bibliothek. So müssen also auch die externen Object-Pascal-Funktionen in der Anwendung als *external* deklariert werden. Die Übereinstimmung von Aufrufformat und Parametern ist leicht herzustellen, da Bibliothek und Anwendung einfach die gleichen Funktionsdeklarationen verwenden können (in der Bibliothek mit Implementierung der Funktionen im *implementation*-Teil der Unit, in der Anwendung implementiert durch die *external*-Direktive).

5.2 Konsolenanwendungen

Im Gegensatz zu anderen bekannten Betriebssystemen hat man sich unter Linux noch wenig Mühe gemacht, die so genannten Kommandozeilen zu verstecken, sei es in Form des Textbildschirms, an dem Sie sich anmelden können (wenn Sie nicht die grafische Anmeldevariante mit automatisch gestarteter X Window-Oberfläche gewählt haben), sei es in Form der Terminalfenster, die Sie unter den bekannten Desktops wie Gnome und KDE mit Schaltern am unteren Rand des Fensters öffnen können. Selbst wenn Sie eine grafische Anmeldevariante gewählt und kein Terminalfenster geöffnet haben, können Sie jederzeit mit Strg+Alt+F1-F6 zu sechs parallel arbeitenden Terminals schalten, für die der Bildschirm jeweils wieder in den Textmodus zurückgeschaltet wird[1].

Eine *Konsolenanwendung* ist ein Programm, das Text auf einem solchen Textbildschirm oder in einem Terminalfenster ausgibt oder/und von dort seine Tastatureingaben bezieht. Sie können mit Kylix unter Linux wie mit Delphi unter Windows eine Konsolenanwendung einfach dadurch erzeugen, indem Sie DATEI|NEU... und dann das Icon KONSOLEN-ANWENDUNG wählen. Ein triviales Programm dieses Typs sieht beispielsweise so aus:

```
program MiniKonsolenAnwendung;

{$APPTYPE CONSOLE}

begin
  writeln('Hello world!'); // Ausgabe einer Textzeile auf der Konsole
  writeln('Programmende mit <Eingabe>');
  readln; // Einlesen einer Textzeile (ohne Speicherung in Variablen)
end;
```

[1] Rückschalten zu X Window mit Strg+Alt+F7

Hier kommen die klassischen *write(ln)*- und *read(ln)*-Anweisungen zum Einsatz, wie sie schon vor fast 20 Jahren in dem Urvater von Kylix, in Turbo Pascal, verwendet wurden.

Während die Konsolenanwendung unter Windows jedoch wirklich eine eigene Art von Anwendung darstellt, ist im Prinzip unter Linux jede Anwendung eine Konsolenanwendung. Die Compiler-Anweisung *$APPTYPE CONSOLE* hat unter Linux keinen Effekt und auch ein grafisches X Window-Progamm kann Text auf eine Konsole ausgeben. Eine *writeln*-Anweisung in einer normalen Kylix-Fensteranwendung führt beispielsweise dazu, dass der Text

- in der Konsole ausgegeben wird, aus der die Anwendung gestartet wurde
- falls das Kylix-Programm aus der IDE gestartet wurde, in der Konsole, aus der die IDE gestartet wurde
- falls die IDE über den Desktop gestartet wurde (z. B. über das Panel-Menü), in der Konsole, aus der X Window gestartet wurde, normalerweise die Konsole, die mit Alt + Strg + F1 zu erreichen ist.

Wie diese Auflistung andeutet, gibt es unter Linux immer irgendeine Konsole, die zumindest entfernt als »Aufrufer« oder »Elternkonsole« einer grafischen Anwendung verstanden werden kann. Folglich brauchen Sie unter Kylix keine besonderen Einstellungen vorzunehmen, um eine Konsolenanwendung zu schreiben. Das Einzige, worin sich die automatisch erzeugte neue Konsolenanwendung von einer normalen Kylix-Anwendung unterscheidet, ist, dass Kylix dafür kein Formular in das Konsolenprojekt einbindet und keine CLX-Units in der *uses*-Zeile auflistet.

Um einer möglichen Zweckentfremdung eines genialen GUI-Entwicklungs-Tools für Textbildschirmanwendung nicht unnötig Vorschub zu leisten, wird sich dieses Buch nicht weiter mit dem Erzeugen von Konsolenanwendungen befassen, wohl aber mit der Verwendung der reichlich vorhandenen und unbestreitbar nützlichen Konsolenanwendungen in einer grafischen Kylix-Anwendung. Dieser kleine Abschnitt diente insofern als Vorbereitung auf das nächste Kapitel und zur Erläuterung der systembedingten »Verwandtschaft« zwischen einer Kylix-Anwendung und einer Kommandozeilen-Anwendung.

Wenn Sie sich näher für Konsolenanwendungen in Kylix interessieren (und eigentlich sprechen keine grundsätzlichen Gründe gegen eine solche »Zweckentfremdung«), sei Ihnen die Online-Hilfe zu *write(ln)*, *read(ln)* und *ExitCode* bzw. *halt* empfohlen (über einen Exit-Code können Sie einem eventuell aufrufenden Shell-Skript einen Rückgabewert übergeben, der z. B. über Erfolg oder Misserfolg der ausgeführten Tätigkeiten Aufschluss gibt).

5.3 Prozesse und Shells

Während Prozesse in jedem Multitasking-Betriebssystem eine wichtige Rolle spielen, haben die Shells vor allem im Unix-Bereich eine besondere Bedeutung, die ihnen etwa in der Windows-Umgebung von Delphi 5 oder Delphi 6 nicht zukommt. Dieser Abschnitt erläutert die grundlegenden Möglichkeiten in Linux, andere Programme zu starten (d.h. neue Prozesse zu erzeugen), sowie einige darauf aufbauende Möglichkeiten zur Verwendung der Betriebssystem-Shell in Kylix-Anwendungen.

Ein *Prozess* ist vereinfacht gesagt einfach ein gerade ablaufendes Programm, für das das System eine eigene Sammlung von Ressourcen und Daten verwaltet. Zu den Ressourcen eines Prozesses gehören der benutzte Speicher und die geöffneten Dateien[2]. Zu den weiteren prozessbezogenen Daten gehören z.B. die Zugriffsrechte, welche von dem Benutzer abhängen, der das Programm gestartet hat, und das aktuelle Verzeichnis. Wichtig ist natürlich auch die aktuelle Ausführungsposition, damit das Betriebssystem auch auf Rechnern mit nur einer CPU beliebig zwischen verschiedenen Prozessen wechseln kann. Wird dasselbe Programm mehrfach gestartet, läuft es in verschiedenen Prozessen. Aufgrund der getrennten Verwaltung durch das System kann jeder dieser Prozesse eigene Speicherreservierungen durchführen und eigene Dateien ffnen.

Wenn *innerhalb* eines nur einmal ablaufenden Programms mehrere gleichzeitig ablaufende Codeabschnitte vorliegen, dann spricht man statt von Prozessen von *Threads*, welche in Kapitel 4.8 behandelt werden. Solche Threads können unter Linux als Spezialfall von Prozessen dargestellt werden, da Prozesse auch Ressourcen teilen können und jeder Thread intern wie ein Prozess verwaltet wird, der alle Ressourcen mit dem Haupt-Thread seines Programms teilt. Das Teilen von Ressourcen sowie komplexere Strukturen des Prozessmodells von Linux (z.B. Prozessgruppen und Sessions) wird in spezieller Literatur zur Linux-Programmierung genauer behandelt und soll nicht Thema dieses Kapitels sein.

5.3.1 Aufrufen anderer Programme und Tools

Das Starten eines Programms ist einer der zentralen Dienste des Betriebssystems und wird von allen Systemen, unter denen es zurzeit »Delphi-kompatible« Tools von Borland gibt, über relativ einfache Funktionsaufrufe angeboten. Ein Linux-Programm geht blicherweise in zwei Schritten vor, um ein anderes Programm zu starten:

- Zuerst wird ein neuer Prozess erzeugt,
- in diesem wird dann das neue Programm geladen.

Ressourcen der grafischen Oberfläche werden nicht vom Kernel, sondern vom X-Server oder dem darauf laufenden Toolkit verwaltet und zählen hier nicht zu den Daten eines Prozesses.

Beide Schritte sind auf den jeweils anderen Schritt angewiesen: Die Funktion zum Erzeugen eines neuen Prozesses (*fork*) erzeugt lediglich eine Kopie des aktuellen Prozesses. *fork* verdoppelt also zunächst einmal eine laufende Anwendung. Im zweiten Schritt wird dann ein neues Programm in die Prozesskopie geladen, die Kopie des aufrufenden Programms wird dadurch ersetzt und das neu geladene Programm kann ablaufen. Wäre nun vorher nicht *fork* aufgerufen worden, hätte das neu geladene Programm das aufrufende Programm ersetzt und das aufrufende Programm wäre beendet.

Prozesse starten

Der Aufruf von *fork* könnte zunächst nicht einfacher sein, da er keine Parameter erwartet. Doch dieser Eindruck täuscht, denn durch *fork* wird ja der aktuelle Prozess verdoppelt und die Kopie des Prozesses befindet sich an der gleichen Ausführungsposition. Der nach *fork* folgende Code wird also zweimal ausgeführt: einmal im Original-Prozess und einmal in der Kopie. Es bedarf daher eines Weges, die beiden Prozesse zu unterscheiden, so dass in dem einen das neue Programm geladen wird, während der andere normal weiterläuft. Diese Unterscheidung geschieht anhand des Rückgabewerts von *fork*:

```
var
  pid: Integer;
begin
  pid := fork;
  if pid = 0 then begin
    writeln('Prozess-Kopie läuft.');
    libc.execlp('xeyes', 'xeyes', nil);
    writeln('Fehler beim Starten des Programms.');
    libc._exit(255);
  end else begin
    writeln('Dies ist eine Meldung vom Original-Prozess.');
    if pid = -1 then ShowMessage('Fehler bei fork');
  end;
```

Wenn das Ergebnis von *fork* gleich Null ist, dann kann der Code davon ausgehen, dass er in der Kopie des Prozesses läuft, die ja durch das neue Programm ersetzt werden soll. Daher wird oben mit *execlp* ein solches Programm geladen. Da dieses Programm die Kopie des Prozesses überschreibt, wird bei Erfolg auch der nach *execlp* folgende *writeln*-Aufruf überschrieben und niemals ausgeführt. Nur, wenn *execlp* das Programm nicht laden kann, kommen die folgenden Programmzeilen zur Ausführung. Im Beispiel wird eine Fehlermeldung auf der Konsole ausgegeben und die unnütz gewordene Prozesskopie beendet.

Hinweis: Versuchen Sie nicht, in der Prozess-Kopie die CLX aufzurufen. Da bereits ein *ShowMessage*-Aufruf nicht richtig funktioniert, werden die ersten beiden Meldungen im obigen Beispiel mit *writeln* auf der Konsole ausgegeben. Wie Sie einen Fehler bei *exec...* über einen Exit-Code an den Hauptprozess weitergeben und dort durch eine Meldung dem Benutzer anzeigen können, zeigt Kapitel 5.3.2.

Auch das Beenden des Prozesses mit *libc._exit* ist wichtig: Dies beendet wirklich nur die Kopie des Prozesses, während ein Aufruf der Delphi-Funktion *Halt* auch den Ursprungsprozess beenden würde.

Der Fall *pid <> 0* tritt nur bei dem Prozess ein, von dem der Aufruf von *fork* ausgegangen ist. Im Erfolgsfall erhält er eine Kennzahl für den gestarteten Prozess, im unwahrscheinlichen Fall mangelnden Speichers trägt *pid* den Wert »-1« und die genauere Fehlerursache lässt sich mit *errno* ermitteln.

Programme ausführen

Zum Ausführen von Programmen gibt es in der *Libc* eine ganze Familie von Funktionen, deren Namen mit *exec* beginnen und die dreierlei Parameter haben können:

- eine Angabe des zu startenden Programms; bei den Varianten *execvp* und *execlp* genügt auch die Angabe des Dateinamens ohne Pfad, es wird dann automatisch der im System eingestellte Pfad nach der Datei abgesucht;
- eine Liste der an das Programm zu übergebenden Parameter,
- eine Liste der für das Programm gültigenden Umgebungsvariablen.

Die im obigen Beispiel verwendete Funktion *execlp* verzichtet auf die Umgebungsvariablen und kopiert einfach die Umgebung des aktuellen Prozesses. Die Parameter für das neue Programm werden in Form einer Liste variabler Länge, bestehend aus *PChar*-Zeigern, übergeben. Der erste Parameter wird jedoch nicht wie ein normaler Parameter ausgewertet, sondern enthält per Konvention den Namen der ausführbaren Datei ohne Pfad, stimmt also im obigen Beispiel genau mit dem zu startenden Programm überein[3]. Das letzte Element der Liste muss ein *nil*-Zeiger sein.

Die *exec*-Varianten mit dem »v« im Namen erwarten die Parameter nicht wie die *execl*-Varianten als Folge von Einzelparametern, sondern in einem Array von *PChar*-Zeigern, dessen Ende durch ein *nil* gekennzeichnet ist. Wenn Sie also beispielsweise einen (normalen) Parameter an ein Programm übergeben wollen, benötigen Sie für *execv* das folgende Array:

Unter Unix kann dieselbe ausführbare Datei unter mehreren Namen gestartet werden, beispielsweise, wenn ein symbolischer Link dieses Namens auf die Datei gesetzt ist. Die Verhaltensweise des Programms richtet sich dann nach dem im ersten Parameter übergebenen Programmnamen.

```
var
  Params: array[0..2] of PChar;
begin
  Params[0] := Programmname;
  Params[1] := '-option'; // übergebener Parameter
  Params[2] := nil;
```

In diesem Fall wäre es natürlich einfacher, wie im vorherigen Beispiel die Variante *execl* zu verwenden, der Sie die drei *PChars* direkt übergeben können. Wenn sich die genaue Anzahl der Parameter allerdings erst später herausstellt wie in der Beispielroutine von Kapitel 5.3.2, ist eine der »p«-Varianten (*execvp* oder *execlp*) praktischer zu verwenden.

Die Ausgabe anderer Programme nutzen *R71*

Eine der Besonderheiten von Unix im Vergleich etwa zu den Windows-Betriebssystemen von Microsoft ist die Flexibilität, mit der man in der Shell-Befehlszeile oder in Skript die Funktionen mehrerer Programme kombinieren kann (Windows erlaubt zwar die gleichen Kombinationstechniken, bringt aber im Gegensatz zu Linux nur ganz wenige als Bausteine verwendbare Konsolenanwendungen mit).

So können Sie z. B. die Ausgabe des Textsuchprogramms *grep* an das Zählprogramm *wc* schicken, um die Zahl der Vorkommnisse des Textes zu zählen, oder an *lpr*, um die gefundenen Textstellen auszudrucken. Wenn Sie eine solche Verknüpfung von Programmen über die Kommandozeile vornehmen, z. B. mit

```
grep "exec" *.pas |wc
```

öffnet die Shell intern eine *Pipe* genannte Datenstruktur, die gleichzeitig dem ersten Kommando (hier *grep*) als Ausgabedatei, dem zweiten (*wc*) als Eingabedatei dient.

Grundlage dieser Programmverknüpfung stellt damit das Umleiten von Konsolenein- und -ausgabe von Programmen dar, was wir hier mit Kylix am Beispiel der Ausgabe ausprobieren wollen: Ein Kylix-Programm soll das Programm *ls* starten um die Dateien eines Verzeichnisses aufzulisten. Die Ausgabe von *ls* soll aber nicht auf irgendeiner Konsole erscheinen, sondern vom Kylix-Programm abgefangen und weiterverarbeitet werden. Dies führt im Endeffekt dazu, dass das externe Programm *ls* in der Kylix-Anwendung wie eine interne Object-Pascal-Funktion ausgeführt werden kann.

Auch für dieses Unterfangen sind wieder Aufrufe der *Libc* erforderlich, denn Standardein- und -ausgabe werden über die Dateideskriptoren 0 (Standardeingabe), 1 (Standardausgabe) und 2 (Standard-Fehlerausgabe) festgelegt. Normalerweise sind diese mit der Konsole verbunden. Um sie mit einer Datei zu verbinden, können Sie die Funktion *Libc.dup2* verwenden, die den Deskriptor einer geöffneten Datei auf einen beliebigen anderen Deskriptor überträgt.

Im folgenden Beispiel wird zunächst mit *libc.open* eine Datei geöffnet. Der dritte Parameter gibt wie bei der in Kapitel 5.1.2 beschriebenen Funktion *creat* die Zugriffsrechte an. Der zweite Parameter besagt Folgendes:

- *O_CREAT* bestimmt, dass eine neue Datei erzeugt wird,
- *O_WRONLY* besagt, dass nur Schreibzugriffe auf die Datei erfolgen sollen,
- *O_TRUNC* ist erforderlich, damit eine bestehende Datei überschrieben wird, auch wenn die neue Datei ohne Inhalt bleibt (Länge 0). Ohne *O_TRUNC* würde, falls die Ausgabedatei beispielsweise für Fehlermeldungen genutzt wird, immer eine Datei mit Fehlermeldungen vorhanden sein. Wenn bei der Ausführung des Programms einmal keine Fehlermeldungen entstehen, würde die Datei noch die zuletzt aufgetretenen Fehlermeldungen enthalten.

Diese Datei wird dann, im mit *fork* neu erzeugten Prozess, mit *dup2* als Standardausgabe für alle Meldungen – auch für die Fehler – festgelegt:

```
var
  fd, pid: Integer;
begin
  fd := libc.open(PChar('ProgramOutput.txt'),
                  O_CREAT or O_WRONLY or O_TRUNC,
                  S_IWUSR or S_IRUSR or S_IRGRP or S_IROTH);
  pid := fork;
  if pid = 0 then begin
    libc.dup2(fd, 1); // Standardausgabe
    libc.dup2(fd, 2); // Fehlerausgabe
    libc.__close(fd);
    exec...
  end;
```

Nach der Übertragung der Dateideskriptoren kann der ursprüngliche Deskriptor *fd* mit *__close* geschlossen werden.

Das nächste Kapitel gibt ein Beispiel, in dem die Ausgabe von *ls* auf diese Weise abgefangen wird.

5.3.2 Eine allgemeine Programmstart-Routine

An dieser Stelle sollen die im vorangegangenen Kapitel gewonnenen Erkenntnisse in einer allgemein verwendbaren Funktion zusammengefasst und in einem Beispielprogramm angewendet werden. Die Funktion *RunProgram* soll alle zum Starten eines externen Programms erforderlichen Aktionen zusammenfassen und auch optional die Umleitung der Ausgabe erlauben. Sie soll durch folgende Parameter gesteuert werden:

- *Command* enthält den Namen der auszuführenden Datei, optional mit voller Pfadangabe.
- *ArgStrs* ist ein offenes Array, das die an das Programm zu übergebenden Parameter aufnimmt. Ein solches offenes Array lässt sich ja in Object Pascal sehr leicht direkt beim Funktionsaufruf angeben. Leere Strings im Array werden nicht an das aufgerufene Programm weitergegeben.
- *CaptureOutput* gibt den Namen einer Datei an, in der die Ausgabe des Programms gespeichert werden soll. Geben Sie hier einen leeren String an, wenn die Ausgabe des Programms *nicht* gespeichert werden soll. Dieser leere String ist auch die Voreinstellung, wenn Sie *RunProgram* nur mit den Parametern *Command* und *ArgStrs* aufrufen.
- *Wait* ist optional und besagt, ob *RunProgram* auf das Beenden des gestarteten Programms warten soll (Voreinstellung ist »nein«). In diesem Fall liefert *RunProgram* den Exit-Code des Programms als Funktionsergebnis.
- *RunFromDir* gibt abschließend und optional ein Verzeichnis an, das dem aufzurufenden Programm als aktuelles Verzeichnis dienen soll.

Die vollständige Deklaration von *RunProgram* lautet:

```
function RunProgram(Command: String;
  ArgStrs: array of string; CaptureOutput: String = '';
  Wait: Boolean = False; RunFromDir: String = ''): Integer;
```

Beispielprogramm

Das Projekt *RunPrograms* (Abbildung 5.2) besteht aus einem browserartigen Fenster, in dessen rechtem Teil Informationen über das im Verzeichnisbaum gewählte Verzeichnis angezeigt werden. Alle diese Informationen werden über externe Programme ermittelt, und zwar mit Hilfe von *ls* und einem Shell-Skript, welches wiederum *grep*, *cat* und *sed* aufruft. Während die vom Shell-Skript stammende zweite Seite der Informationen in Kapitel 5.3.3 zur Sprache kommt, geht es hier um die *Dateiliste*, die per *ls* ermittelt wird.

Um gegenüber dem direkten Aufruf von *ls* in einer Konsole einen Vorteil zu haben, stellt das Programm einige Schalter bereit, mit dem der Programmbenutzer die Ausgabe von *ls* konfigurieren kann. Das erspart ihm immerhin, sich die Befehlszeilenoptionen *-A*, *-B*, *-l*, *-L* und zehn verschiedene Sortieroptionen zu merken.

Die entscheidende Methode des Programms ist *UpdateLs*, welche die in den Steuerelementen eingestellten Optionen in *ls*-Parameter umwandelt, *ls* ausführt und dessen Ausgabe im *Memo1* anzeigt. Damit *ls* auch das im Verzeichnisbaum gewählte Verzeichnis (*DTV.Directory*) auflistet, wird es in diesem Verzeichnis ausgeführt (letzter Parameter von *RunProgram*). Natürlich hätte das gewählte Verzeichnis auch als Parameter an *ls* übergeben werden können.

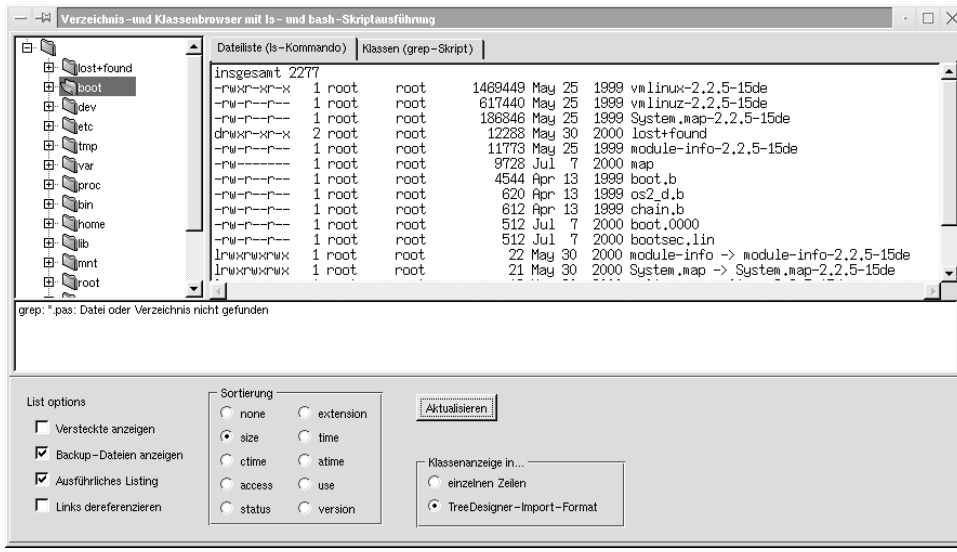


Abbildung 5.2: Die Ausgabe von ls im Fenster des Beispielprogramms

```
procedure TForm1.UpdateLs;
var
  i: integer;
  SortOption : string;
  MoreOptions: array[0..3] of string;
begin
  SortOption := '--sort='+rbSort.Items[rbSort.ItemIndex];
  for i := Low(MoreOptions) to High(MoreOptions) do
    MoreOptions[i] := '';
  if cbHidden.Checked then MoreOptions[0] := '-A';
  if not cbBackup.Checked then MoreOptions[1] := '-B';
  if cbLongFormat.Checked then MoreOptions[2] := '-l';
  if cbFollowLinks.Checked then MoreOptions[3] := '-L';
  // ls starten:
  if RunProgram('ls', [MoreOptions[0], MoreOptions[1],
    MoreOptions[2], MoreOptions[3], SortOption], fnLsOutput, True,
    DTV.Directory) <> 0
  then
    ShowMessage('ls gab einen Fehlercode zurück. Möglicherweise '+
                'wird ein anderes Aufrufformat verwendet.')
  else
    Memo1.Lines.LoadFromFile(fnLsOutput);
end;
```

Der Dateiname *fnLsOutput* wird bei Programmstart wie folgt als *ls.txt* im Verzeichnis der ausführbaren Datei der Anwendung initialisiert:

```
function FileFromAppDir(FileName: string): String;
begin
  Result := IncludeTrailingPathDelimiter(
    ExtractFilePath(Application.ExeName)) +
      FileName;
end;

initialization
  fnLsOutput := FileFromAppDir('ls.txt');
end.
```

Hinweis: Die Initialisierung von *fnLsOutput* findet bereits vor der *OnCreate*-Methode des Formulars statt, weil das Programm die in Kapitel 4.2.3 beschriebene *TStateSaver*-Komponente verwendet, um bei jedem Programmstart automatisch die letzten Einstellungen der Steuerelemente wiederherzustellen. *TStateSaver* verursacht auf diese Weise, dass die Methode *UpdateLs* bereits vor dem *OnCreate*-Event aufgerufen wird.

Die RunProgram-Funktion *R72*

Die Funktion *RunProgram* besteht zu einem großen Teil aus der Anwendung der in Kapitel 5.3.1 beschriebenen Grundlagen:

- Erzeugen einer Prozess-Kopie mit *fork*
- Ausführen eines Programms mit *execvp*
- Konstruktion eines *PChar*-Arrays für die Parameter des aufgerufenen Programms
- Umleitung der Ausgabe des aufgerufenen Programms (optional, wenn *CaptureOutput* nicht leer ist).

Außerdem wechselt *RunProgram* noch in das Verzeichnis, das in *RunFromDir* angegeben ist. Beachten Sie, dass der Verzeichniswechsel mit *chdir* im *if*-Teil für *pid = 0* stattfindet, also in der Kopie des aktuellen Prozesses. Der Verzeichniswechsel wirkt sich dadurch auch nur auf diese Kopie aus und beeinträchtigt den aufrufenden Prozess nicht.

Noch zu erläutern bleibt, wie die Funktion eventuelle Fehler beim Starten des Programms an den Aufrufer zurückgibt. In Kapitel 5.3.1 wurde für den Fall, dass die *exec*-Funktion scheitert, lediglich eine *writeln*-Ausgabe auf der Konsole vorgenommen. Das Listing von *RunProgram* zeigt eine elegantere Möglichkeit, den Fehler zu behandeln:

```
function RunProgram(Command: String;
  ArgStrs: array of string; CaptureOutput: String = '';
  Wait: Boolean = False; RunFromDir: String = ''): Integer;
var
  fd, pid, status, i: Integer;
```

```
    args: array of PChar;
  begin
    Result := 0;
    if CaptureOutput <> '' then
      fd := libc.open(PChar(CaptureOutput), O_CREAT or O_WRONLY or O_TRUNC,
        S_IWUSR or S_IRUSR or S_IRGRP or S_IROTH);
    pid := fork;
    if pid = 0 then begin
      if CaptureOutput <> '' then begin
        libc.dup2(fd, 1); // Standardeingabe
        libc.dup2(fd, 2); // Fehlerausgabe
        libc.__close(fd);
      end;
      if RunFromDir <> '' then
        chdir(RunFromDir);
      SetLength(Args, Length(ArgStrs)+2);
      Args[0] := PChar(Command);
      for i := 1 to Length(ArgStrs) do begin
        if ArgStrs[i-1] = '' then begin
          SetLength(Args, i+2);
          break;
        end;
        Args[i] := PChar(ArgStrs([i-1]));
      end;
      Args[Length(Args)-1] := nil;
      libc.execvp(PChar(Command), addr(Args[0]));
      libc._Exit(255);
    end else begin
      if Wait then begin
        libc.wait4(pid, integer(addr(status)), 0, nil);
        Result := status;
      end;
    end;
  end;
```

Exit-Codes und Libc.Wait4

Ein neuer Aspekt des Listings von *RunProgram* besteht darin, dass die Funktion, falls der Parameter *Wait* den Wert *True* hat, auf das Ende des gestarteten Programms wartet. Hierfür verwendet sie die Libc-Funktion *wait4*, deren erste beide Parameter hier von Bedeutung sind (die anderen beiden sind in *man wait4* dokumentiert):

- Zunächst gibt uns *wait4* endlich eine Gelegenheit, die von *fork* zurückgelieferte Prozess-ID sinnvoll zu verwenden, denn der erste Parameter von *wait4* ist die ID des Prozesses, dessen Ende abgewartet werden soll.
- Im zweiten Parameter wird die Adresse einer Integer-Variablen übergeben, die zur Rückgabe eines Exit-Codes verwendet wird. Per Konvention zeigt ein Exit-Code von Null die erfolgreiche Ausführung des Programms an. Dies ist auch der vorein-

gestellte Exit-Code, falls ein Programm normal endet. Falls ein anderer Exit-Code gewünscht ist, muss dieser vom Programm selbst bestimmt werden, beispielsweise über die Zahl, die es an die *_Exit*-Systemfunktion übergibt[4].

Ein Exit-Code ist auch eine einfache Möglichkeit, einen Fehler beim Starten des anderen Programms abzufangen. Wie schon in Kapitel 5.3.1 festgestellt wurde, kommen die auf eine *exec*-Funktion folgenden Anweisungen eines Programms nur dann zur Ausführung, wenn die *exec*-Funktion scheitert. *RunProgram* ruft an dieser Stelle die *_Exit*-Funktion auf, um dem aufrufenden Prozess einen Exit-Code von 255 zurückzuliefern. Der aufrufende Prozess ist nichts anderes als der Programmcode, der nur drei Zeilen weiter unten steht – der Code, der per *wait4* auf das Ende des aufgerufenen Prozesses wartet.

Falls also *execvp* das gewünschte Programm nicht starten kann, landet die 255 in der Variablen *status* und letztlich im *Result* von *RunProgram*. Im früheren Listing von *TForm1.UpdateLs* ist zu sehen, wie das Beispielprogramm den Rückgabewert von *RunProgram* interpretiert: Ist er Null, lädt es die durch *ls* erzeugte Textdatei in das Memo. Alle Rückgabewerte ungleich 0 werden als Fehler interpretiert und resultieren in der Anzeige einer Meldung durch *ShowMessage*.

Hinweis: *_Exit* sollte nur in Programmen aufgerufen werden, die nicht mit der CLX arbeiten. Dies trifft für die oben mit *fork* angelegte Prozess-Kopie zu. Normale Kylix-Anwendungen sollten beendet werden, indem das Hauptformular geschlossen oder im Falle eines schwer wiegenden Fehlers *Application.Terminate* aufgerufen wird.

5.3.3 Ausführen von Shell-Skripten

Eine noch höhere Flexibilität als beim Ausführen externer Programme können Sie durch Shell-Skripte erhalten, welche in der Komplexität von einer einfachen Liste von Anweisungen bis zu strukturierten Programmen vergleichbar mit einer Object-Pascal-Funktion reichen. Auch beim Start von Kylix ist ein seitenlanges Shell-Skript (`kylix/bin/kylixpath`) beteiligt, wenn es auch »nur« der korrekten Einstellung des Pfades dient.

Ein Skript zum Finden von Klassen *R70*

Das aus dem letzten Abschnitt bekannte Beispielprogramm *RunPrograms* verwendet ein kleines Skript, um alle Object-Pascal-Dateien eines bestimmten Verzeichnisses nach Klassendeklarationen zu durchsuchen und die Vererbungsbeziehungen herauszusu-

4. Den Exit-Code einer Kylix-Anwendung können Sie durch Zuweisung an die globale Variable *ExitCode* bestimmen.

chen (siehe Abbildung 5.3). Es sucht zunächst mit *grep* alle Zeilen, die den Text *class(...)* enthalten (mit beliebigen Zeichen zwischen den Klammern). Diese Zeilen werden in die Datei geschrieben, die dem Skript als erster Parameter übergeben wird. Dann versucht es, mit dem Streameditor *sed* exakt die Namen von Basisklasse und abgeleiteter Klasse herauszufinden und in einem weniger gut lesbaren Format ohne Leerzeichen in eine zweite Datei zu schreiben, deren Name ebenfalls in einem Parameter an das Skript bergeben wird.

Während die erste Datei sich leichter zum Lesen durch einen Menschen eignet, ist die zweite Datei für den Import in den TreeDesigner vorgesehen, der aus dem Datenstrom direkt eine Grafik erstellen kann.

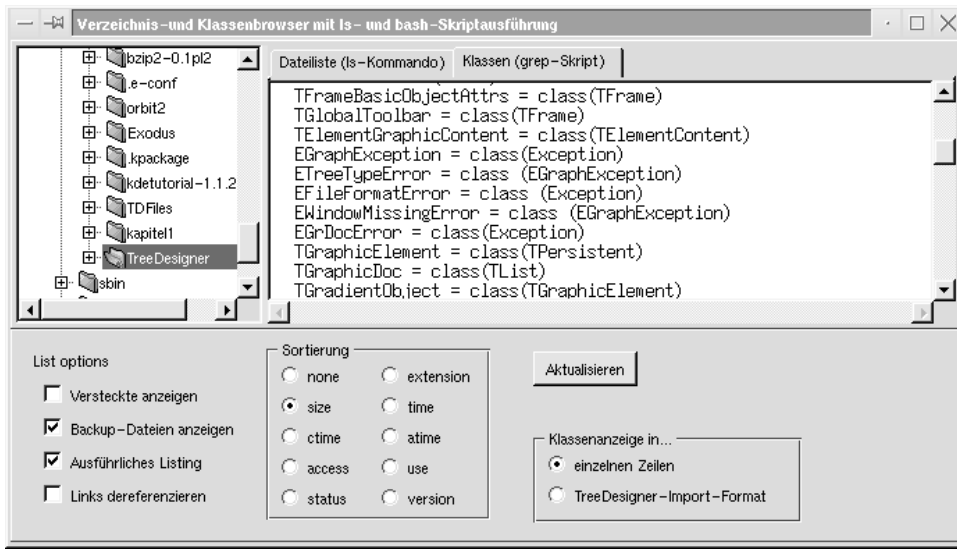


Abbildung 5.3: Die Liste der im aktuellen Verzeichnis gefundenen Klassen wird mit Hilfe eines Shell-Skripts hergestellt.

Ohne hier näher auf die Grundlagen der Skript-Programmierung eingehen zu können, sei im Folgenden das vollständige Skript als Anschauungsmaterial präsentiert, die ürzel *$1* und *$2* beziehen sich dabei auf die beiden Parameter:

```
#! /bin/sh
if test $# -ne 2; then
  echo "Es müssen genau zwei Parameter für die zu erzeugenden "
  echo "Dateien angegeben werden."
  exit 1
fi
grep -h class[[:space:]]*\(.*\) *.pas > $1
cat $1 | sed 's/[[:space:]]*\([[:alnum:]_]*\)[[:space:]]*=[[:space:]]*
```

```
            class[[:space:]]*([[:space:]]*\([[:alnum:]_]*\)
            [[:space:]]*.*/\1,\2;/' | tr -d '\012' > $2
exit 0
```

Notizen zu *sed:* Die vorletzte Zeile der Skript-Datei wurde im obigen Listing auf drei Zeilen aufgespalten. *Sed* wird darin mit einem String-Parameter (eingeschlossen in Hochkommata) aufgerufen. Der String definiert eine Suchen- und Ersetzen-Aktion (nach dem Muster '*s/regexp/replacement*', siehe *man*-page), bei der die beiden Klassennamen durch zwei Gruppen des Suchmusters identifiziert (»ge-*match*ed«) werden. Die Klassennamen werden am Schluss durch die Kürzel *\1* und *\2* in den Ergebnisstring eingesetzt. Der reguläre Ausdruck *regexp* für das Suchmuster lässt sich vielleicht wie folgt veranschaulichen:

```
LZ Klassenname1 LZ = LZ class LZ ( LZ Klassenname2 LZ Restzeile
   TStringList   =   class   (   TStrings      );
```

Die Klassennamen werden im Suchmuster durch eine Folge von alphanumerischen Zeichen plus dem optional verwendbaren Unterstrich angegeben und durch Klammerung zu den beiden Gruppen *\1* und *\2* gemacht. Falls die regulären Ausdrücke auf Ihrem System nicht auf der *man*-Seiten von *sed* dokumentiert sind, sollten Sie stattdessen bei *grep* weitergehende Informationen finden.

Die von *sed* demonstrierten Fähigkeiten zur Bearbeitung regulärer Ausdrücke stehen übrigens auch als *Libc*-Funktionen zur Verfügung. Drei Funktionen sind für eine Anwendung erforderlich: *regcomp*, *regexec* und *regfree*.

Da das Klassendaten-Import-Format des TreeDesigners aus historischen Gründen keine Zeilenumbrüche enthält, werden im oben gezeigten Skript vor der Erzeugung der endgültigen Ausgabedatei *$2* noch alle Zeilenumbrüche entfernt – hierfür dient das GNU-Utility *tr*, welches die Zeilenumbrüche am Code '\012' erkennt.

Die Skript-Ausführung

Zur Ausführung eines Shell-Skripts braucht man natürlich eine Shell. Da Shells zunächst als ganz normale Programme angesehen werden können, lassen sie sich auch genauso starten wie im letzten Kapitel etwa das *ls*-Kommando. Das Beispielprogramm setzt das Vorhandensein der Standard-Linux-Shell *bash* voraus. Der erste Parameter wird von *bash* als auszuführende Skript-Datei interpretiert, alle weiteren Parameter werden an dieses Skript weitergegeben (*bash* lässt sich auch durch eine Vielzahl weiterer Optionen steuern, die in *man bash* dokumentiert sind). *FnClassLines* bezeichnet im Folgenden die erste der vom Skript erzeugten Dateien, *fnClassList* die zweite:

```
// Initialisierung der Strings,
// zu FileFromAppDir siehe Listing in Kapitel 5.3.2:
   fnScriptErrors := FileFromAppDir('clerr.txt');
   fnClassLines := FileFromAppDir('originalclasslines');
   fnClassList := FileFromAppDir('classlist');
```

```
procedure TForm1.UpdateGrep;
var
  Command: String;
  BashResult : Integer;
begin
  Screen.Cursor := crHourglass; // Vorgang kann etwas länger dauern
  try
    Command := FileFromAppDir('makecurrentdirclasslist');
    BashResult := RunProgram('bash', [Command, fnClassLines, fnClassList],
                             fnScriptErrors, True, DTV.Directory);
    UpdateClassView;
    ErrorOutputMemo.Lines.LoadFromFile(fnScriptErrors);
    ErrorOutputMemo.Visible := (ErrorOutputMemo.Lines.Count > 0);
    if BashResult <> 0 then begin
      Screen.Cursor := crDefault;
      ShowMessage('Bash gab einen Fehlercode zurück. '+
                  'Möglicherweise fehlt die Skript-Datei.')
    end;
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TForm1.UpdateClassView;
begin
  // Abfrage der Radiogruppe zur "Klassenanzeige"
  if rgClassViewMode.ItemIndex = 1 then begin
    if FileExists(fnClassList) then
      Memo1.Lines.LoadFromFile(fnClassList)
    else Memo1.Lines.Clear;
  end else
    if FileExists(fnClassLines) then
      Memo1.Lines.LoadFromFile(fnClassLines)
    else Memo1.Lines.Clear;
end;
```

Gegenüber der direkten Ausführung von *ls* im vorherigen Beispiel ist die Auswertung der Skriptausführung etwas komplizierter, denn Fehler können auf zwei Ebenen auftreten:

- Bei der Ausführung des Skripts selbst kann beispielsweise *grep* eine Fehlermeldung ausgeben, weil es keine Dateien gefunden hat, oder die Shell kann melden, dass ein im Skript genanntes Programm nicht gefunden werden konnte. Diese Fehlermeldungen erscheinen auf der Standard-Fehlerausgabe, die ja von *RunProgram* auf eine im Parameter angegebene Datei umgeleitet werden kann. Im obigen Beispiel ist das der Dateiname *fnScriptErrors*. Diese Datei wird – zusätzlich zu den vom Skript produzierten Dateien mit den Klassennamen, die in *UpdateClassView* geladen werden, in einem kleinen Memo-Feld angezeigt, das nur dann eingeblendet wird, wenn die Datei überhaupt einen Inhalt hat (treten bei der Skriptausführung keine besonderen Meldungen auf, bleibt diese Datei leer).

- Die zweite Ebene, in der ein Fehler auftreten kann, ist das Starten der *bash* selbst. Wie in Kapitel 5.3.2 gezeigt, gibt zunächst *RunPrograms* einen Code von 255 zurück, falls die *bash* überhaupt nicht gestartet werden kann. Dann kann es grundsätzlich sein, dass die *bash* selbst einen von Null verschiedenen Exit-Code zurückgibt, z. B. wenn sie die Skriptdatei nicht finden kann, mit falschen Parametern aufgerufen wird oder wenn das Skript selbst einen solchen Exit-Code zurückliefert. Der Exit-Code des Skripts wird durch den Exit-Code des als letzte Anweisung ausgeführten Programms oder durch die explizite *exit*-Anweisung bestimmt. Das Beispielskript gibt nur dann einen Fehler-Code zurück, wenn die Zahl der Parameter nicht stimmt. Dies sollte aufgrund der automatischen Parameterübergabe durch die Kylix-Anwendung jedoch niemals vorkommen. Die obige Methode geht von einer erfolgreichen Ausführung aus, wenn *RunProgram* eine Null zurückliefert, und zeigt ansonsten eine Meldung an.

5.3.4 Die Umgebung eines Prozesses

Abschließend soll an dieser Stelle noch auf wichtige Elemente aus der Umgebung eines Prozesses eingegangen werden – die Umgebungsvariablen. Auch diese hängen wieder eng mit der kommandozeilenorientierten Herkunft von Unix zusammen und bieten nach wie vor aktuelle Möglichkeiten der Konfiguration einzelner Anwendungen.

Eine Umgebungsvariable ist ein vom Betriebssystem verwalteter, mit einem Namen versehener Speicherbereich, der von den einzelnen Anwendungen ausgelesen werden kann. Für jeden Prozess verwaltet das System einen eigenen Satz von Umgebungsvariablen. Jeder Prozess erbt zunächst einmal die Umgebungsvariablen des aufrufenden Prozesses, durch Änderungen können aber spezifische Variablen gesetzt werden, die nur für eine einzelne Anwendung gelten. Für ein Beispiel sei auf Kapitel 5.1.1 verwiesen, in dem die Umgebungsvariable *LD_LIBRARY_PATH* zum Starten einer Kylix-Anwendung gesetzt wurde.

Ermittlung des Home-Verzeichnisses *R118*

Auf Object-Pascal-Ebene können Sie mit der *SysUtils*-Funktion *GetEnvironmentVariable* auf die für Ihre Anwendung gültigen Umgebungsvariablen zugreifen. *GetEnvironmentVariable* ist übrigens portabel, weil auch unter Windows verfügbar. Eine sinnvolle (und wieder nicht portable) Anwendungsmöglichkeit für diese Funktion ist unter Linux die Ermittlung des Heimatverzeichnisses des Anwenders:

```
HomeDir := GetEnvironmentVariable('HOME');
```

Dies wurde in R124 bereits zur Speicherung einer Konfigurationsdatei in diesem Verzeichnis verwendet.

5.4 Die Anbindung an die Qt-Bibliothek

Um das Qt-Toolkit nutzen zu können, benötigt die CLX zur Vermittlung die Bibliothek *qtintf.so*. Die Klassenschnittstelle der Qt ist in Object Pascal nicht direkt ansprechbar. Dieses Kapitel befasst sich mit der Umsetzung dieser Klassenschnittstelle in der Unit *Qt.pas* und demonstriert die Verwendung der Qt anhand der Grafikausgabe in Weltkoordinatensystemen. Die Grundlage dafür liefert eine ganz allgemeine Untersuchung der Unterschiede zwischen Schnittstellen von C++ und Object Pascal.

5.4.1 C++ und Object Pascal

Schon beim Aufruf normaler, nicht objektorientierter C-Funktionen von Object Pascal aus sind gewisse Vorkehrungen notwendig, damit der Maschinencode des C-Compilers korrekt mit dem des Object-Pascal-Compilers zusammenarbeitet. Wie in Kapitel 5.1.2 näher erläutert wurde, ist dazu eine Übereinstimmung des Aufrufformats notwendig, also unter anderem in der Reihenfolge der Parameter.

Methodenaufrufe sind auf Maschinenebene jedoch komplexer als einfache Funktionsaufrufe, denn bei Methoden wird implizit immer eine Referenz auf das *self*-Objekt (in C++ *this* genannt) übergeben. In diesem versteckten Zusatzparameter stimmen C++ und Object Pacsal noch überein. Die Schwierigkeit beginnt jedoch schon bei der Tabelle virtueller Methoden (VMT), auf die der *self*-Parameter Zugriff verschafft. In dieser abelle befinden sich an vorherbestimmten Indizes die Adressen der virtuellen Methoden des konkreten Objekts (auf diese Weise wird die späte Bindung auf Maschinenebene realisiert – da der Compiler zur Übersetzungszeit noch nicht weiß, welche virtuelle Methode tatsächlich aufgerufen werden muss, stellt der Maschinencode dies zur Laufzeit über die VMT fest). Die VMT weist nun aber je nach Compiler ein unterschiedliches Format auf.

Doch selbst eine Übereinstimmung der VMT-Formate würde noch lange keine perfekte Interoperabilität zwischen C++ und Object Pascal gewährleisten, denn neben virtuellen Methoden hat eine C++-Klasse ja auch noch statische Methoden, spezielle Konstruktoren und Destruktoren und evtl. ist sie auch noch von mehreren anderen Klassen abgeleitet. All diese Eigenschaften lassen sich nicht nach Object Pascal übertragen. Insbesondere ist es in Object Pascal nicht möglich, eine Klassendeklaration oder Methoden als *external* zu deklarieren, wie es in Kapitel 5.1.2 bei den externen Funktionen gemacht wurde.

Ausblick: C++-Methoden direkt aufrufen

Mit einem standardisierten VMT-Format wäre ein Aufruf von virtuellen C++-Methoden grundsätzlich möglich. Tatsächlich gibt es auch eine Spezifikation eines Application Binary Interfaces (ABI) für Unix, an die sich Kylix hält, die jedoch noch nicht in

den unter Linux üblicherweise installierten Bibliotheken verbreitet ist. Wenn Sie jedoch selbst eine C++-Bibliothek erzeugen, können Sie wie in der Readme-Datei von Kylix erwähnt den *gcc* ab der Version 2.95 zur Übersetzung verwenden und mit Hilfe des Attributs *COM_INTERFACE* Kylix-kompatible VMTs erzeugen.

Klassendeklarationen in C++ ließen sich dann nach Object Pascal übersetzen, allerdings beschränkt auf die virtuellen Methoden, die in der gleichen Reihenfolge und mit kompatiblen Parameterlisten in Object Pascal deklariert werden müssten. Die Deklaration als *virtual* und *abstract* in Object Pascal würde bewirken, dass der Kompiler von Ihnen keine neue Implementierung der Methoden verlangt.

Um schließlich an ein C++-Objekt zu gelangen, wird dann eine normale, nicht-objektorientierte Funktion in C++ benötigt, die den C++-Konstruktor aufruft, was von Object Pascal aus nicht möglich ist. Diese C++-Funktion kann dann einen Zeiger auf das Objekt zurückliefern, der mit dem Object-Pascal-Objekt der übersetzten Klasse kompatibel ist. Über dieses Objekt können dann die virtuellen C++-Funktionen aufgerufen werden. Die Object-Pascal-Seite dieser Verbindung soll durch das folgende Listing kurz skizziert werden:

```
type
  // C++-Klasse, übersetzt nach Object Pacsal:
  TAbstractCppClass = class
    procedure CppMethod1; virtual; abstract;
  end;

// Externe C++-Methode (muss in C++ als extern "C" deklariert werden)
function GetCppObject: TAbstractCppClass; cdecl; external CppLibrary;

// Benutzung des C++-Objekts:
procedure UseCppObject;
var
  cppObject: TAbstractCppClass;
begin
  cppObject := GetCppObject;
  cppObject.CppMethod1;
end;
```

5.4.2 Die Brücke von Qt zu Object Pascal

Die Bibliothek *qtintf.so*, die standardmäßig im *bin*-Verzeichnis von Kylix installiert wird und zum Start jeder CLX-Anwendung erforderlich ist, stellt einen Großteil der Qt-Schnittstellen für Object Pascal zur Verfügung. Es gibt jedoch auch Qt-Klassen, die nicht in *qtintf* berücksichtigt sind, so z. B. die Klasse *QCanvas* (nicht zu verwechseln mit *TCanvas* der CLX, die der Qt-Klasse *QPainter* entspricht) oder sämtliche XML-Klassen.

Die Beschreibung der *qtintf* in diesem Kapitel dient damit einerseits der Verwendung der Schnittstelle, andererseits als Muster für eigene Schnittstellen zu anderen C++-Klassen, die Sie vielleicht in Object Pascal nutzen wollen.

Wir beginnen gleich mit einem Beispiel für eine Qt-Klasse und deren Übersetzung in Object Pascal. *QWMatrix* kapselt eine Matrix, mit der verschiedene aufeinander folgende Transformationen in einem Weltkoordinatensystem ausgedrückt werden können. Das folgende Listing zeigt nur eine kleine Auswahl ihrer Methoden, darunter sind Beispiele für Konstruktoren (die beiden Funktionen mit dem Namen *QWMatrix*), für berladene Methoden und für einfache Methoden. Ein Destruktor wurde im Listing brigens *nicht* weggelassen. Da *QWMatrix* keine dynamischen Speicherreservierungen vornimmt, reicht der vom C++-Compiler automatisch erzeugte Destruktor aus (auch in Object Pascal würde diese Klasse ja keinen Destruktor benötigen).

```
class Q_EXPORT QWMatrix
{
public:
    { Dieses Listing zeigt nur einen kleinen Ausschnitt des Originals! }
    QWMatrix();
    QWMatrix( double m11, double m12, double m21, double m22,
              double dx, double dy );

    void        map( int x, int y, int *tx, int *ty )         const;
    void        map( double x, double y, double *tx, double *ty ) const;

    void        reset();
    QWMatrix   &rotate( double a );
};
```

Auf solche Deklarationen bezieht sich die gesamte Dokumentation von Qt. Diese ist brigens im HTML-Format bei Trolltech (`www.trolltech.com`) erhältlich und wird möglicherweise auch mit Ihrer Linux-Distribution mitgeliefert.

In Object Pascal sieht dies alles etwas weniger übersichtlich aus, da die Klasse *QWMatrix* selbst nicht nach Object Pascal exportierbar ist und sich die Deklarationen durch ein Namenspräfix und einen zusätzlichen Parameter erheblich verlängern:

```
function QWMatrix_create(): QWMatrixH; overload; cdecl;
procedure QWMatrix_destroy(handle: QWMatrixH); cdecl;
function QWMatrix_create(m11: Double; m12: Double; m21: Double; m22:
           Double; dx: Double; dy: Double): QWMatrixH; overload; cdecl;
procedure QWMatrix_map(handle: QWMatrixH; x: Integer; y: Integer;
              tx: PInteger; ty: PInteger); overload; cdecl;
procedure QWMatrix_map(handle: QWMatrixH; x: Double; y: Double;
              tx: PDouble; ty: PDouble); overload; cdecl;
procedure QWMatrix_reset(handle: QWMatrixH); cdecl;
function QWMatrix_rotate(handle: QWMatrixH; a: Double): QWMatrixH; cdecl;
```

Wie zu erkennen ist, werden zunächst alle Methodenbezeichner mit dem Namen der Qt-Klasse als Präfix versehen. Statt des C++-Konstruktornamens, der ja grundsätzlich mit dem Klassennamen identisch ist, tragen die Konstruktions-Funktionen in *qtintf* den Namen *create*, entsprechend die Funktionen für den Destruktor den Namen *destroy*. *QWMatrix_create* ruft also den C++-Konstruktor auf und liefert eine Referenz auf das von diesem erzeugte Objekt zurück.

QWMatrix_destroy ruft den C++-Destruktor auf, der ja in diesem Fall wie erwähnt vom C++-Compiler automatisch erzeugt wurde. Aber unabhängig davon, wie die Destruktoren in C++ aussehen – in jedem Fall müssen Sie die *destroy*-Prozedur in Object Pascal manuell aufrufen, damit der vom C++-Objekt belegte Speicher wieder freigegeben wird.

Wenn Sie von *QWMatrix_create* eine Referenz auf ein Objekt erhalten haben, können Sie diese Referenz als ersten Parameter *handle* an die anderen Prozeduren und Funktionen übergeben. Diese entsprechen bis auf diesen ersten *handle*-Parameter meist sehr genau den C++-Methoden; im Falle von Strings gibt es Unterschiede, die in einem späteren Abschnitt erläutert werden. Überladene C++-Methoden werden in Object Pascal ganz einfach durch überladene Pascal-Routinen übertragen, wobei hier die *overload*-Direktive Anwendung findet.

Eine Hierarchie leerer Klassen

Eine Besonderheit liegt in dem Handle auf das Qt-Objekt, das im obigen Beispiel den Typ *QWMatrixH* hat und entsprechend bei anderen Qt-Objekten beispielsweise ein *QWidgetH*, *QPainterH* oder *QObjectH* ist. Intern handelt es sich dabei jeweils einfach um einen Zeiger auf ein C++-Objekt. Eigentlich kann nur der C++-Code etwas mit diesem Zeiger anfangen, jedoch sind die *Q...H*-Typen in *Qt.pas* wie folgt deklariert:

```
type
  QWMatrixH = class(TObject) end;
  QtH = class(TObject) end;
        QObjectH = class(QtH) end;
                QWidgetH = class(QObjectH) end;
                        QButtonH = class(QWidgetH) end;
                                QCheckBoxH = class(QButtonH) end;
```

Alle Qt-Handles sind also als Klassen deklariert, wobei die tatsächliche Vererbungshierarchie von Qt nachgebildet wird, in der beispielsweise die Klasse *QCheckBox* von *QButton* abgeleitet ist. Und obwohl diese Klassen in Object Pascal alle leer sind, weil statt Methoden ja die schon gezeigten Prozeduren und Funktionen aufgerufen werden, erfüllen diese Klassendeklarationen einen wichtigen Zweck: Sie sorgen für Typenkompatibilität zwischen dem Handle einer bestimmten Klasse und den Handles aller von dieser Klasse abgeleiteten Klassen.

Wenn also beispielsweise eine Prozedur einen *QWidgetH*-Parameter verlangt, können Sie dieser stattdessen auch ein *QButtonH* übergeben, jedoch kein *QWMatrixH*. Die Typenprüfung des Object-Pascal-Compilers stellt damit die Vererbungsregeln der C++-Klassenhierarchie sicher, obwohl die Klassen nur sehr indirekt verwendet werden.

QString und WideString

Die Verwendung von Klassen ist im Qt-Toolkit so weit verbreitet, dass der Object-Pascal-Code sehr kompliziert werden würde, wenn *Qtintf* nicht einige Vereinfachungen durchführen würde. So werden etwa String-Parameter von Qt-Methoden normalerweise als *QString*-Objekt angegeben. Dieses ist natürlich nicht mit einem Pascal-String kompatibel, sondern müsste erst extra über einen C++-Konstruktor initialisiert werden, dann müsste der Pascal-String als *PChar* an das *QString*-Objekt übergeben werden und nach der Verwendung müsste noch der Destruktor aufgerufen werden. Dies alles wäre wohlgemerkt für jeden simplen Methodenaufruf, an dem ein String beteiligt ist, erforderlich.

Glücklicherweise hat jedoch Borland sämtliche *QString*-Referenzen aus *Qt.pas* herausgehalten. So gibt es in dieser Unit gar kein *QString_create*, weil Sie es nicht benötigen. *QString*-Parameter werden durch die *Qtintf* in *PWideString*-Parameter übersetzt:

```
// Die C++-Deklaration...
class Q_EXPORT QWidget: public QObject, public QPaintDevice {
  ...
  QString caption() const;
  virtual void setCaption( const QString &);

// ...wird in Qt.pas zu:
procedure QWidget_setCaption(handle: QWidgetH; p1: PWideString); cdecl;
procedure QWidget_caption(handle: QWidgetH; retval: PWideString); cdecl;
```

Die Besonderheit hierbei liegt darin, dass Funktionen, die einen *QString* als Ergebnis liefern, in *Prozeduren* übersetzt werden, die keinen neuen *WideString* erzeugen, sondern einen bereits bestehenden *WideString* als Parameter erwarten. Der bisherige Inhalt dieses Strings (falls vorhanden) wird dann durch das Funktionsergebnis der Qt-Methode überschrieben.

Beim Aufruf einer solchen Prozedur muss mit dem Adressoperator die Adresse der *WideString*-Variablen ermittelt werden:

```
var
  Caption: WideString; // vom Compiler automatisch mit '' initialisiert
begin
  QWidget_caption(handle, @Caption);
  // Caption enthält nun den Titel des QWidget-Objekts
```

Qt-Aufzählungstypen in Qt.pas

Ähnlich wie die Methoden von Klassen werden auch die Aufzählungstypen von Qt in der Unit *Qt* nach Object Pascal übertragen. Die einzelnen Aufzählungswerte erhalten als Präfix jeweils den Namen des *enum*-Typs von C++. So wird beispielweis aus dem Enum-Typ *Qt::AlignmentFlags*, der unter anderem die Werte *AlignLeft*, *AlignRight* und *AlignHCenter* annehmen kann, ein Pascal-Aufzählungstyp namens *AlignmentFlags*, dessen Werte nun mit den etwas länglichen Namen *AlignmentFlags_AlignLeft*, *AlignmentFlags_AlignRight* und *AlignmentFlags_AlignHCenter* bedacht wurden.

5.4.3 Das Weltkoordinatensystem von Qt

Die einfachen Grafikmethoden von *TCanvas* basieren auf entsprechenden Grafikmethoden von Qt. Bisher entsprachen die Koordinaten, die wir an diese Grafikmethoden übergeben haben, den Pixelkoordinaten innerhalb eines Fensters oder Widgets. Ein interessanter Qt-Bereich, der keine Berücksichtigung in der CLX findet, ist der Bereich der Koordinatentransformationen. Damit können Grafikelemente verschoben, rotiert und skaliert werden, ohne dass die an die Grafikmethoden übergebenen Koordinaten geändert werden müssen. Nach der Transformation erhält man ein so genanntes Welt-Koordinatensystem, in dem ein Fenster (Window) platziert werden kann, welches den am Bildschirm angezeigten Bereich eingrenzt. Dieser Bereich wird schließlich auf einen Viewport projiziert, der die tatsächliche Ausgabefläche, z. B. in einem Fenster am Bildschirm oder auf dem Drucker, darstellt.

Die Stufen zwei und drei, Window und Viewport, werden in Kapitel 6.6.1 am Beispiel des TreeDesigners erläutert, so dass wir uns hier auf die erste Stufe, die Weltkoordinaten-Transformation, konzentrieren können.

Hinweis: Wenn Sie die Transformationsfunktionen von Qt verwenden, sind Sie wieder von dem darunter liegenden Betriebssystem unabhängig, was insbesondere für die Windows-Systeme eine angenehme Sache ist, da dort die Windows-Versionen mit NT-Technologie Weltkoordinaten-Transformationen unterstützen. Mit Qt können Sie nun auch auf Windows 9x solche Transformationen vornehmen.

Da die virtuellen Koordinatensysteme sich nicht über die *TCanvas*-Klasse der CLX einstellen lassen, müssen wir hier an *TCanvas* vorbei direkt die Methoden von Qt aufrufen. Hierbei gelten die in den vorangegangenen Abschnitten beschriebenen Prinzipien: Da die Qt-Klassen in C++ geschrieben sind, werden sie in Object Pascal über eine prozedurale C-Schnittstelle aufgerufen, welche ja für die meisten Qt-Klassen mit Kylix bereits mitgeliefert wird, so erfreulicherweise auch für die hier benötigten Methoden.

Qt-Methoden in C++ und Kylix

Wenn Sie die vollständige Distribution von Qt besitzen, die als Free Edition kostenlos von Trolltech heruntergeladen werden kann, können Sie anhand des C++-Beispielprogramms *drawdemo* sehen, wie die Koordinatensysteme in C++ eingestellt werden. Das folgende Beispiel verdeutlicht die grundsätzliche Vorgehensweise:

```
// C++-Code, ähnlich dem aus drawdemo/drawdemo.cpp
QWMatrix matrix;
matrix.translate(100, 10);
matrix.rotate(winkel);
Painter->setWorldMatrix(matrix);
Painter->drawText(x, y, "Rotierter Text");
```

Die Matrix speichert eine Folge von Transformationen, in diesem Fall eine Verschiebung um 100 Einheiten in x-Richtung und 10 Einheiten in y-Richtung sowie eine Rotation um *winkel* Grad. Wenn eine solche Matrix mit *setWorldMatrix* an das *Painter*-Objekt weitergegeben wird, werden die in ihr gespeicherten Transformationen für alle folgenden Grafikausgabemethoden verwendet.

In der prozeduralen Schnittstelle von Object Pascal heißen die oben verwendeten Methoden *QWMatrix_translate*, *QPainter_setWorldMatrix* usw., zusätzlich müssen die verwendeten Matrix-Objekte explizit mit einem speziellen Prozeduraufruf erzeugt *QWMatrix_create*) und wieder freigegeben werden (*QWMatrix_destroy*). Dies entspricht der standardmäßigen, in Kapitel 5.4.2 erläuterten Vorgehensweise.

Matrizen und Modelle

Ein Grund dafür, dass die Transformationsdaten in ein eigenes Matrix-Objekt ausgelagert werden, anstatt alleine durch das *Painter*-Objekt verwaltet zu werden, liegt darin, dass in vielen Grafikanwendungen mehrere bestimmte Objekte gezeichnet werden, die verschieden transformiert werden sollen, wozu man mehrere Matrizen braucht. Wenn diese auch in mehreren dauerhaften Objekten gespeichert werden, müssen sie nicht bei jedem *Paint*-Ereignis neu erzeugt, sondern nur noch der Reihe nach mit *setWorldMatrix* an das *Painter*-Objekt weitergegeben werden.

Dies ist vielleicht am einfachsten vorstellbar, wenn Sie an eine 3D-Anwendung denken (in *QPainter* gibt es allerdings *keine* 3D-Koordinatensysteme!), die zwei unterschiedliche sich bewegende Objekte zeichnet, beispielsweise einen Vogel und einen Schmetterling. Jedes dieser Grafikobjekte hat ein eigenes Modellkoordinatensystem, dessen Ursprung im Zentrum des Objekts liegen sollte, also etwa im vermeintlichen Schwerpunkt des Vogels. Ändert sich nun der Aufenthaltsort des Vogels, wird seine Matrix angepasst, was sich nicht auf die eigentliche Grafikausgabe auswirkt: Diese findet für den Vogel weiterhin in einem Modellkoordinatensystem statt, in dessen Nullpunkt sich der Vogel befindet (wenn es sich um einen animierten Vogel handelt, wird sich durch die Bewegung allerdings auch die Grafik in diesem Modellkoordinatensystem ändern).

Dies lässt sich auch auf die 2D-Anwendungen von *QPainter* übertragen: Jedes Objekt sollte immer im Nullpunkt seines Modellkoordinatensystems gezeichnet werden. Wenn es sich etwa um ein einfaches Quadrat handelt, so müsste es mit *Rectangle(-50, -50, 50, 50*) gezeichnet werden, statt mit *Rectangle(0, 0, 100, 100)*. Andernfalls kann es beim Rotieren und Skalieren zu unerwünschten Effekten kommen, wie Abbildung 5.4 zeigt: Das Objekt wird nicht nur rotiert, sondern gleichzeitig auch verschoben. Damit mehrere solchermaßen zentrierte Objekte am Bildschirm nicht übereinander liegend erscheinen, müssen sie lediglich entsprechend mit *translate* verschoben werden.

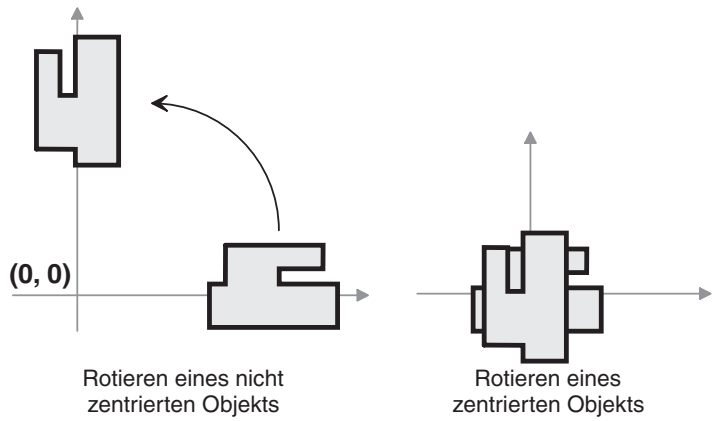


Abbildung 5.4: Das Rotieren von nicht zentrierten Objekten führt zu einer möglicherweise ungewollten Verschiebung.

Transformationsmethoden im Überblick

Das Beispielprogramm wird der Einfachheit halber keine eigenen *QWMatrix*-Objekte verwenden, sondern alle Transformationen direkt im *QPainter*-Objekt vornehmen, welches automatisch eine einzelne solche Matrix intern verwendet. Dessen mit der Transformation zusammenhängende Methoden sind im Folgenden kurz aufgeführt:

QPainter-Methode	**Beschreibung**
setWorldXForm(Boolean)	schaltet die gesamte Welttransformation aus oder ein.
hasWorldXForm: Boolean	Funktion, die besagt, ob die Transformation eingeschaltet ist.
resetXForm	setzt die interne Transformationsmatrix zurück (keine Verschiebung, keine Rotation, usw.).
translate(x, y: Double)	verschiebt die Grafik um *x*/*y* Einheiten. Wenn vorher bereits eine Rotation angewendet wurde, ist zu beachten, dass sich dadurch auch die Richtungen der Koordinatenachsen geändert haben.
rotate(a: Double)	rotiert die Grafik um *a* Grad um den Nullpunkt.

QPainter-Methode	Beschreibung
scale(sx, sy: Double)	skaliert die Grafik um den Faktor *sx* in X-Richtung, um *sy* in Y-Richtung.
shear(sh, sv: Double)	verändert den Winkel zwischen den Koordinatenachsen.
xForm(...)	Diese Methode liegt in verschiedenen Varianten vor, um einzelne Punkte, Rechtecke und Punktelisten zu transformieren, ohne dabei Grafik auszugeben. Wenn Sie diese Methoden verwenden wollen, sollten Sie sich in *Qt.pas* zunächst über das übersetzte Aufrufformat in Object Pascal informieren (Prozeduren *QPainter_xForm*).

Die Tabelle die Parameter der Methoden bereits in Pascal-Syntax an, jedoch müssen Sie eim Aufruf von Object Pascal aus noch alle Methoden um das Präfix *QPainter_* erweitern und als ersten Parameter ein *QPainterH* hinzufügen.

Zugriff auf das Painter-Objekt *R104*

Als zusätzliche Schwierigkeit kommt nun hinzu, dass unser *Painter*-Objekt von Qt mit den Widgets der CLX von Kylix zusammenarbeiten soll. Wir können also nicht einfach ins Blaue hinein einen neuen *Painter* konstruieren, sondern benötigen einen, der genau auf einem CLX-Steuerelement zeichnet. Dieser ist zunächst schnell gefunden, denn jedes *TCanvas*-Objekt von Kylix verwaltet von sich aus ein *QPainter*-Objekt. Über *TCanvas.Handle* können Sie ganz einfach eine Referenz auf dieses *QPainter*-Objekt erhalten (der genaue Typ von *TCanvas.Handle* lautet *QPainterH* und ist somit mit dem ersten Parameter der erwähnten Routinen kompatibel).

Start und Stop

Es bleibt dann nur noch ein Punkt zu bedenken: Die CLX initialisiert das *QPainter*-Objekt standardmäßig bei jeder Zeichenoperation neu, was etwaige Einstellungen des Koordinatensystems wieder löschen würde. Aber auch hier haben die Entwickler bei Borland mit einem flexiblen Design vorgesorgt und uns durch die Methoden *Start* und *Stop* in *TCanvas* erlaubt, selbst Anfang und Ende einer Zeichenoperation zu definieren. Sie können daher am Anfang der Grafikausgabe *Start* aufrufen (die CLX führt darin ihre eigenen Initialisierungen des *QPainter*-Objekts durch), danach können Sie beliebige Koordinatensystem-Einstellungen vornehmen und Methoden zur Grafikausgabe aufrufen. Am Schluss rufen Sie *Stop* auf und erlauben der CLX damit, die Einstellungen des *QPainter*-Objekts wieder zu vergessen.

Start/End-Blöcke können auch geschachtelt werden: Während *Start* einen internen Zähler um eins erhöht, zieht *Stop* eins von diesem Zähler ab. *Start* führt nur dann eine *QPainter*-Initialisierung durch, wenn der Zähler vorher noch bei Null stand.

Eine Zeichenmethode in Weltkoordinaten

Die in Abbildung 5.5 gezeigte Grafik des Beispielprogramms *WorldXForm* setzt sich aus zwei Teilen zusammen:

- Der Hintergrund wird mit vier mal vier Quadraten abgedeckt, die jeweils um einen anderen Winkel rotiert werden und in ihrem Zentrum (Nullpunkt ihres Modell-Koordinatensystems) mit einer Markierung sowie einem Ausdruck der Winkelzahl versehen sind.

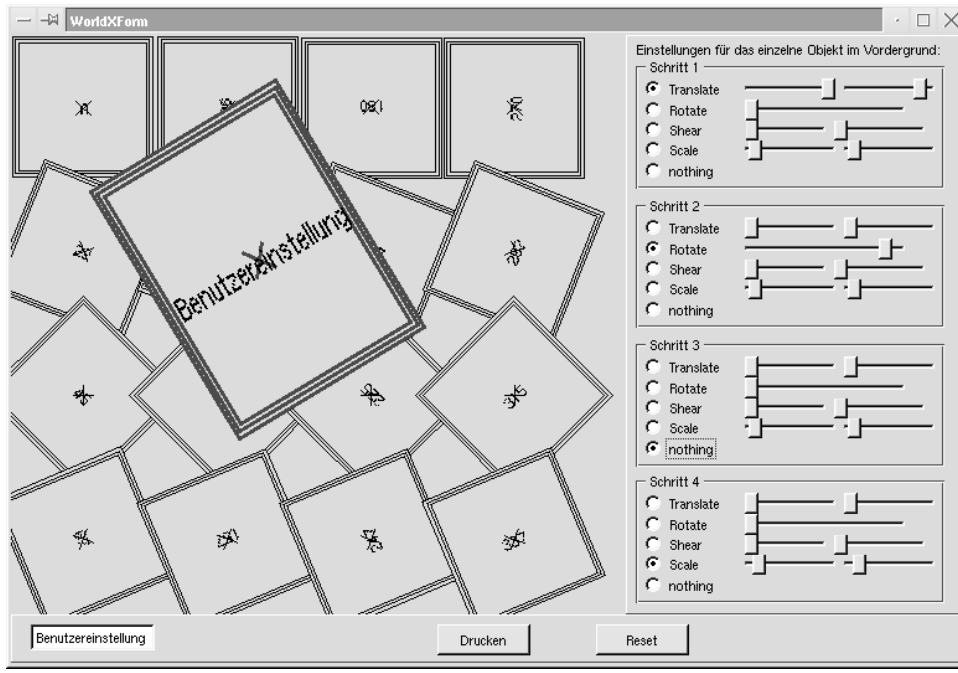


Abbildung 5.5: Über die Transformations-Steuerelemente im Beispielprogramm können sehr anschauliche Experimente gemacht werden.

- Auf diesem Hintergrund wird ein weiteres, einzelnes Quadrat gezeichnet, das Sie zur Laufzeit mit den Steuerelementen am rechten Fensterrand transformieren können (so dass es unter Umständen auch nicht mehr als Quadrat erscheint). In jeder Steuerelement-Gruppe stellen Sie einen einzelnen Transformationsschritt ein, dessen Parameter Sie über ein bzw. zwei TrackBars einstellen. Auf diese Weise können Sie bequem und interaktiv ausprobieren, welche unterschiedlichen Effekte sich etwa durch die Vertauschung von Translation und Rotation ergeben. Das Programm lässt sich im Übrigen auch gut als »Spiel« verwenden, bei dem man versucht, eine ganz bestimmte Lage des Rechtecks durch Einstellung der Transformationsparameter zu erreichen.

Alle Rechtecke setzen sich aus drei *Rectangle*-Aufrufen zusammen und werden mit einer einzigen Prozedur gezeichnet, die immer dieselben (Modell-)Koordinaten verwendet. Dabei wird das erwähnte Prinzip beachtet, dass die Mitte des Grafikelements im Nullpunkt liegt, die Koordinaten von *Rectangle* erstrecken sich also gleichermaßen in den negativen wie in den positiven Zahlenbereich.

Im Listing wurden der Einfachheit halber die Parameter zur Einstellung von Stiftbreite und -farbe sowie der Textfarbe weggelassen, einziger Parameter bleibt der im Zentrum des Rechtecks geschriebene Text:

```
procedure PaintObject(Canvas: TCanvas; str: String);
var
  WideStr: WideString;
begin
  WideStr := str;
  Canvas.Pen.Width := PenWidth;
  Canvas.Pen.Color := PenColor;
  // Umrandung malen:
  Canvas.Rectangle(-49, -49, 49, 49);
  Canvas.Rectangle(-47, -47, 47, 47);
  Canvas.Rectangle(-45, -45, 45, 45);
  QPainter_drawtext(Canvas.Handle, -49, -49, 99, 99,
                    integer(AlignmentFlags_AlignHCenter) or
                    integer(AlignmentFlags_AlignVCenter),
                    @WideStr, -1, nil, nil);
end;
```

Damit auch die Mitte des Textes genau mit dem Nullpunkt übereinstimmt, wird dieser oben nicht mit den Canvas-Methoden *TextOut* oder *TextRect* gezeichnet, denn diese positionieren den Text mit der linken oberen Ecke. Die *QPainter*-Methode *drawtext* hingegen gestattet durch die Angabe der *AlignmentFlags AlignHCenter* und *AlignVCenter* die exakte Zentrierung des Textes in einem angegebenen Rechteck. Dieses Rechteck wird oben durch die Koordinaten (-49, -49, 99, 99) angegeben. Dabei geben die Zahlen drei und vier Breite und Höhe des Rechtecks an. Auch dieses Rechteck ist also im Nullpunkt zentriert, so dass letztlich auch der im Rechteck zentrierte Text sein Zentrum im Nullpunkt erhält.

ansformation und Aufruf von PaintObject

Die Prozedur *PaintObject* wird nun zunächst für die Rechtecke im Hintergrund aufgerufen. Die *Paint*-Methode hat also die Aufgabe,

- für den Aufruf von *Start* und *Stop* in *TCanvas* zu sorgen
- die Koordinaten-Transformation einzustellen
- das Zeichnen der Objekte mit *PaintObject* zu veranlassen.

Wichtig ist die Reihenfolge, in der die Transformationsmethoden *scale*, *translate* und *rotate* angewendet werden. Die zuerst angewendete Transformation wirkt sich auf alle folgenden Transformationen aus. Wird also zuerst die Rotation angewendet, findet eine folgende Translation entlang rotierter Koordinatenachsen statt, was im Beispielprogramm nicht erwünscht ist.

Wenn andererseits die Skalierung zuerst angewendet wird, wie im Beispielprogramm, werden auch die Koordinaten der folgenden Translation automatisch skaliert. Die Parameter für die Translation werden daher im Folgenden fest angegeben (50, 150, 250 und 350 für die vier Rechtecke einer Spalte bzw. Zeile) und durch die vorher angewendete Skalierung automatisch auf die aktuelle Fenstergröße skaliert:

```
procedure TForm1.PaintboxPaint(Sender: TObject);
var
  x, y: Integer;
  Painter: QPainterH;
  Winkel: Integer;
  // Von der verfügbaren Fläche wird nur das größte noch sichtbare
  // Quadrat genutzt, das in 4x4 Quadrate aufgeteilt wird und dessen
  // Seitenlänge in SideLength gespeichert wird:
  SideLength: Integer;
begin
  with Paintbox1.Canvas do begin
    Start(True);
    Painter := Handle;
    for x := 0 to 3 do begin
      for y := 0 to 3 do begin
        QPainter_resetXForm(Painter);
        QPainter_setWorldXForm(Painter, True);
        Winkel := trunc(360*((x*4+y) / 16));
        SideLength := Paintbox1.Width;
        if Paintbox1.Height < Paintbox1.Width then
          SideLength := Paintbox1.Height;
        QPainter_scale(Painter, (SideLength div 4) / 100,
                       (SideLength div 4) / 100);
        QPainter_translate(Painter, 50+100 * x, 50+100 * y);
        QPainter_rotate(Painter, Winkel);
        PaintObject(Paintbox1.Canvas, IntToStr(Winkel));
      end;
    end;
    ... (Hier folgt noch das Zeichnen des einzelnen
         Rechtecks, siehe unten!)
    Stop; // <- Wichtig!
  end;
end;
```

In diesem Beispiel findet übrigens eine Schachtelung von *Start* und *Stop* statt, wenn auch unsichtbar: Die *TCanvas*-Methode *Rectangle* ruft, wie auch die anderen Zeichenmethoden, selbst *Start* und *Stop* auf. Da wir bereits vorher mit *Start* den internen Zähler auf eins gesetzt und damit die Initialisierung des *Painters* veranlasst haben, wird die Initialisierung des Painters durch *TCanvas* während der Ausführung der *TCanvas*-Ausgabemethoden nicht noch einmal wiederholt und das Koordinatensystem bleibt erhalten.

Hinweis: Dem richtigen Aufruf von *Stop* kommt eine besondere Wichtigkeit zu. Nicht nur, dass bei vergessenem *Stop*-Aufruf die Zeichenoperation immer als offen betrachtet wird und spätere Neu-Initialisierungen verhindert werden; wenn Sie *Stop* zu früh aufrufen, verlieren Sie alle nach diesem Aufruf getätigten Grafikausgaben, denn nach *Stop* verpuffen direkte Aufrufe von Grafikausgabe-Methoden über die Qt-Schnittstelle wirkunglos.

ames

Das Zeichnen des einzelnen Rechtecks bringt nun keine Neuigkeiten bezüglich der Koordinatentransformation mehr. Die eigentliche Ausgabe findet ja in der schon gezeigten Prozedur *PaintObject* statt und zur Einstellung des Koordinatensystems werden die gleichen Methoden verwendet wie oben (zusätzlich noch die *shear*-Methode), nur dass nun die vom Benutzer im »Transformations-Panel« am rechten Fensterrand eingestellten Parameter verwendet werden.

Die eigentliche Besonderheit liegt hierbei darin, dass das Transformations-Panel ein gutes Beispiel für die Verwendung von Frames gibt. Die Steuerelemente für einen einzelnen Transformationsschritt wurden nur einmal in einem Frame entworfen, das dann mehrfach in das Formular eingezeichnet wurde. Dieses Frame besitzt auch eine Methode, die die Einstellungen seiner Steuerelemente auf ein *Painter*-Objekt überträgt. Diese Methode ruft lediglich wieder *scale*, *translate* usw. auf und braucht daher nicht abgedruckt zu werden.

An der im letzten Listing mit »...« markierten Stelle kommen noch die folgenden Anweisungen hinzu, um das einzelne Rechteck zu zeichnen:

```
QPainter_resetXForm(Painter);
frmStep1.ApplySettings(Painter); // Aufruf der Methode des Frames
frmStep2.ApplySettings(Painter);
frmStep3.ApplySettings(Painter);
frmStep4.ApplySettings(Painter);
PaintObject2(Edit1.Text, 3 (* Stiftbreite *), clRed);
```

5.5 Ein KDE-Widget als Komponente

Der Umstand, dass C++-Klassenbibliotheken sich in Object Pascal nicht direkt vollständig nutzen lassen, wurde bereits in Abschnitt 5.4 erläutert. Dort wurde auch gezeigt, wie Borland dieses Problem mit Qt gelöst hat, indem die C++-Klassen von Qt in einfachen C-Funktionen verpackt wurden, die in Kylix so aufgerufen werden können, wie es in den Abschnitten 5.1 und 5.3 mit anderen C-Funktionen vorgemacht wurde.

Dieser Abschnitt geht auf die Benutzung von Klassen der Desktop-Umgebung KDE in Kylix ein. Da auch KDE in C++ geschrieben ist, aber keine Schnittstellen-Bibliothek für KDE mit Kylix mitgeliefert wird, müssen wir nun selbst eine solche schreiben. In Abschnitt 5.5.1 wird das Web-Browser-Widget von KDE in Kylix importiert, welches auch die Browsing-Fähigkeiten des Konqueror-Web-Browsers bereitstellt. Abschnitt 5.5.2 zeigt, wie eine Anwendung so umgeschrieben werden kann, dass sie nicht mehr das Vorhandensein einer von ihr verwendeten Bibliothek (hier: die Bibliothek *khtml*) voraussetzt, sondern auch ohne diese Bibliothek läuft – wenn auch mit eingeschränktem Funktionsumfang.

Lizenz-Fragen

Vorher soll jedoch nicht versäumt werden, auf eine Besonderheit hinzuweisen: Wenn Sie externe Bibliotheken verwenden, müssen Sie natürlich deren Lizenzbedingungen beachten. Wird eine Bibliothek etwa unter der GNU General Public License (GPL) verbreitet, bedeutet das unter anderem, dass alle Programme, die die Bibliothek verwenden, ebenfalls unter die GPL zu stellen sind und dass sie nur dann weitergegeben werden dürfen, wenn ihr Quelltext frei verfügbar ist.

Während die fertigen Anwendungen von KDE unter der GPL stehen, verwenden grundlegende Bibliotheken aus dem *kdelibs*-Package die LGPL (Library GPL). Zu diesen Bibliotheken gehört auch *khtml*, deren HTML-Widget wir in diesem Kapitel verwenden werden. Die LGPL erlaubt das dynamische Linken der KDE-Bibliotheken, ohne dass dadurch eine Verpflichtung zu einem bestimmten Lizenzmodell entstehen würde. Da mit Kylix sowieso nur ein dynamisches Linken von Shared Libraries möglich ist, hält sich eine Kylix-Anwendung automatisch an diese Bestimmung.

5.5.1 Ein fest eingebundenes KDE-Widget

Der Web-Browser von KDE hat seit der KDE-Version 2.0 deutlich an Funktionalität gewonnen und unterstützt nun Java, Javascript, HTML 4.0, CCS1 und CCS2, zumindest teilweise (durch die ständige Weiterentwicklung des Konquerors kann dieses Buch Ihnen hierüber keine endgültigen Informationen liefern). Der Vergleich in Abbildung 5.6 zeigt, dass es auch bei der Darstellung von Standardelementen einer Web-Seite Unterschiede zwischen dem KHTML-Widget und dem Qt-Widget *QTextBrowser* gibt.

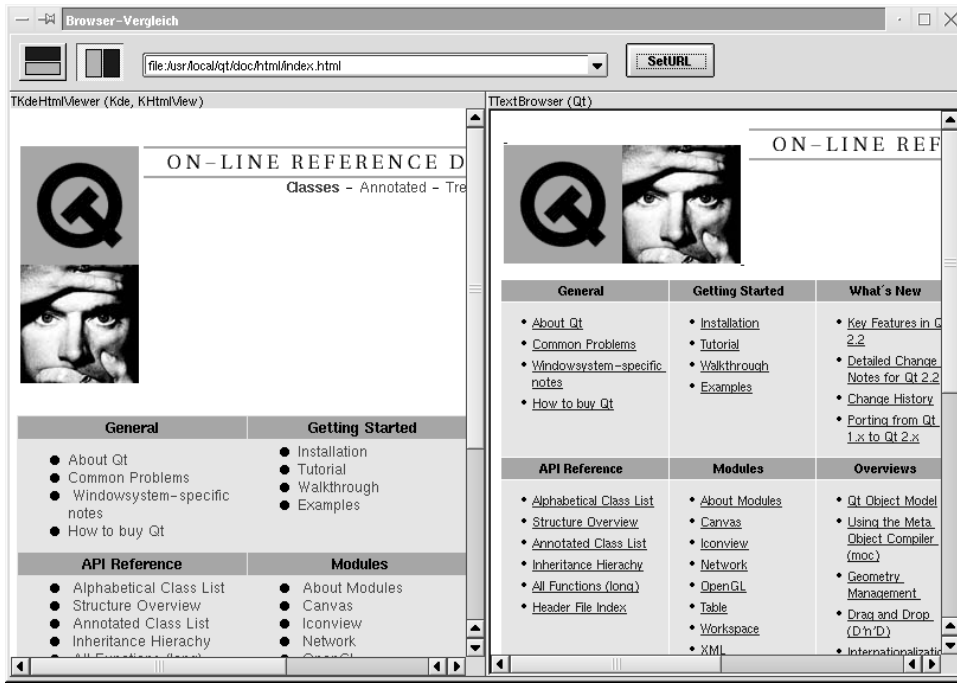


Abbildung 5.6: Ein TextBrowser und ein KHTML-Widget in einer Kylix-Anwendung zeigen unterschiedliche Darstellungsweisen derselben Web-Seite.

Hinweis: Sofern Sie in Ihrer Anwendung feste, von Ihnen bereitgestellte HTML-Dokumente darstellen wollen und die *TTextBrowser*-Komponente dazu ausreicht, sollten Sie diese Komponente auch verwenden, wenn Sie Ihrer Anwendung eine zusätzliche Abhängigkeit ersparen wollen. *TTextBrowser* läuft auf allen Systemen, auf denen überhaupt CLX-Anwendungen laufen (es wird ja lediglich Qt benötigt). Das KDE-Widget läuft zwar auch auf anderen Desktops als KDE, jedoch muss dafür wenigstens das *kdelibs*-Package installiert sein.

Schreiben des C-Interfaces *151*

Bevor wir in Kylix eine Komponentenklasse erzeugen können, benötigen wir von Object Pascal aus einen Zugriff auf die C++-Klassen von KDE. Diesen soll hier eine kleine in C geschriebene Schnittstellen-Bibliothek liefern (auf der CD: `Complib/khtml_wrapper.cpp`), die vom Prinzip her der in Kapitel 5.4 beschriebenen *qtintf* von Borland ähnlich, jedoch auf ein Minimum von drei Funktionen beschränkt ist:

- Mit *ktmlview_create* wird ein Objekt der Klasse *KHTMLView* als Teil eines *KHTMLPart* erzeugt. Als Parameter erwartet die Funktion eine Referenz auf ein Qt-Widget, das als Eltern-Widget für das HTML-Widget dienen soll. Dieses Qt-Widget kann natürlich auch eine *TWidgetControl*-Komponente der CLX sein.
- Mit *khtmlview_getPart* erhalten Sie zu einem gegebenen *KHTMLView* den damit verbundenen *KHTMLPart*.
- Schließlich öffnen Sie mit *khtml_openUrl* ein neues Dokument, dessen URL Sie in einem nullterminierten String als Parameter angeben:

Alle drei Funktionen werden im C-Quelltext in einem *extern "C"*-Block eingeschlossen, damit die Funktionsbezeichner originalgetreu in die Bibliothek übernommen und nicht wie unter C++ üblich mit zusätzlichen Typinformationen erweitert werden.

Die Implementierung der Funktionen besteht überwiegend nur aus kurzen Aufrufen von KDE und hält sich dabei an die Dokumentation von KDE, in der auch die Konstruktion eines einfachen Web-Browsers beschrieben wird (in der Quelltext-Datei `khtml_part.h` oder in der HTML-Dokumentation unter *KHTMLPart*; im Internet unter `developer.kde.org` in der Online-API-Referenz).

```
include <khtmlview.h>
#include <khtml_part.h>
#include <qstring.h>

extern "C" {

KHTMLView *khtmlview_create(QWidget *parent, char *name)
{
    KHTMLPart *part = new KHTMLPart(parent);
    KHTMLView *html = part->view();
    return html;
}

KHTMLPart *khtmlview_getPart(KHTMLView *view)
{
    return (view!=NULL) ? view->part() : NULL;
}

void khtmlpart_openUrl(KHTMLPart *part, char *URL)
{
    if (!part) return;
    KURL url(URL);
    part->openURL(url);
}

} // extern "C"
```

Die Routine *khtmlpart_openUrl* erfüllt auch noch eine typische Aufgabe für eine Bibliothek, die zwischen Object Pascal und C++ vermittelt: Sie benötigt zum Aufruf der *openURL*-Methode von KDE ein *KURL*-Objekt, das als komplexe Verpackung für einen String angesehen werden kann (ähnlich der *QString*-Parameter in zahlreichen Qt-Methoden). Die obige Funktion entlastet ihren Aufrufer nun von der Verpflichtung, ein *KURL*-Objekt konstruieren zu müssen, und erwartet einfach einen nullterminierten String (*PChar* in Object Pascal) als Parameter (zum Vergleich: Die in *Qt.pas* deklarierten Funktionen zum Aufruf von Qt-Methoden erwarten statt eines *PChar* normalerweise einen *PWideString*-Parameter).

Hinweis: Falls Sie eine *khtml_destroy*-Funktion vermissen – diese wird nicht benötigt, da wir das KDE-Widget in einem Formular einer CLX-Anwendung verwenden und sicherstellen werden, dass das KDE-Widget automatisch durch die CLX mit *QWidget_destroy* freigegeben wird.

ompilierungs-Fragen

Diese Datei muss nun noch mit *gcc* in eine *so*-Bibliothek übersetzt und in einem Verzeichnis abgelegt werden, das über den Bibliothekspfad von Linux erreichbar ist. Beides kann etwa mit folgendem einfachen Skript, das Sie unter dem Namen *build_khtmlwrapper* auf der CD finden, erledigt werden:

```
g++ khtml_wrapper.cpp -I$KDEDIR/include -I$QTDIR/include -c
g++ -shared -Wl,-soname,libkhtmlintf.so -o libkhtmlintf.so.1.0.0
    khtml_wrapper.o -L$KDEDIR/lib -lkhtml
rm /usr/lib/libkhtmlintf.so     # alte Dateiversion löschen
ln libkhtmlintf.so.1.0.0 /usr/lib/libkhtmlintf.so # harter Link
#oder: Kopieren mit cp libkhtmlintf.so.1.0.0 /usr/lib/libkhtmlintf.so
```

Dabei muss die Umgebungsvariable *QTDIR* auf das Verzeichnis der Qt-Installation gesetzt sein (etwa `/usr/local/qt`) und *KDEDIR* entsprechend auf die KDE-Installation (z.B. `/opt/kde2`). Beide Variablen sollten in aktuellen Linux-Distributionen automatisch gesetzt werden, wenn KDE installiert wird.

Im ersten Schritt wird mit *g++* die Quelltextdatei *khtml_wraper.cpp* in eine Objektdatei bersetzt (Option *-c*), dabei werden mit der Option *-I* die Verzeichnisse angegeben, in denen sich die von der Datei verwendeten Include-Dateien befinden.

Im zweiten Aufruf von *g++* wird der Linker aufgerufen, um die Bibliothek *libkhtmlintf.so.1.0.0* zu erzeugen. Diese soll jedoch wie unter Linux üblich nicht durch den vollen Namen, sondern durch *libkhtmlintf.so* angesprochen werden können. Normalerweise wird unter diesem kurzen Namen ein Link auf die jeweils aktuelle Datei gelegt (dies kann später auch eine Datei z.B. mit der Versionsnummer 1.0.1 sein). Das obige Skript legt einen harten Link auf die Datei im Verzeichnis `/usr/lib` ab, so dass der Programm-Lader keine Probleme haben sollte, die Datei zu finden.

Hinweis für Windows-Nutzer: Symbolische Links können unter Linux nicht auf dem Dateisystem von Windows angelegt werden. Ein harter Link ist weder vom Windows-Dateisystem auf das Linux-Dateisystem, noch umgekehrt möglich. In diesem Fall muss eine Bibliothek, die über `/usr/lib` angesprochen werden soll, eventuell vom Windows-Dateisystem in `/usr/lib` kopiert werden.

Übertragen der C-Funktionen nach Object Pascal

Nun können die drei C-Funktionen in Object Pascal deklariert werden, wobei die Direktiven *cdecl* und *external* verwendet werden, wie in Kapitel 6.1 beschrieben.

Die Qt-Klasse *QWidget* sowie die KDE-Klassen *KHTMLView* und *KHTMLPart* werden in die Handle-Typen *QWidgetH*, *KHtmlPartH* und *KHtmlViewH* übersetzt. *QWidgetH* ist bereits in *Qt.pas* vordefiniert. Die beiden anderen Typen werden nach dem in Kapitel 5.4.2 unter *Eine Hierarchie leerer Klassen* beschriebenen Muster deklariert, wobei die Original-Vererbungshierarchie von KDE nachgebildet wird – *KHTMLView* ist von *QScrollView* abgeleitet, folglich leiten wir *KHtmlViewH* von *QScrollViewH* (definiert in *Qt.pas*) ab:

```
const
  khtmlintfname = 'libkhtmlintf.so.1.0.0';

type
  KHtmlViewH = class(QScrollViewH) end;
  KHtmlPartH = class end;

function khtmlview_create(Parent: QWidgetH; Name: PChar): KHtmlViewH;
  cdecl; external khtmlintfname name 'khtmlview_create';
function khtmlview_getPart(view: QWidgetH): KHtmlPartH; cdecl;
  external khtmlintfname name 'khtmlview_getPart';
procedure khtmlpart_openUrl(part: KHTMLPartH; URL: PChar); cdecl;
  external khtmlintfname name 'khtmlpart_openUrl';
```

Entwicklung der Komponente *R143*

Nach diesen Vorarbeiten ist nun der Weg frei, um eine Komponente für das KDE-Widget zu programmieren. Zunächst stellt sich die Frage, von welcher Klasse die neue Komponente abgeleitet werden soll. Es bieten sich *TWidgetControl* und die von dieser abgeleiteten Klassen an. Wir wählen hier die Klasse *TFrameControl*, die dem *TWidgetControl* noch einen Rahmen hinzufügt und von der auch die CLX-Klasse *TTextBrowser* abgeleitet ist:

```
type
  TKdeHtmlView = class(TFrameControl)
  private
  protected
    procedure CreateWidget; override;
```

```
  public
    procedure SetUrl(Url: String);
  published
  end;
```

Entscheidend für die Funktionsweise der Komponente ist die geschützte Methode *CreateWidget*. Diese virtuelle Methode wird von *TWidgetControl* eingeführt, damit jede spezielle Klasse darin ein Qt-Widget der passenden Klasse erzeugt. *TWidgetControl* beispielsweise erzeugt in *CreateWidget* darin ein Widget der ganz allgemeinen Klasse *QWidget*, *TFrameControl* wendet sich an die Klasse *QFrame*, und zwar wie folgt:

```
// Aus der CLX, QControls.pas:
procedure TFrameControl.CreateWidget;
begin
  FHandle := QFrame_create(ParentWidget, nil, WidgetFlags, True);
end;
```

Damit nun bei der Erzeugung unserer neuen Komponente kein überflüssiges *QFrame* erzeugt wird, darf die neue Version von *CreateWidget* nicht mit *inherited* die geerbte *CreateWidget*-Methode aufrufen, sondern muss sich darauf beschränken, mit der selbst geschriebenen C-Funktion ein *KHTMLView* zu erzeugen:

```
procedure TKdeHtmlView.CreateWidget;
begin
// kein  inherited; !!
  FHandle := khtmlview_create(ParentWidget, PChar(Name));
end;
```

FHandle ist ebenfalls von *TWidgetControl* geerbt und ist für die Speicherung einer Referenz auf das verwendete C++-Widget-Objekt vorgesehen. Komponentenbenutzer sprechen diese Variable über das *Handle*-Property an. Indem wir das neu erzeugte Widget in dieser Variablen eintragen, ermöglichen wir der CLX, automatisch für die Freigabe des Widgets zu sorgen. Für diese Freigabe definiert *TWidgetControl* analog zu *CreateWidget* eine Methode *DestroyWidget*, die unter anderem bei der Freigabe der Komponente ausgeführt wird, also spätestens wenn das zugehörige Formular geschlossen wird. Die vordefinierte *TWidgetControl.DestroyWidget* ruft nun *QWidget_destroy* mit dem Wert von *FHandle* auf. Da *QWidget::destroy* ein virtueller Destruktor ist, führt der Aufruf von *QWidget_destroy* automatisch zum Aufruf des KDE-Destruktors *KHTMLiew::destroy*. Unsere neue Klasse *TKdeHtmlView* braucht daher *DestroyWidget* nicht zu berschreiben und die C-Bibliothek erspart sich eine Funktion *khtmlview_destroy*.

Die einzige neue öffentliche Methode von *TKdeHtmlView* ist *SetUrl*, mit der der Browser zum Ansteuern einer neuen Adresse angeleitet werden soll. Durch den anfänglichen Aufruf von *HandleNeeded* wird sichergestellt, dass das Widget bei Bedarf noch konstruiert wird (dabei wird dann unter anderem *CreateWidget* aufgerufen).

```
procedure TKdeHtmlView.SetUrl(Url: String);
var
  part: KHtmlPartH;
begin
  HandleNeeded; // TWidgetControl-Methode
  part := khtmlview_getPart(FHandle);
  if part <> nil then
    khtmlpart_openUrl(part, PChar(Url));
end;
```

Hinweis: Es ist wichtig, dass das Objekt, das wir an *FHandle* zuweisen, mit der Widget-Klasse kompatibel ist, die von *TFrameControl* in *FHandle* erwartet wird, also mit *QFrame*. Tatsächlich ist *KHTMLView* von *QScrollView* abgeleitet und diese wiederum von *QFrame*. Also weist *KHTMLView* alle Eigenschaften von *QFrame* auf und kann an *TFrameControl.FHandle* zugewiesen werden. Wenn *KHTMLView* eine inkompatible Klasse wäre, dürfte unsere Komponente *nicht* von *TFrameControl* abgeleitet werden. Notfalls ist aber immer eine direkte Ableitung von *TWidgetControl* möglich, da diese nur ein *QWidget* voraussetzt und alle Widgets in Qt und KDE letztlich von *QWidget* erben.

Bemerkung: Die letzten drei Zeilen dieser Methode hätten natürlich auch schon in einer C-Funktion zusammengefasst werden können, dann hätte unsere C-Interface-Bibliothek nur noch zwei Funktionen gehabt.

Einklinken in das KDE-Anwendungs-Framework *R77*

Das Einbinden von KDE-Widgets hat jedoch noch einen Haken, aufgrund dessen die oben beschriebene Komponente nicht einfach so funktionieren kann: *KHTMLView* und *KHTMLPart* erwarten nämlich, als Teil einer KDE-Anwendung gestartet zu werden. Technisch gesehen handelt es sich dabei um eine Anwendung, die von einer Klasse *KApplication* verwaltet wird, wobei es sich um eine Verfeinerung von *QApplication* handelt. Das in einer Kylix-Anwendung waltende *Application*-Objekt ist nun aber nur mit einem *QApplication*-Objekt verbunden.

Glücklicherweise hat Borland jedoch einen Weg vorgesehen, dieses durch ein *KApplication*-Objekt zu ersetzen, und zwar können Sie, noch vor dem Zeitpunkt, an dem normalerweise das *QApplication*-Objekt erzeugt wird, die Funktionsvariable *LoadThemeHook* der Unit *Qt* auf eine selbst geschriebene Funktion setzen. Diese Funktion muss dann eine ebenfalls selbst geschriebene C-Bibliothek laden, welche zwei Funktionen bereitstellt, mit denen ein *KApplication*-Objekt erzeugt und freigegeben werden kann. Diese Funktionen müssen die Namen *KApplication_create* und *KApplication_create* tragen und sich an das folgende, in der Unit *QForms* deklarierte Aufrufformat halten:

```
type
  TKApp = function (Argc: Integer; Argv: PPChar;
                    AppName: PChar): ApplicationH; cdecl;
  TKAppDestroy = procedure (App: QApplicationH); cdecl;
```

In C lassen sich diese beiden geforderten Funktionen wie folgt implementieren:

```
#include <kapp.h>

extern "C" {

QApplication *KApplication_create(int Argc, char **Argv, char *AppName)
{
  return new KApplication(Argc, Argv, AppName);
}

void KApplication_destroy(QApplication *App)
{
  delete App;
}

} // extern "C"
```

ür das Beispielprogramm wurden diese beiden Funktionen der Einfachheit halber zusammen mit den drei anderen C-Funktionen in *khtml_wrapper.cpp* aufgenommen, sind also Teil der Bibliothek *libkhtmlintf.so*, deren Kompilierung bereits erläutert wurde.

Nun geht es noch darum, die Variabe *LoadThemeHook* mit einer Funktion zu versorgen, die die Bibliothek mit den *create*- und *destroy*-Funktionen enthält. *LoadThemeHook* muss schon gesetzt werden, bevor die *initialization*-Teile der CLX-Units ausgeführt werden, denn schon darin wird sonst das *QApplication*-Objekt initialisiert und ein nachträgliches Setzen von *LoadThemeHook* hätte keinen Effekt mehr.

Die einzige Möglichkeit, so früh in den Initialisierungsprozess einzugreifen, ist das Schreiben einer eigenen Unit und das Setzen von *LoadThemeHook* im *initialization*-Teil dieser Unit. Dies ist auch der einzige Zweck der im Folgenden abgedruckten Unit *KdeThemeHook*. Sie enthält nur die mit *LoadThemeHook* zu verknüpfende Funktion und den besagten *initialization*-Teil:

```
unit KdeThemeHook;

interface

uses Libc, Qt;

implementation

function LoadKHooks: pointer;
begin
  Result := dlopen('libkhtmlintf.so', 0);
```

```
end;

initialization
  Qt.LoadThemeHook := LoadKHooks;
end.
```

Wichtig: Damit der Initialisierungsteil dieser Unit vor den CLX-Units ausgeführt wird, sollte diese Unit als erste Unit in die Projektdatei (PROJEKT | QUELLTEXT ANZEIGEN) eines jeden Projekts eingebunden werden, das die *TKdeHtmlView*-Komponente verwendet.

Wenn die CLX nun ihr *Application*-Objekt initialisiert, überprüft sie die Variable *LoadThemeHook*. Wurde sie nicht gesetzt, konstruiert die CLX das standardmäßige *QApplication*-Objekt. Sonst lässt sie durch *LoadThemeHook* die spezielle Bibliothek laden, in der sie nach den beiden Funktionen mit den vorgegebenen Namen sucht. Mit *KApplication_create* erzeugt sie dann statt des *QApplication* ein *KApplication*-Objekt.

Die in obigem Listing verwendete Linux-Funktion *dlopen* ist für das dynamische Laden von Bibliotheken zuständig, auf das wir in Kapitel 5.5.2 noch einmal zurückkommen werden.

KHTML in Kylix

Die fertige Komponente kann nun wie jede andere Komponente auch in Kylix verwendet werden. Das in Abbildung 5.6 gezeigte Testprogramm erzeugt die Komponente noch dynamisch zur Laufzeit, so dass sie nicht in der Kylix-IDE installiert sein muss:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if KHTMLAvailable then begin
    KHV := TKdeHtmlView.Create(Panel1);
    with KHV do begin
      Parent := Panel1;
      Name := 'TestComponentInstance';
      Align := alClient;
    end;
  end else begin
    Panel1.Caption := 'KHTML nicht verfügbar.';
  end;
  FormResize(Sender);
end;
```

Die zweite Browser-Komponente des Beispielprogramms (*TTextBrowser*) wird nicht in *FormCreate* erzeugt, sondern wurde schon zur Entwurfszeit erzeugt. Die oben aufgerufene *FormResize*-Methode (eigentlich für das *OnResize*-Ereignis vorgesehen), stellt die Größe der beiden Browser bzw. ihrer Panel-Elternkomponenten so ein, dass die Fläche

des Fensters genau in zwei Hälften aufgeteilt wird. Über zwei Schalter im Testformular können Sie zwischen einer horizontalen und einer vertikalen Teilung des Fensters umschalten.

5.5.2 Laden von Bibliotheken zur Laufzeit

Externe Funktionen, die über eine *external*-Direktive deklariert wurden, werden beim Start der Anwendung mit der Anwendung verknüpft. Dazu muss der Programm-Loader von Linux die Bibliotheken laden, in denen diese Funktionen enthalten sind. Wenn jedoch nur eine einzige Bibliothek nicht auf dem System installiert wird, kann das Programm nicht geladen werden (siehe auch Kapitel 5.1).

Beispiel TKdeHtmlView *R76*

Wie sich dies vermeiden lässt, wie eine Anwendung also unabhängig von verwendeten Bibliotheken geladen werden kann, soll hier wieder an der Komponente *TKdeHtmlView* demonstriert werden: Statt der festen Einbindung mit einer *external*-Direktive versucht die Komponente zur Laufzeit selbst, die Bibliothek zu laden. Wenn dieser Versuch fehlschlägt, wird einfach kein KDE-Widget erzeugt, aber die Anwendung kann trotzdem weiterlaufen.

Dafür wird die in Kapitel 5.5.1 gezeigte *external*-Deklaration der drei C-Funktionen durch drei Funktions*zeiger* und drei Typdeklarationen, die das Aufrufformat der mit diesen Zeigern verbundenen Funktionen angeben, ersetzt:

```
type
  Tkhtmlview_create = function(Parent: QWidgetH; Name: PChar):
     KHtmlViewH; cdecl;
  Tkhtmlview_getPart = function(view: QWidgetH): KHtmlPartH; cdecl;
  Tkhtmlpart_openUrl = procedure(part: KHTMLPartH; URL: PChar); cdecl;

var
  khtmlview_create: Tkhtmlview_create = nil;
  khtmlview_getPart: Tkhtmlview_getPart = nil;
  khtmlpart_openUrl: Tkhtmlpart_openUrl = nil;
  khtmlintf: pointer;
```

Die Typdeklarationen entsprechen dabei den bereits bekannten C-Funktionen, die Funktionszeiger werden als globale Variablen deklariert und mit *nil* initialisiert.

Beim Programmstart versucht die Unit der Komponente (*KdeHtmlView.pas*) nun, die C-Bibliothek zu laden und darin die drei Funktionen zu finden, und zwar mit Hilfe der *Libc*-Funktionen *dlopen* und *dlsym*. Beide erwarten im Wesentlichen nur die Namen der Bibliothek bzw. der zu ladenden Funktionen als Parameter.

```
unit KdeHtmlView;

interface

uses
  Qt, Libc, SysUtils, Types, Classes, QGraphics,
  QControls, QForms, QDialogs;

...
implementation
...
// Initialisierungsteil am Ende der Unit:
initialization
  khtmlintf := dlopen('libkhtmlintf.so', 0);
  if khtmlintf <> nil then begin
    khtmlview_create := dlsym(khtmlintf, 'khtmlview_create');
    khtmlview_getPart := dlsym(khtmlintf, 'khtmlview_getPart');
    khtmlpart_openUrl := dlsym(khtmlintf, 'khtmlpart_openUrl');
  end;
end.
```

Wenn die Bibliothek nicht geladen werden konnte, gibt *dlopen nil* zurück, wenn die Funktionen fehlen, zeigt *dlsym* dies mit einer *nil*-Rückgabe an. Da diese *nils* in globalen Variablen gespeichert werden, kann *TKdeHtmlView* später feststellen, dass die externen Funktionen nicht verfügbar sind, und entsprechend auf die Konstruktion des Widgets verzichten:

```
procedure TKdeHtmlView.CreateWidget;
begin
  if not Assigned(khtmlview_create) then
    inherited // Erzeugung eines QFrame durch TFrameControl
  else begin
    FHandle := khtmlview_create(ParentWidget, PChar(Name));
    KdeIntfAvailable := True; // neue Variable von TKdeHtmlView
  end;
end;
```

Auch *SetUrl* wird ein wenig erweitert, damit keine undefinierten Funktionen aufgerufen werden:

```
procedure TKdeHtmlView.SetUrl(Url: String);
var
  part: KHtmlPartH;
begin
  if KdeIntfAvailable then begin
    ...
  end;
end;
```

Eine Sache beibt noch zu bedenken: Der oben gezeigte Aufruf sucht mit *dlopen* ja lediglich die selbst definierte Bibliothek *libkhtmlintf.so*. Und diese sollte immer vorhanden sein, da sie ja quasi zur Anwendung dazugehört. Sollte *dlopen* nicht vielmehr versuchen, die KDE-Bibliothek *libkhtml.so* mit dem HTML-Widget zu laden? Die Antwort ist nein«, denn ein explizites Überprüfen, ob *libkhtml.so* vorhanden ist, erübrigt sich, da der Linux-Loader beim Laden von *libkhtml_wrapper.so* bereits nach dieser und aller weiteren benötigten Bibliotheken sucht. Fehlt irgendeine davon, kann der Linux-Loader auch *libkhtml_wrapper.so* nicht laden und die Klasse *TKdeHtmlView* akzeptiert, dass sie das gewünschte Widget nicht erzeugen kann.

Hinweis: Bei der in Kapitel 5.5.1 besprochenen Erzeugung des *KApplication*-Objekts geht die CLX nach demselben Muster vor. Den Aufruf von *dlsym* für *KApplication_create* und *KApplication_destroy* finden Sie im Quelltext der Unit *QForms* in der Methode *InitDynaProcs*.

6 Dokumentbasierte Anwendungen

Kylix ist ein ganz allgemein einsetzbares Entwicklungssystem, in dem nahezu beliebige Anwendungen aufgebaut werden können. Datenbankanwendungen sind nur ein kleiner Teil des möglichen Spektrums und so beschäftigt sich dieses Kapitel nun – aufbauend auf den Grundlagen der Kapitel 3 und 4 – mit einer größeren Beispielanwendung, die ihre Daten selbst in Dokumenten verwaltet. Dabei kommen zahlreiche neue Themen zur Sprache, die natürlich auch beim Entwickeln von Datenbankanwendungen interessant sein können, wie etwa das Management von Symbolleisten, die geräteunabhängige Grafikausgabe für Bildschirm und Drucker, Programmierung von Mauszieh-Aktionen und Drag&Drop, MDI-Anwendungen und XML.

Allerdings sollten Sie nicht erwarten, dass dieses Kapitel Ihnen den TreeDesigner vollständig erklären kann. Schon alleine das komplette Listing seiner Object-Pascal-Dateien würde mehr als 150 Seiten in Anspruch nehmen. Sie sollten den TreeDesigner daher eher als eine besondere Attraktion der CD-ROM ansehen denn als ein (Lern-) Ergebnis dieses Kapitels. Dieses Kapitel orientiert sich an wichtigen Themen und verwendet den TreeDesigner, um eine von vielen Möglichkeiten der praktischen Umsetzung zu zeigen. Damit die wesentlichen Aspekte nicht untergehen, sind die Listings gegenüber dem Original-Quelltext auf der CD-ROM teilweise gekürzt worden. Die Fokussierung auf die wesentlichen Aspekte soll es Ihnen auch erleichtern, diese in eigene Anwendungen zu übernehmen.

Wenn Ihnen dies jedoch nicht ausreichen sollte, sondern Sie die Arbeitsweise speziell des TreeDesigners im Detail durchschauen wollen, so sind die Listings dieses Kapitels nur die erste Stufe zu diesem Verständnis. Als zweite Stufe bietet es sich an, den TreeDesigner in die Kylix-IDE zu laden, in den Sie interessierenden Methoden die Erweiterungen gegenüber den Buch-Listings zu sichten und mit dem Debugger den Ablauf des Programms zu verfolgen.

Die Abschnitte dieses Kapitels müssen nicht in der vorgegebenen Reihenfolge durchgelesen werden. Allerdings kann es sehr hilfreich sein, wenn Sie sich zuerst mit dem groben Aufbau des TreeDesigners und dessen Bedienung vertraut machen. Hierfür bietet sich das Kapitel 6.1.3 zur Lektüre an.

6.1 Der TreeDesigner

Der charakteristischste Unterschied zwischen der Beispielanwendung dieses Kapitels, dem TreeDesigner, und den anderen Beispielprogrammen des Buches besteht darin, dass der TreeDesigner ein Dokument bearbeitet, das Sie in verschiedenen Ansichten am Bildschirm darstellen, ausdrucken und in Dateien speichern können, während die dialogorientierten Applikationen der anderen Kapitel eher Utility-Funktionen haben oder »fremde« Daten bearbeiten, die Sie nicht selbst speichern müssen (dies gilt vor allem für die Datenbankanwendungen). Diese beiden Anwendungstypen zeigen auch in der Entwicklung Unterschiede.

Entwicklung dialogorientierter Anwendungen

Mit den visuellen Werkzeugen der Kylix-IDE lassen sich komplexe Benutzerschnittstellen mit vielen Dialogboxen und Eingabeformularen sehr leicht entwerfen. Für bestimmte Anwendungstypen genügen die in den bisherigen Kapiteln besprochenen Grundlagen bereits – so z.B. für Programme, die eine einfache Aufgabe zu erfüllen haben, die der Benutzer in einem komfortablen Dialog genauer beschreiben kann. Auch Datenbankanwendungen sind meistens stark dialogorientiert (jedoch benötigen Sie für diese auch die in Kapitel 7 beschriebenen Grundlagen).

Entwicklung dokumentbasierter Anwendungen

Der andere Anwendungstyp soll an dieser Stelle als *dokumentbasiert* bezeichnet werden. Allein schon das Dokument bringt einige Aufgaben mit sich, die in Kylix nicht visuell gelöst werden können:

- Um die im Dokument enthaltenen Daten effizient verwalten zu können, benötigt das Programm entsprechende Datenstrukturen. Bei einfachen Texteditoren können diese aus einer Stringliste bestehen, wie sie von *TMemo* zurückgeliefert wird, die meisten Anwendungen müssen jedoch ihre eigene Datenstruktur definieren.
- Das Speichern und Laden der Daten von und zur Datei wird zwar meistens durch ein im Objektinspektor stehendes Event veranlasst, muss aber ansonsten ohne Komponentenhilfe auskommen (zumindest bieten die Komponenten, die sich im Lieferumfang von Kylix befinden, noch keine Unterstützung dafür).
- Die Darstellung des Dokuments ist zwar für den Benutzer der Anwendung visuell, kann aber nicht zur Entwurfszeit festgelegt werden, sofern Sie die Grafikausgabemethoden von *TCanvas* aufrufen und keine der vorgefertigten (Grafik-)Komponenten benutzen.

Darüber hinaus enthält die Benutzerschnittstelle einer dokumentbasierten Anwendung oft Merkmale, die für ein dialoggesteuertes Tool teilweise gar nicht benötigt werden:

- Viele Anwendungen erlauben dem Benutzer, mehrere Dokumente gleichzeitig zu bearbeiten, und brauchen entsprechend eine geeignete Schnittstelle zur Handhabung mehrerer Dokumente, wie etwa das Multi Document Interface (MDI). Mit Kylix ist es besonders einfach, aus einer gut entworfenen SDI-Anwendung (Single Document Interface) eine MDI-Anwendung zu machen.
- Bei der Bearbeitung des Dokuments verarbeitet die Anwendung die Eingaben des Benutzers direkter als eine dialogorientierte Anwendung. Während letztere relativ komplexe Ereignisse wie *OnClick* bearbeitet, muss eine dokumentbasierte Anwendung die zwei oder drei Bestandteile des *OnClick*-Ereignisses oft einzeln betrachten: »Taste drücken«, optional »Maus bewegen« und abschließend »Taste loslassen«. Ein weiteres Beispiel ist die Tastatureingabe: In dialogorientierten Anwendungen findet sie oft automatisch in Editierfeldern statt, so dass Sie nur noch das Ergebnis abfragen müssen, z. B. über *Edit1.Text*. Bei der Dokumentbearbeitung müssen Sie die Tastaturereignisse vielleicht selbst bearbeiten.

6.1.1 Wahl eines Beispielprogramms

Für unser Beispielprogramm benötigen wir als Erstes eine Aufgabe, in diesem Fall also den Typ des zu bearbeitenden Dokuments. Bei genauerer Betrachtung der elementaren Anwendungstypen Textverarbeitung, Tabellenkalkulation und Grafikprogramm stellt sich schnell heraus, welcher für unsere Beispielanwendung der geeignete ist:

- Textverarbeitungsprogramme, die mehr können als die *Memo*-Komponente, erfordern so viel Aufwand für die Datenverwaltung, dass sie eher in Bücher über allgemeine Programmiertechnik gehören.
- Für Tabellenkalkulationsprogramme gilt das Gleiche, sobald neben nackten Zahlen auch Ausdrücke in den Zellen stehen sollen, die geparst werden wollen.
- Grafikprogramme erlauben auch ohne komplizierte Algorithmen bereits den Aufbau von sinnvollen Dokumenten, wie in Abbildung 6.1 gezeigt.

Unser Beispielprogramm wird also ein Grafikprogramm, das die Grafik aus einzelnen, frei skalierbaren Objekten wie Rechtecken, Kreisen, Ellipsen, Linien und anderen Formen wie Rauten und Sechsecken, die jeweils durch einen zentrierten Text beschriftet werden können, zusammensetzt.

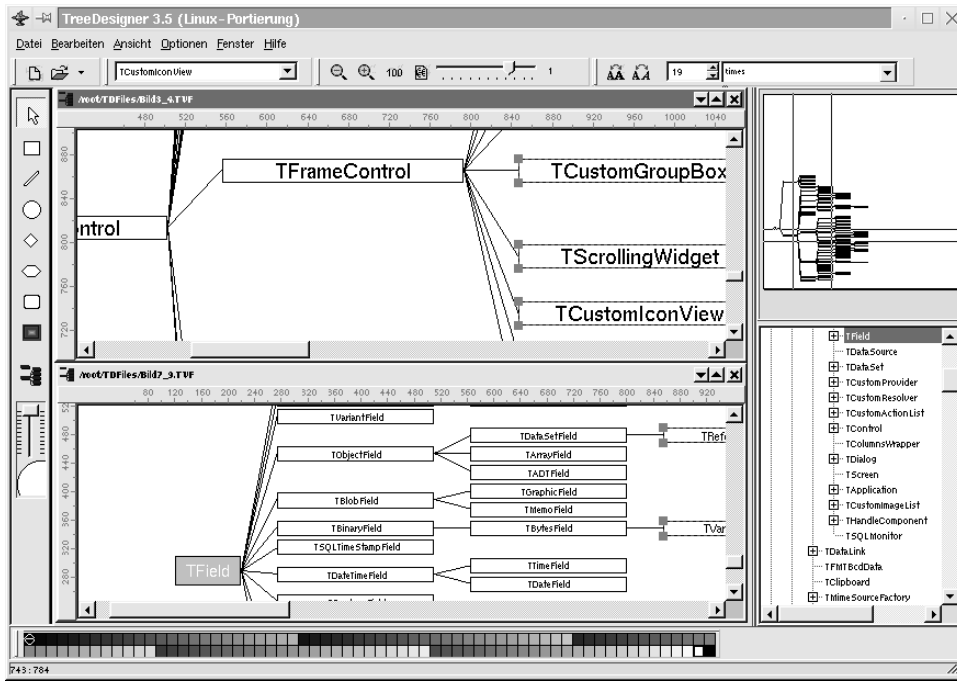


Abbildung 6.1: Der TreeDesigner mit zwei geöffneten Dokumenten und am rechten Rand angedockten Hilfsfenstern.

6.1.2 Spezielle Fähigkeiten

Da unsere Anwendung als allgemeine Grafikanwendung kaum mit professionellen Paketen konkurrieren kann, werden einige spezielle Funktionen eingebaut, die ihre Benutzung dennoch lohnend machen. So können Sie zwischen den grafischen Objekten Verbindungen definieren, die vom Programm automatisch durch Verbindungslinien dargestellt werden. Gegenüber handgezeichneten Verbindungslinien gibt es mehrere Vorteile:

- Wenn Sie ein Objekt verschieben, wird die Verbindung automatisch angepasst,
- oder alle verbundenen Objekte werden ebenfalls verschoben.
- Eine so verbundene Struktur kann vom Programm als Baum interpretiert werden,
- der mit etwas zusätzlichem Code sogar automatisch erzeugt werden kann.

Abbildung 6.1 zeigt auch den *TreeExplorer*, der alle Grafikelemente des aktuellen Dokuments in einem TreeView darstellt und Ihnen die Navigation innerhalb einer größeren Grafik erleichtert, ähnlich wie der CodeExplorer der Kylix-IDE es für die Pascal-Quelltexte vormacht.

Anzeige von Klassenhierarchien

Insbesondere die Fähigkeit der automatischen Baumerzeugung ist für Kylix-Programmierer interessant, da durch sie eine Alternative zu der Hierarchiedarstellung des in Kylix integrierten Browsers entsteht: Während die Wurzel des Hierarchiebaumes in diesem ganz oben steht und alle weiteren Klassen nur untereinander (mit etwas Einrückung versehen) angeordnet sind, lässt sich die grafische Baumdarstellung innerhalb des Grafikprogramms vielfältig ändern, zurechtrücken, farbig markieren und auf diese Weise übersichtlicher gestalten und sogar ausdrucken.

Lieferant für die Daten der Klassenhierarchie ist ein kleines Shell-Skript, das in Kapitel 5.3.3 vorgestellt wird und eine Liste von Klassenpaaren (abgeleitete Klasse und Basisklasse) in eine Textdatei schreibt. Die haben zwei Möglichkeiten, diese Daten in den TreeDesigner zu bringen:

- Wenn Sie das Tool *RunPrograms* aus Kapitel 5.3.3 geöffnet haben, können Sie die über das Shell-Skript gefundenen Klassendaten aus dem Memo-Feld in die Zwischenablage kopieren und dann im TreeDesigner den Menüpunkt BEARBEITEN | KLASSENHIERARCHIE EINFÜGEN oder die entsprechenden Aktionsschalter wählen.
- Sie können die vom Shell-Skript erzeugten Dateien aber auch direkt, ohne *RunPrograms*, in den TreeDesigner laden, indem Sie dessen Menüpunkt DATEI | TEXTDATEIEN IMPORTIEREN | EINZELNEN-TEXTDATEI ausführen.

Die auf diese Weise geladenen Grafiken werden automatisch zu einem Baum angeordnet. Sie können das Anordnen aber auch von Hand steuern, indem Sie den gewünschten Wurzelknoten markieren (z.B. über einen Doppelklick auf den Knotennamen im TreeExplorer) und dann den Anordnen-Befehl (BEARBEITEN | BAUM NEU ANORDNEN) ausführen. Wenn Sie sich für eine andere Wurzel als *TObject* entscheiden, erhalten Sie einen möglicherweise sehr übersichtlichen Teilbaum aus der Klassenhierarchie. Natürlich können Sie auf dieselbe Weise auch von der CLX unabhängige, selbst erstellte Baumstrukturen anordnen.

Hinweis: Die CD-ROM enthält bereits eine fertige TreeDesigner-TVF-Datei mit der Grafik der CLX-Hierarchie. Diese können Sie im TreeDesigner mit DATEI | ÖFFNEN... laden.

6.1.3 Kurzbeschreibung und Bedienung

Wie bereits angedeutet, handelt es sich beim TreeDesigner um ein Grafikprogramm, in dem Sie mit einfachen grafischen Formen wie Linien, Rechtecke, Ellipsen und mit spezielleren Formen arbeiten, die jeweils mit einem zentrierten Text beschriftet werden können. Zwischen einzelnen Objekten können Sie Verbindungen definieren, die automatisch gezeichnet werden. Die Arbeitsfläche des TreeDesigners ist ein Zeichenblatt,

das per Voreinstellung eine quadratische Form hat, die mit 3000*3000 Pixeln weit über den Bildschirm hinausgeht und daher innerhalb des Fensters gescrollt werden kann.

Falls Sie eines der Delphi-Bücher mit einer älteren TreeDesigner-Version besitzen, werden Sie sich möglicherweise für die Unterschiede zwischen den Versionen interessieren, die folgende Aufstellung zeigt die Neuerungen seit der Version 2.0:

- Die auffälligste Erweiterung der Version 3.5 ist die Aufbereitung einer XML-Datei in einer grafischen Baumstruktur (siehe Abbildung 6.2). Dafür war es notwendig, einen neuen Typ von Grafikelement einzuführen, der auch mehrzeiligen Text mit automatischem Zeilenumbruch darstellen kann. Die Grafikelemente nutzen nun die Polymorphie, damit verschiedene Elemente verschiedener Klassen in Dateien gespeichert werden können. Damit kann nun auch das Verlaufselement (ein Rechteck, das mit einem Farbverlauf gefüllt wird) in Dateien gespeichert werden. Eine zweite XML-Erweiterung des TreeDesigners besteht darin, dass dieser auch seine eigenen Daten (bisher in binären `.tvf`-Dateien abgelegt) in XML-Dateien schreiben kann (in den Grafikelementen angezeigte Bilder werden *nicht* in der XML-Datei gespeichert).
- Die Version 3.0 brachte gegenüber der Version 2.5 folgende Erweiterungen: Ein Hilfsfenster gibt eine Übersicht über das aktuelle Dokument und zeigt, welcher Ausschnitt gerade im Dokumentfenster sichtbar ist. Durch das Dokument-View-Konzept erlaubt der TreeDesigner seit dieser Version die Bearbeitung eines Dokuments in mehreren gleichzeitig geöffneten Dokumentfenstern. Außerdem wurden die Lineale optimiert, die Handhabung des TreeExplorers verbessert, einfache Einstellungsdialoge für die einzelnen Grafikobjekte hinzugefügt und weitere Verbesserungen eingebaut.
- Neu in der Version 2.5 waren unter anderem der *TreeExplorer* (ein TreeView mit allen Grafikelementen des aktuellen Dokuments), die Möglichkeit, neben Text auch Bilddateien in einzelne Grafikelemente einzublenden, und die Griffe, mit denen Sie Grafikobjekte wie vom Formularentwurf in der Delphi-IDE bekannt in der Größe ändern können.

Mausaktionen

Zum Zeichnen von neuen Objekten wählen Sie aus der Werkzeugleiste den Schalter mit der gewünschten grafischen Form aus und zeichnen das Objekt mit der Maus auf die Arbeitsfläche ein. Solange der entsprechende Schalter in der Werkzeugleiste gedrückt ist, können Sie nur neue Objekte einzeichnen, aber keine bestehenden Objekte verändern. Diesen Zeichenmodus können Sie auch daran erkennen, dass der Mauszeiger die Form eines Fadenkreuzes angenommen hat.

Wenn Sie in der Werkzeugleiste den Schalter mit dem Mauszeigersymbol wählen, werden keine neuen Objekte gezeichnet, sondern Sie können die bestehenden Objekte manipulieren:

- Zum Verschieben ziehen Sie das Objekt einfach mit der Maus an seinen neuen Ort.
- Zum Markieren klicken Sie es mit der Maus an, zusammen mit ⇧ können Sie mehrere Objekte auswählen.
- Um einen Bereich von Objekten zu markieren, zeichnen Sie mit der Maus ein Rechteck um die zu markierenden Objekte. Sie dürfen die Maus dazu jedoch nicht über einem Objekt drücken, da Sie dieses sonst verschieben.
- Um die Größe eines markierten Objekts zu verändern, gehen Sie so vor wie beim Verändern der Größe von Komponenten im Formulareditor der Kylix-IDE: Ziehen Sie an einem der »Griffe«, die im Rahmen des Objekts gezeichnet werden.
- Um Objekte miteinander zu verbinden, markieren Sie einige »Zielobjekte« und klicken, während Sie Strg halten, auf das Objekt, von dem die Linien zu den Zielobjekten ausgehen sollen (mehr über diese Funktion können Sie in Kapitel 6.3.4 lesen).
- Um ein Objekt zu verschieben, ohne dass der gesamte von diesem Objekt ausgehende Teilbaum ebenfalls verschoben wird, drücken Sie ⇧, bevor Sie mit dem Verschieben beginnen.

Tastaturaktionen

Unabhängig von der Einstellung des Zeichenwerkzeugs können Sie zwei Tastaturbefehle einsetzen (sofern der Eingabefokus sich nicht in einem Eingabeelement der Symbolleisten befindet):

- Um Objekte zu löschen, markieren Sie sie und drücken Entf.
- Um das oberste Grafikelement hinter alle anderen Elemente zu verschieben, drücken Sie Bild ↓. Diese Funktion können Sie auch dazu verwenden, durch alle vorhandenen Grafikelemente zu blättern, denn der TreeDesigner rückt das neue oberste Element automatisch in den sichtbaren Fensterbereich.

Editieren der Attribute

Sie können bei jedem Objekt die folgenden Attribute verändern: Linien-, Füll- und Schriftfarbe, Schriftart und -größe, die Form des Objekts sowie den Beschriftungstext.

Alle Attribute können Sie über die Symbolleisten wie folgt einstellen: Markieren Sie zuerst die Objekte, die Sie verändern wollen, und stellen Sie dann die Attribute ein:

- Geben Sie einen Beschriftungstext in das Editierfeld ein.
- Wählen Sie eine Schriftart aus der aufklappbaren Liste aus.
- Geben Sie neben dieser Listbox die Größe der Schrift ein. Über die beiden zusätzlichen Schalter können Sie auch Fett- und Kursivschrift einstellen.
- Um die Dicke der Linien zu verändern, verschieben Sie den Regler am unteren Ende der vertikalen Werkzeugleiste.
- Um die Form der gewählten Objekte zu ändern, doppelklicken Sie auf den Schalter für die neue Form.

ür die Farben stellt der TreeDesigner eine Farbpalette zur Verfügung, in der drei Farben markiert sind (siehe Abbildung 6.2):

- Die mit einer Linie markierte Farbe wird zum Zeichnen der Linie verwendet (markieren Sie diese Farbe mit der linken Maustaste).
- Die Farbe, die mit einem Kreis versehen ist, findet sich zum Füllen des Objekts wieder (Markieren mit rechter Maustaste).
- Und die Farbe mit der »T«-Markierung liefert die Farbe für die Beschriftung (Markieren mit ⇧+linker Maustaste).

Die Füllungsfarbe der Objekte können Sie auch per Drag&Drop ändern, indem Sie die gewünschte Farbe mit der linken Maustaste auf das Objekt ziehen (dabei sollten keine Objekte markiert sein, da sonst deren Linienfarbe geändert würde).

Die Koordinaten eines Elements können Sie auch dialoggesteuert einstellen, wenn Sie das Element mit der rechten Maustaste anklicken und den Popup-Menüpunkt ATTRIBUTE... ausführen (es werden dann nur die Attribute des angeklickten Elements geändert, auch wenn ein anderes Element markiert sein sollte).

Alle Einstellungen in den Symbolleisten werden auch zum Zeichnen eines neuen Objekts verwendet, mit Ausnahme des Beschriftungstextes, der nach einmaliger Verwendung wieder gelöscht wird (allerdings können Sie mehrere Objekte markieren und deren Beschriftung gleichzeitig verändern).

Unter dem Regler für die Linienstärke befindet sich außerdem ein kleines Vorschaufeld, in dem Sie aktuelle Hintergrundfarbe, Linienfarbe und Linienbreite erkennen önnen.

Standardaktionen

Der TreeDesigner verfügt natürlich über die üblichen von MDI-Anwendungen gewohnten Merkmale wie die Funktionen des Fenstermenüs, das Speichern und Laden von Dateien (das Format der vom TreeDesigner erzeugten Dateien wird durch

die Endung `.tvf` gekennzeichnet) und über die typischen Zwischenablageoperationen BEARBEITEN | AUSSCHNEIDEN, BEARBEITEN | KOPIEREN und BEARBEITEN | EINFÜGEN. AUSSCHNEIDEN und KOPIEREN beziehen sich dabei jeweils auf alle im aktuellen Dokument markierten Objekte.

Auch für diese Zwischenablage-Operationen verwendet der TreeDesigner sein eigenes Format, das heißt, Sie können aus dem TreeDesigner kopierte Objekte nicht in anderen Anwendungen einfügen und umgekehrt keine Objekte aus anderen Anwendungen in den TreeDesigner übernehmen – zumindest nicht über dieses Menü. Innerhalb eines einzelnen TreeDesigner-Grafikelements können Sie jedoch wie folgt Grafiken aus anderen Quellen anzeigen.

Anzeige von Grafiken innerhalb der Objekte

Im Popup-Menü der einzelnen Grafikelemente finden Sie unter anderem drei Punkte, die sich mit dem Inhalt des Elements befassen:

- GRAFIK AUS DATEI LADEN... öffnet einen Dialog zur Auswahl einer Grafikdatei, deren Inhalt daraufhin im Grafikelement angezeigt wird.
- GRAFIK AUS ZWISCHENABLAGE EINFÜGEN fügt den Grafikinhalt der Zwischenablage in das aktuelle Grafikelement ein. Das Format muss eines der von der CLX unterstützten Grafikformate sein, siehe hierzu Kapitel 4.7.2.
- INHALT LÖSCHEN stellt den Ursprungszustand des Grafikelements wieder her, eine eventuell mit den beiden vorher genannten Funktionen eingefügte Grafik wird also gelöscht.

Da zur Verwaltung der Grafik ein *TPicture*-Objekt verwendet wird, können Sie alle Grafikdateien laden, die mit *TPicture* kompatibel sind. Durch jedes Einfügen eines neuen Elementinhalts wird der bisherige Inhalt gelöscht. Die Grafik wird per *StretchDraw* gezeichnet und füllt die rechteckigen Umrisse eines Grafikelements immer voll aus, dabei findet auch bei nicht-rechteckigen Elementen kein Clipping statt. Beim Speichern des Dokuments wird die Grafik über *TPicture.SaveToStream* in die TVF-Datei geschrieben, wird also zu einem fest integrierten Bestandteil des TreeDesigner-Dokuments.

Diese Funktionen sind hauptsächlich zur Demonstration gedacht, wie durch relativ kleine Erweiterungen am Programmcode andere Inhalte innerhalb der einzelnen Grafikelemente angezeigt werden können. Die Implementierung dieser Funktionen kann hier aus Platzgründen nicht erläutert werden, aber sie wurde über eine eigene Unit namens *GrCont* mit der Absicht entworfen, leicht nachvollziehbar und auf andere Typen von Grafiken erweiterbar zu sein.

Hinweis: Da die neue Version des TreeDesigners auch die Speicherung abgeleiterer Grafikelement-Klassen unterstützt, ist es unter Umständen empfehlenswert, eine neue Klasse von *TGraphicElement* abzuleiten, statt nur den Inhalt eines Elements einer bestehender Klasse zu definieren, wie von der Unit *GrCont* vorgesehen.

Ansichten und Drucken

ber den Regler in der Mauspalette können Sie die Vergrößerung, mit der die Grafik angezeigt wird, zwischen 0,1 und 10 einstellen. Dabei versucht der TreeDesigner, je nach Zoom-Modus den derzeitigen Mittelpunkt oder die markierten Objekte auch nach der Vergrößerung/Verkleinerung zentriert darzustellen.

Der Druckdialog lässt Ihnen die Wahl zwischen zwei verschiedenen Druck-Modi:

- Im Modus *1 Einheit = 0,1 mm* wird der gesamte Bereich des Arbeitsblatts gedruckt. Mit dem Menübefehl ANSICHT | SEITENANZEIGE können Sie das Arbeitsblatt auf die zurzeit im Druckertreiber gewählte Seitengröße »zurechtschneiden«. Um es ungefähr in der Größe des Drucks auf dem Bildschirm zu sehen, müssen Sie beispielsweise bei 17-Zoll-Monitoren den Vergrößerungsfaktor 0,2 einstellen.
- Im Modus *Druckgröße anpassen* können Sie den Ausdruck frei auf der Seite positionieren und skalieren. Siehe hierzu Kapitel 6.6.3.

Anpassen der Benutzeroberfläche

Zur Laufzeit haben Sie beim TreeDesigner einige Möglichkeiten, die Oberfläche des Programms anzupassen:

- Die Fenster von TreeExplorer und Übersichtsfenster können Sie auf der rechten Seite des Hauptfensters andocken, indem Sie den Menüpunkt MANUELLES DOCKING aus dem Popup-Menü des anzudockenden Fensters auswählen (die Bezeichnung »manuell« beschreibt dieses Docking aus der Sicht des Programmierers, der das Docking selbst programmieren muss – im Vergleich zum automatischen Docking, das in der Windows-Version des TreeDesigners durch die VCL durchgeführt wird, mit der aktuellen CLX-Version aber noch nicht möglich ist).
- Das Popup-Menü zu den Linealen hat nur einen Menüpunkt, mit dem Sie die Schriftart anpassen können, die in den Linealen verwendet wird.
- Im Popup-Menü des TrackBars wählen Sie zwischen zwei verschiedenen Modi für die Skalierung des Zoom-Reglers (siehe Kapitel 6.6.2) und zwischen drei zusätzlichen Zoom-Modi, die festlegen, ob beim Zoomen auf die Bildmitte, auf die Markierung oder überhaupt nicht fokussiert werden soll.

- Mit OPTIONEN | TASTENKÜRZEL... gelangen Sie in den Tastenkürzeleditor aus Kapitel 4.6.3, in dem Sie jedem der Hauptmenüpunkte ein anderes Tastenkürzel zuweisen können.

Eine Übersicht über die Dialogfenster des TreeDesigners finden Sie in Kapitel 6.1.4.

Der TreeExplorer

Der TreeExplorer listet alle Grafikelemente des aktuellen Dokuments anhand ihrer Beschriftung auf, wobei verbundene Elemente hierarchisch dargestellt werden. Er ist am rechten Rand des Hauptfensters andockbar. Sein Inhalt wird nur dann automatisch aktualisiert, wenn er aktiviert wird und seit seiner letzten Aktualisierung ein anderes Dokumentfenster aktiviert wurde. Durch einen Doppelklick in den TreeView-Bereich ist jedoch jederzeit eine manuelle Aktualisierung möglich.

Gegenüber der Version 2.5 des TreeDesigners wurde die Handhabung des TreeExplorers weiter auf die Bearbeitung von VCL/CLX-Hierarchiegrafiken optimiert:

- Gleich nach der Aktivierung des TreeExplorer-Fensters wird in diesem automatisch ein Knoten markiert, der im aktuellen Grafikdokument sichtbar ist. Dies ist wichtig, wenn viele Knoten in einer Grafik unsichtbar sind, weil sie einem ausgeblendeten Teilbaum angehören.
- Durch einen Doppelklick auf einen Baumknoten markieren Sie das entsprechende Grafikelement im Dokumentfenster – vorausgesetzt, dass dieses durch eine eindeutige Beschriftung identifizierbar ist.
- Bei einer normalen Auswahl (mit Einfachklick) eines TreeView-Elements wird das entsprechende Element (falls eindeutig beschriftet) im Zeichenfenster nur sichtbar gemacht, aber nicht markiert. Wenn Sie einen unsichtbaren Knoten wählen, erhalten Sie einen entsprechenden Hinweis in der Statuszeile.
- Umgekehrt können Sie ein Grafikelement im Zeichenfenster markieren und dieses automatisch im TreeView hervorheben lassen. Hierzu dient ein entsprechender Punkt im Popup-Menü des TreeExplorers.

Das Übersichtsfenster

Ein weiteres andockbares Hilfsfenster ist das Übersichtsfenster. Es bezieht sich ebenfalls immer auf das aktuelle Dokument und zeigt dieses in einer Gesamtansicht an (der Dokumentinhalt wird also auf die Größe dieses Fensters verkleinert). Hilfslinien in der Übersicht markieren außerdem den im Dokumentfenster sichtbaren Bereich. Das Übersichtsfenster wird immer automatisch aktualisiert (bei jeder Änderung im aktuellen Dokument, beim Wechsel zwischen Dokumentfenstern und beim Scrollen innerhalb dieser Fenster).

Dialoge

Weitere, eventuell noch nicht erwähnte Dialogfenster des TreeDesigners gehen aus der Formular- und Dateiübersicht des nachfolgenden Abschnitts hervor.

6.1.4 Dateien auf der CD

Auf der CD finden Sie zwei verschiedene Programmversionen des TreeDesigners. In der vereinfachten Version (Abbildung 6.2) können Sie immer nur ein Grafikdokument gleichzeitig bearbeiten, es sei denn, Sie starten den TreeDesigner mehrmals (was aber nur außerhalb der Kylix-IDE möglich ist). Wenn Sie also eine neue Grafik laden, ersetzt diese die bisherige Grafik im TreeDesigner-Fenster. Die Ein-Dokument-Version wird im Folgenden auch als SDI-Version (Single Document Interface) bezeichnet. Ihre Projektdatei finden Sie auf der CD-ROM unter dem Namen TreeDesignerSDI.

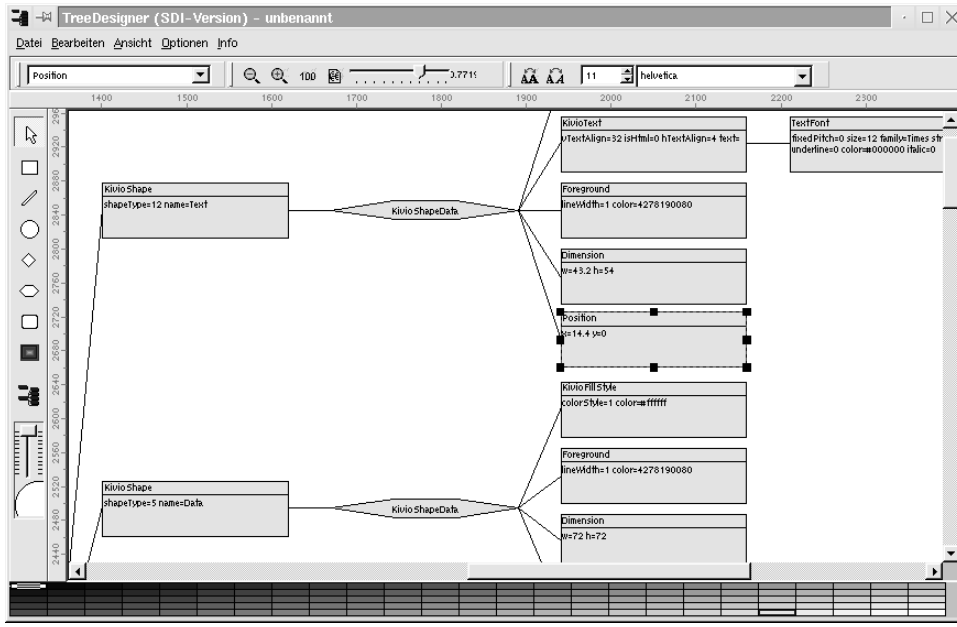


Abbildung 6.2: Die SDI-Version des TreeDesigners mit der grafischen Darstellung einer XML-Datei (einem von der Office-Anwendung Kivio erzeugten Dokument)

Die zweite Version (*TreeDesigner*, siehe Abbildung 6.1) ist eine MDI-Anwendung, in der Sie mehrere Grafiken in eigenen Fenstern bearbeiten können, welche sich alle innerhalb eines übergeordneten Hauptfensters befinden. Die Unterschiede zwischen SDI- und MDI-Version werden in Kapitel 6.7 behandelt, in allen anderen Kapiteln spielt es keine Rolle, ob Sie die MDI- oder die SDI-Version vor sich haben.

Formulare und Programmdateien

Der TreeDesigner beinhaltet in der MDI-Version elf Formulare, die zur Laufzeit als sichtbare Fenster auftreten, drei Formulare bzw. Frames, die durch Vererbung in einigen der elf sichtbaren Formulare erscheinen, sowie ein weiterer Frame, der die Werkzeugleiste am linken Rand des Hauptfensters definiert. Er verwendet sechs der auf der CD mitgelieferten Komponenten, von denen vier in Kapitel 8 entwickelt werden. Zur Laufzeit ständig sichtbar sind die auf den folgenden Formularen basierenden Fenster:

- In der MDI-Version dient das Formular *MainForm* (Unit *MainForm*) als Hauptfenster. Es beinhaltet ein kleines Hauptmenü, das die von den einzelnen Dokumenten unabhängigen Menüpunkte enthält. Außerdem finden Sie in diesem Fenster die Werkzeugleiste, die Farbpalette, Platzhalter für die Symbolleisten der Kindfenster sowie die Andockstelle für den TreeExplorer.
- *TDocumentForm* (Unit *DocForm*) ist die Klasse für das Dokumentfenster. *MainForm* verwendet diese Klasse als Schablone für die MDI-Fenster; in der SDI-Version gibt es nur ein Exemplar dieser Klasse, das dann das Hauptfenster ist (Name: *DocumentForm*). Zur SDI-Version siehe auch den Abschnitt *Verwenden des MDI-Kindfensters als SDI-Hauptformular* in Kapitel 6.7.6.
- *TreeExplorerForm* (Unit *TreeExplorer*) definiert den in Kapitel 6.1.3 beschriebenen, am rechten Rand des Hauptfensters andockbaren TreeExplorer.
- *OverviewForm* (Unit *Overview*) ist das zweite andockbare Hilfsfenster. Es zeigt immer eine Gesamtansicht des aktuellen Dokuments und wurde ebenfalls schon in Kapitel 6.1.3 erwähnt.

Die sieben restlichen Formulare werden als modale Dialoge verwendet:

- *PrintDialog* ist der Dialog, der vor dem Drucken angezeigt wird; er enthält eine Druckvorschau und verschiedene Optionen für die Druckgröße und -position.
- *DocListForm* wird über den Menüpunkt ANSICHT | LISTE DER GEÖFFNETEN DOKUMENTE aufgerufen und listet getreu seines Namens alle Dokumente auf, die in irgendeinem der Dokumentfenster angezeigt werden. Außer den Dateinamen gibt es in der Liste noch an, in wie vielen Fenstern das jeweilige Dokument angezeigt wird (eine eventuelle Anzeige des Dokuments im Übersichtsfenster wird mitgezählt) und ob der Inhalt seit dem letzten Speichern verändert wurde.
- *ObjectAttrDlg* (Unit *AttrDlg1*) stellt einen Dialog dar, in dem Sie die Koordinaten, die Beschriftung und die Form des aktuellen Grafikelements anpassen können. Für die genaue Einstellung der Koordinaten ist dieser Dialog nützlich, ansonsten dient er hauptsächlich als Beispiel für den Einsatz von Frames (siehe Kapitel 3.7).
- *InfoForm* dient zur Anzeige von Informationen zum TreeDesigner und wird wie üblich über das Hilfemenü aufgerufen.

- *ScaleDialog* ist ein modaler Dialog und wird durch den Menüpunkt BEARBEITEN | SKALIEREN aufgerufen. Er enthält nur zwei Editierfelder, in denen Sie angeben, um welchen Faktor die Grafik in Breite und Höhe skaliert werden soll.
- *TreeArrangeOptionsDialog* enthält einige Optionen für die automatische Baumanordnungsfunktion: Abstand zwischen den Hierarchieebenen, Breite und Höhe einer Ebene sowie Ausrichtung des Baumes (horizontal oder vertikal).

Schließlich finden Sie in der Formularliste noch fünf weitere Mitglieder, die mit den in Kapitel 3.7 beschriebenen Vererbungs-Techniken zu tun haben:

- *FrameBasicObjectAttrs* definiert den Frame mit den Dialogelementen zur Einstellung von grundlegenden Objekteigenschaften.
- *ObjectAttrBaseForm* (Unit *AttrDlg0*) definiert die Basisklasse für die Dialoge zum Einstellen der Objekteigenschaften. Die einzigen darin bereits definierten Dialogelemente stammen aus dem Frame *FrameBasicObjectAttrs*. Die zur Laufzeit in Erscheinung tretenden Dialoge, die auf diesem Formular basieren, sind *ObjectAttrDlg* (siehe oben) und *VerlaufObjectAttrDlg*.
- *GradientObjectAttrDlg* (Unit *AttrDlg2*) definiert den Dialog zum Einstellen der Attribute der speziellen Farbverlaufs-Grafikobjekte. Zu weiteren Details bezüglich der Attributdialoge siehe Kapitel 3.7.
- *GlobalToolBar* ist das Frame, das als Werkzeugleiste an der linken Seite des Hauptfensters eingeblendet wird. Das Werkzeugleisten-Panel hätte natürlich ohne Frame auch direkt im Hauptfenster eingezeichnet werden können. Die Entscheidung für die Auslagerung dieses Panels in einen Frame hängt damit zusammen, dass der TreeDesigner sowohl in einer MDI- als auch in einer SDI-Version erscheinen sollte. Dies spielt aber im weiteren Verlauf dieses Kapitels keine Rolle.
- *DockableForm* stellt eine gemeinsame Basisklasse für die Formulare *OverviewForm* und *TreeExplorerForm* dar. Es definiert den in beiden Formularen benötigten Code zum Andocken an den dafür vorgesehenen Rand des TreeDesigner-Hauptformulars (siehe auch Kapitel 6.8.2).

Da die Erläuterung des TreeDesigners sich nicht über das ganze Buch ausdehnen kann und Themen wie Dialoge und TreeViews bereits in anderen Kapiteln erläutert wurden, werden sich die folgenden Abschnitte auf das Haupt- und Dokumentformular konzentrieren. In Kapitel 6.6.3 kommt außerdem der Drucken-Dialog zur Sprache, und das reeExplorer-Fenster wird in Kapitel 6.8.2 noch einmal erscheinen, aber nur in seiner Eigenschaft als Docking-Fenster.

6.2 Das Hauptfenster

Dieses Kapitel beschäftigt sich mit dem groben Aufbau der Fenster des TreeDesigners und behandelt dabei beispielhaft sowohl die zahlreichen Verwendungsmöglichkeiten von Symbolleisten als auch grundlegende Aufgaben von Dokumentformularen aller Art.

Hinweis: In der hier vorausgesetzten MDI-Version des TreeDesigners werden drei Symbolleisten, die zur Laufzeit unter der Menüzeile des Hauptfensters erscheinen, im Dokumentformular definiert. Die besonderen Vorteile dieser Vorgehensweise werden in Kapitel 6.7.5 beim dynamischen Werkzeugleistenmanagement erläutert, in diesem Kapitel spielt es noch keine Rolle, wie die Leisten zur Laufzeit in das Hauptfenster gelangen.

6.2.1 Der Entwurf von Symbol- und Werkzeugleisten

Obwohl es nicht verboten ist, ein Dokumentformular ähnlich wie ein Dialogfenster an beliebigen Positionen mit Steuerelementen anzureichern, werden die benötigten Steuerelemente üblicherweise in Symbol- oder Werkzeugleisten zusammengefasst. Als Basiskomponenten zur Aufnahme der einzelnen Steuerelemente kommen in Kylix vornehmlich zwei Komponentenklassen in Frage: *TToolbar* von der Seite COMMON CONTROLS und *TPanel* von der Standardseite.

Beide dienen den in ihnen enthaltenen Komponenten als Elternfenster, während das Formular der Besitzer der Komponenten ist (Genaueres zu Steuerelement-Gruppen siehe Kapitel 3.5.1). Ein besonderer Vorteil der Elternschaft ist, dass die Komponenten bei jeder Verschiebung des Panels mitbewegt werden – das beinhaltet auch eine vom Anwender zur Laufzeit gewünschte Verschiebung, zum Beispiel vom oberen an den unteren Fensterrand (per Umschaltung des *Align*-Properties von *alTop* auf *alBottom*).

Panels

Um aus einem Panel eine Zeile oder Spalte zu machen, die an einem der vier Fensterränder angeheftet ist und die sich bei jeder Größenänderung des Fensters mit anpasst, müssen Sie lediglich das Panel-Property *Align* auf einen der Werte *alTop*, *alBottom*, *alLeft* oder *alRight* setzen (siehe Kapitel 3.3.1). Drei der Koordinaten werden daraufhin automatisch bestimmt. Im Falle einer Mauspalette (*alTop*) sind das z. B. die Properties *Left*, *Width* und *Top*. Sie können also die Breite und Position eines *alTop*-ausgerichteten Panels nicht mehr verändern, da es sich hier nach seinem Elternfenster, also normalerweise dem Formular, richtet. Lediglich Änderungen der Höhe sind bei *alTop* und *alBottom* noch möglich.

oolBars *R2*

Die Klasse *TToolBar* beinhaltet automatische Mechanismen zur Anordnung von Unterelementen: Grundsätzlich werden die Elemente in einer Zeile von links nach rechts angeordnet, und zwar dicht aneinander. Um Zwischenräume zu erhalten, müssen Sie spezielle Trennelemente einfügen. Zur Entwurfszeit können Sie die einzelnen Elemente außerdem per Drag&Drop an eine andere Position bewegen.

Als Unterelemente kommen grundsätzlich alle Komponenten in Frage, normalerweise besteht eine ToolBar jedoch hauptsächlich aus Elementen der Klasse *TToolButton*, die Sie zur Entwurfszeit über das Kontextmenü der ToolBar einfügen. In diesem Menü ist von einem »neuen Schalter« und einem »neuen Trenner« die Rede, tatsächlich handelt es sich aber bei beiden um die gleichen *TToolButton*-Elemente, die sich nur durch das Property *Style* unterscheiden. Und es gibt sogar noch mehr als diese beiden Stile:

- *tbsButton* ist der schon erwähnte normale Schalter. Seine Oberfläche wird normalerweise durch ein grafisches Symbol geziert, kann aber auch eine einfache Textbeschriftung zugewiesen bekommen (*TToolBar.ShowCaptions* muss dann *True* sein).
- *tbsSeparator* steht für das schon erwähnte Trennelement, dessen Breite Sie frei wählen können.
- *tbsDivider* ist eine Alternative zu *tbsSeparator*, es zeigt zusätzlich eine vertikale Linie an.
- *tbsCheck* ist eine Variation von *tbsButton*. Äußerlich stimmt ein *tbsCheck*-Element mit einem normalen Schalter überein, das Verhalten gleicht jedoch eher einer CheckBox, denn ein einmaliges Anklicken versetzt den Schalter in einen dauerhaften gedrückten Zustand, der bis zum nächsten Klicken anhält.
- *tbsDropDown* ergänzt den *tbsButton*-Schalter durch einen Pfeil, über den sich ein DropDown-Menü aufklappen lässt. Über die normale Schaltfläche lässt sich weiterhin eine Aktion direkt, also ohne Menü, auslösen. Typisches Beispiel für diese Schalter sind Schalter zum Öffnen von Dateien, bei denen über den Pfeil eine Liste der zurückliegenden Dateien aufgerufen werden kann, z.B. in der Kylix-IDE oder im TreeDesigner.

Um einen *TToolButton* mit einem Bild zu versehen, geben Sie im Property *ImageIndex* einen Bildindex an. Dieser bezieht sich auf die Bilderliste, mit der die Toolbar verknüpft ist, und zwar über das Property *TToolBar.Images*. Für die Gestaltung einer ToolBar benötigen Sie also auch noch eine *TImageList*-Komponente, in die Sie die Bilder für alle ToolButtons laden, wie in Kapitel 3.6.2 in Zusammenhang mit ListViews beschrieben.

Die Schalterleisten des TreeDesigners

Da es nahe liegt, die Schalter mit den verschiedenen grafischen Formen als »Zeichenwerkzeuge« zu bezeichnen, soll die Leiste am linken Rand des TreeDesigner-Hauptfensters als »Werkzeugleiste« bezeichnet werden, alle anderen Leisten am oberen und unteren Rand als »Symbolleisten«. Während für alle anderen Symbolleisten die Klasse *TToolbar* verwendet wurde, basiert die Werkzeugleiste auf der Komponente *TPanel*, da es in dieser einfacher ist, Schalter vertikal übereinander anzuordnen. Die Werkzeugleiste soll außerdem nach den Schaltern noch einen kleinen Vorschaubereich für die eingestellten Farben und die Liniendicke enthalten. Auch dieser lässt sich in einem *TPanel* erheblich stressfreier entwerfen als in einer Toolbar, die einem mit diversen Funktionen zur automatischen Anordnung und Größeneinstellung des Öfteren ins Handwerk pfuscht.

Die Symbolleisten enthalten außer weiteren Schaltern auch interaktive Elemente. Diese und alle weiteren Komponenten der Werkzeug- und Symbolleisten, die in den folgenden Kapiteln innerhalb des Beispielcodes erwähnt werden, sind in der folgenden Tabelle zusammengefasst:

Komponente	**Klasse**	**Aufgabe**
SpeedBtnArrow	TSpeedButton	schaltet den Manipulationsmodus an, in dem die Maus zum Verschieben, Markieren und Ändern der Größe verwendet wird.
SpeedBtnRectangle, SpeedBtnEllipse, SpeedBtnLine, speedBtnRhomb, ...	TSpeedButton	wählen jeweils eine Form aus, die gezeichnet werden soll. In Frage kommen die Formen, die im Aufzählungstyp *TShapeType* der Unit *GrDoc* genannt sind (stNone, stRect, stEllipse, stLine, stHexagon, stRhomb, stRoundRect).
SpeedBtnArrangeTree	TSpeedButton	Schalter zum Anordnen des Baums.
ShapeText	TEdit	Hier geben Sie die Beschriftung für das gewählte Element oder die gewählten Elemente ein.
FontList	TComboBox	Hier können Sie eine andere Schrift für die markierten Elemente auswählen.
FontSize	TSpinEdit	In diesem Eingabefeld geben Sie die Größe der Schrift ein.
ZoomFactorBar	TTrackBar	Regler, mit dem Sie fließend auf den Mittelpunkt des dargestellten Bildbereichs zoomen können bzw. mit dem Sie das Bild verkleinern können.
ZoomLabel	TLabel	zeigt den aktuellen Vergrößerungsfaktor an (0.1 bis 10.0).

Einstellungen der Properties im Objektinspektor sind für verschiedene Funktionsbereiche notwendig: zur Anzeige von Symbolen auf den Schaltflächen, zur Bildung von Schaltergruppen (mit diesen beiden Themen befassen sich zwei der folgenden Abschnitte) und zum allgemeinen optischen Erscheinungsbild von Schaltern und Leisten.

Zu Letzterem gehört vor allem, dass der TreeDesigner flache Schalter verwendet. Hierzu wurden alle *Flat*-Properties von *TToolBar* und *TSpeedButton*-Komponenten auf *ue* gesetzt. *TToolButton*-Komponenten brauchen nicht einzeln auf flach geschaltet zu werden, da sie immer zusammen mit der gesamten Toolbar zwischen flach und dreidimensional hin- und hergeschaltet werden.

Property-Einstellungen nachvollziehen *R5*

Die weiteren Properties sollen hier nicht im Einzelnen untersucht werden, da sich die meisten schon durch die Bezeichnung im Objektinspektor selbst erklären. Wenn Sie sich einmal fragen sollten, wie in einem beliebigen Beispielprogramm, etwa auch einem aus dem Lieferumfang von Kylix, welche Properties bei einer Komponente verndert wurden, brauchen Sie nur ANSICHT ALS TEXT aus dem lokalen Menü des Formulars auszuwählen. In der Textdarstellung werden *alle* von der Standardeinstellung abweichenden Werte genannt, die nicht davon abweichenden Werte werden normalerweise nicht genannt. Mit der Suchfunktion finden Sie auch in der Textdarstellung jede Komponente sofort. Für die Toolbar mit den Zoom-Einstellungen im TreeDesigner werden beispielsweise die folgenden Werte angezeigt:

```
object ZoomToolbar: TToolBar
    Left = 0
    Top = 45
    Width = 796
    Height = 28
    ButtonWidth = 25
    Caption = 'Zoomeinstellung'
    EdgeBorders = []
    Flat = True
    Images = MenuImages
    ...
```

Bebilderung

Das Dokumentformular des TreeDesigners enthält eine *TImageList*-Komponente mit ca. 20 Bildern, die sowohl für die Schalter als auch für die Menüs verwendet werden. Das bedeutet, dass die *Images*-Properties aller Toolbars und auch das der *TMainMenu*-Komponente auf diese Bilderliste verweisen. Damit die Bilder in die Menüs passen, ohne die üblichen Maße für einen Menüeintrag durcheinander zu bringen, wurden sie auf eine Größe von 16*16 Pixeln beschränkt. Ein weiterer Grund für diese Vorgehensweise ist, dass die mit Kylix mitgelieferten Beispielbilder in dieser Größe vorliegen. Die im TreeDesigner verwendeten Bitmaps stammen zu einem Teil aus den Bilddateien des Verzeichnisses `kylix/images`, zum Teil aus der Projektschablone für eine MDI-Anwendung. Das Bild für die Baum-Anordnung ist vom Autor erstellt und von einem Bildverarbeitungsprogramm auf 16*16-Pixel heruntergerechnet worden.

Die SpeedButton-Schalter in der Werkzeugleiste verwenden ein etwas größeres Schalter-Format und 20*20-Pixel große Bitmaps, die allerdings nicht in eine Bilderliste, sondern direkt in das *Glyph*-Property der einzelnen Schalter geladen werden.

Hinweis: Da *Glyph*-Properties Objekte des Typs *TBitmap* sind, können Sie mit den einfachen *TBitmap*-Methoden auch zur Laufzeit neue Bilder in die Schalter laden. Wenn Sie für verschiedene Schalterzustände verschiedene Bitmaps verwenden wollen, müssen Sie die in der Online-Hilfe beschriebenen *TSpeedButton*-Properties *Glyph*, *Layout*, *Margin*, *Spacing* und *NumGlyphs* verwenden und mehrere Bilder in einer Bitmap speichern.

Schalter-Gruppen

Die Zeichenwerkzeug-Schalter im TreeDesigner verhalten sich ähnlich wie eine Radioschaltergruppe, es kann also immer nur ein Schalter gewählt sein. Während der TreeDesigner in diesem Fall *TSpeedButtons* verwendet, können Sie Schaltergruppen auch mit *ToolButtons* realisieren. Damit die CLX die Einhaltung der Gruppen-Regel sicherstellt und den bisher gewählten Schalter automatisch abwählt, wenn ein anderer Schalter der Gruppe gedrückt wird, müssen Sie

- bei *TSpeedButtons* allen Schaltern der Gruppe eine übereinstimmende Kennzahl im Property *GroupIndex* geben, im TreeDesigner ist das der Index 1. *GroupIndex* ist normalerweise mit Null voreingestellt, was bedeutet, dass ein Schalter zu keiner Gruppe gehört.
- bei *TToolButtons* das *Grouped*-Property aller Schalter, die eine Gruppe bilden, auf *True* setzen. Wenn mehrere Gruppen in einer Toolbar vorhanden sind, erkennt die Toolbar die Gruppengrenzen an Separatorschaltern oder an anderen Schaltern, bei denen *Grouped* auf *False* gesetzt ist.
- In beiden Fällen bewirkt das Property *AllowAllUp* mit seiner standardmäßigen *False*-Einstellung, dass immer ein Schalter der Gruppe gedrückt sein muss.

TControlBar

Für die uns hierbei als Nächstes begegnende Komponente gibt wieder einmal die Kylix-IDE selbst das beste Anwendungsbeispiel. Menüleiste, Symbolleisten und die Komponentenpalette befinden sich in einem Exemplar der *TControlBar*-Komponente, die Borland einst unter Windows als Alternative zu einem Common Control von Microsoft (der CoolBar) entwickelt hat.

In der Kylix-IDE können Sie bereits sehen, dass eine *ControlBar* dazu dient, mehrere Symbolleisten auf einer Art Zeilenraster anzuordnen und dem Benutzer zu erlauben, die Leisten flexibel auf diesem Raster anzuordnen. Die Teile, die innerhalb dieses Zei-

lenrasters bewegt werden können, werden *Bänder* genannt. Ein Band hat an seiner linken Seite einen Griff, an dem Sie das Band innerhalb einer Zeile verschieben können (wobei die Position der anderen Bänder derselben Zeile automatisch angepasst wird) oder es in eine andere Zeile versetzen. Hat das *AutoSize*-Property der ControlBar den Wert *True*, können Sie auf diese Weise sogar aus einer einzeiligen eine mehrzeilige Komponente machen (zu Entwurfszeit und Laufzeit). Die CLX-Version der *TControlBar* unterscheidet sich von der Windows-VCL-Version optisch dadurch, dass die Griffe durch eine einzelne Linie hervorgehoben werden, während es in der VCL zwei Linien sind (wie in der Kylix-IDE zu sehen; dieser Unterschied beeinträchtigt nicht die Übereinstimmung in der Funktionsweise der beiden *TControlBar*-Versionen).

Die Bänder werden von *TControlBar* automatisch erzeugt, wenn Sie Komponenten in die ControlBar einfügen: Dabei erhält *jede* eingefügte Komponente ein eigenes Band mit eigenem Griff, daher werden normalerweise mehrere Einzelkomponenten in einer größeren Komponente zusammengefasst, so dass sie zusammen als Gruppe ein einziges Band zugeteilt bekommen.

Das Hauptformular des TreeDesigners verwendet eine ControlBar mit fünf Bändern am oberen Fensterrand und eine weitere mit nur einem Band am unteren Rand.

Programmierung

Die Programmierung von Symbolleisten kommt normalerweise mit Standard-Techniken aus und soll daher an dieser Stelle lediglich stichpunktartig abgehandelt werden:

- Zur Bearbeitung von Aktionsschaltern haben Sie die Wahl, die Bearbeitungsmethode einfach mit dem *OnChange*-Ereignis des Schalters zu verknüpfen oder eine Aktion in einer Aktionsliste dafür bereitzustellen, die Sie dann auch mit einem gleichbedeutenden Menüpunkt verknüpfen können. Zu Vorteilen und Verwendung von Aktionslisten siehe Kapitel 4.6.4.
- Die Einstellungen, die in den Kontrollelementen der Leisten vorgenommen werden, können direkt über Properties der Komponenten abgefragt werden, z. B. *ShapeText.Text* für das Editierfeld der Beschriftung und *PenWidth.Position* für die Einstellung des Trackbars für die Stiftbreite.
- Zur Abfrage von Schaltergruppen gibt es spezielle Techniken, die in Kapitel 3.5.2 behandelt sind. So können Sie im TreeDesigner etwa über die SpeedButtons an der linken Fensterseite eine Grafikform einstellen. Jeder SpeedButton steht dabei für einen anderen Wert in der Aufzählung *TShapeType*. Die *Tag*-Properties der SpeedButtons wurden auf den jeweils zu ihnen passenden *TShapeType*-Wert eingestellt, damit sich der Wert schneller ermitteln lässt (zu dieser Verwendung des *Tag*-Properties siehe R13 Seite 352).

- Bei manchen Leisten-Steuerelementen soll sich eine Änderung sofort auf das bearbeitete Dokument auswirken. Im TreeDesigner wird z.B. der editierte Beschriftungstext (Komponente *ShapeText*) bei jeder Änderung im Grafikdokument aktualisiert und auch die Wahl einer anderen Farbe aus der Farbpalette wirkt sich sofort auf die selektierten Grafikobjekte aus. Hierzu müssen lediglich die Änderungs-Ereignisse bearbeitet werden, bei *ShapeText* das Event *TEdit.OnChange* und bei der Farbpalette die Events *OnPenColorChange*, *OnBrushColorChange* und *OnFontColorChange*. Ein Beispiel für die Bearbeitung eines solchen Events finden Sie in Kapitel 6.3.2. Eine zusätzliche Herausforderung im Falle des TreeDesigners besteht darin, diese Events an das richtige Fenster weiterzuleiten, denn es gibt im TreeDesigner nur eine einzige Farbpaletten-Komponente für alle Dokumentfenster. Und die Farbänderungs-Events sollen sich nur auf das aktuelle Fenster auswirken. Wie dies im TreeDesigner gelöst ist, zeigt Kapitel 6.7.6.

6.2.2 Komponenten für die Zeichenfläche

Da die wichtigste Aufgabe der Zeichenfläche, das Ausgeben der Grafik, in den Kapiteln 6.5 und 6.6 besprochen wird, werfen wir hier nur einen kurzen Blick auf den zur Entwurfszeit festgelegten Aufbau der Zeichenfläche.

Bereiche für die Grafikausgabe

Das Dokumentformular ist ein Beispiel für ein Formular, dessen *Canvas*-Zeichenfläche nicht für die Grafikausgabe in Frage kommt, da Teile des Formulars und damit der Zeichenfläche durch automatisch angeordnete Komponenten wie die Lineale belegt werden. Selbst wenn sich ein fester Bereich zur Grafikausgabe finden ließe, würde dieser vielleicht bei einer Weiterentwicklung der Anwendung verändert, wenn z.B. eine neue Leiste in das Fenster integriert werden würde. (Die CLX passt die Zeichenfläche des *Canvas*-Properties des Formulars nicht wie eine mit *alClient* ausgerichtete Komponente automatisch in den freien Bereich des Formulars ein.)

Aus diesem Grund werden Sie wahrscheinlich immer eine weitere Komponente in das Formular einfügen, die den Bereich für die Grafikausgabe klar definiert und die auch über ein eigenes *Canvas*-Property verfügt. Unter den Standardkomponenten bietet sich hierzu die Klasse *TPaintBox* an.

Im Formular des TreeDesigners sind zunächst einmal zwei einfache Lineal-Komponenten der Klasse *TRuler* eingebunden. *TRuler* wird auf der CD-ROM mitgeliefert und soll hier lediglich als »Black Box« betrachtet werden. Die Lineare werden später zur maßstabsgerechten Positionsanzeige beim Scrolling verwendet (Kapitel 6.5.4). An dieser Stelle genügt es zu wissen, dass die beiden Komponenten mit *alTop* bzw. mit *alLeft* an den oberen und den linken Rand des Formulars geheftet wurden.

Die *TPaintBox*-Zeichenfläche füllt den gesamten noch freien Bereich aus. Für den TreeDesigner genügt eine einfache Zeichenfläche jedoch noch nicht, denn auf einem normalen Bildschirm wäre eine solche für die vollständige CLX-Hierarchie viel zu klein. Es müssen also noch eine horizontale und eine vertikale Bildlaufleiste hinzukommen, die sich in einem praktischen Zweierpack bereits in der Komponente *TScrollBox* befinden.

Es ist diese *TScrollbox*-Komponente, die den gesamten noch freien Platz des Formulars einnimmt, also mit *alClient* ausgerichtet ist. Die *TPaintBox*-Komponente ist dieser crollbox als Kindfenster untergeordnet. Ihre Größe ist im TreeDesigner standardmäig auf 3000 mal 3000 Pixel eingestellt, so dass bei einem Vergrößerungsfaktor von 1 das gesamte Grafikdokument hineinpasst. Die ScrollBox stellt dazu automatisch passende Scrollbars bereit. Weitere Erläuterungen zum Scrolling und zur Vergrößerung der Grafikdarstellung überlassen wir den Kapiteln 6.5 und 6.6.

Hinweis: Der TreeDesigner verwendet für seine Scrollbox nicht die Klasse *TScrollBox*, sondern die in Kapitel 8.5.2 erläuterte *TScrollboxEx*.

6.2.3 Wichtige Menübefehle

Normalerweise unverzichtbare Bestandteile eines Dokumentformulars sind auch Menüpunkte, die das Speichern und Laden des Dokuments gestatten. Da *TGraphicDoc*, die Dokument-Klasse des TreeDesigners, ähnlich einfache Methoden zum Speichern und Laden zur Verfügung stellt wie verschiedene Klassen der CLX mit *SaveToFile* und *LoadFromFile*, bleiben die Ereignismethoden für die *OnClick*-Ereignisse der Menüpunkte sehr überschaubar.

Hier geht es um den Code der Menüpunkte SPEICHERN, SPEICHERN UNTER ..., ÖFFNEN... und SCHLIESSEN. Grundlage für diese Methoden ist, dass sich das Programm merkt, welche Datei gerade bearbeitet wird. Nur so kann es die Datei sofort unter dem richtigen Namen speichern, ohne den Benutzer vorher mit dem *Speichern-unter*-Dialog danach zu fragen.

Speichern

Der TreeDesigner speichert den Dateinamen eines Dokuments im zugehörigen Dokumentobjekt. Das Dokument*fenster* kann über *TGraphicDoc.FileName* auf diesen zugreifen. Falls die Datei noch unbenannt ist, enthält dieses Property einen Leerstring.

Die *Speichern-unter*-Dialogbox wird aufgerufen, wenn der Benutzer den entsprechenden Menüpunkt aufruft oder eine bisher unbenannte Datei gespeichert wird. In beiden ällen erledigt die folgende Methode den Aufruf:

```
procedure TDocumentForm.CmSaveAsClick(Sender: TObject);
{ OnClick für "Speichern unter..." }
begin
  SaveDialog1.FileName := Document.FileName;
  if SaveDialog1.Execute then begin
    Document.SaveToFile(Filename);
    // den neuen Dateinamen an die Liste der zurückliegenden
    // Dateien anhängen:
    FileList.AddString(SaveDialog1.FileName);
  end;
end;
```

Wenn der Benutzer den Menüpunkt SPEICHERN wählt, kann die Datei sofort gespeichert werden, falls sie schon einen Namen hat. Ansonsten wird einfach die eben gezeigte Methode für den *Speichern-unter*-Dialog aufgerufen:

```
procedure TDocumentForm.CmSaveClick(Sender: TObject);
{ OnClick für "Speichern" }
begin
  { Testen, ob Datei schon benannt ist: }
  if Document.FileName <> '' then
    { Wenn ja, unter diesem Namen speichern }
    Document.SaveToFile(Document.FileName);
  { wenn nicht, Methode für "Speichern unter" aufrufen: }
  else CmSaveAsClick(Sender);
end;
```

Aktualisierung von Dateiname, Fenstertitel und Änderungsflag

Beim Laden oder Speichern eines Dokuments müssen normalerweise drei Dinge berücksichtigt werden:

- Eine Flag oder ein Zähler, der speichert, ob an dem Dokument Änderungen vorgenommen worden sind, die noch nicht gespeichert wurden. Dieses Flag bzw. der Zähler muss sowohl beim Laden einer neuen Datei als auch beim Speichern zurückgesetzt werden, so dass beim Schließen des Fensters keine unnötige Sicherheitsabfrage durchgeführt wird.
- Der Dateiname muss intern gespeichert werden, damit der Benutzer ihn beim nächsten Speichern nicht noch einmal angeben muss.
- Der Fenstertitel muss aktualisiert werden, sofern er den Dateinamen angibt.

In den obigen Methoden gab es von alledem nichts zu sehen, weil Dateiname und Änderungszähler im Dokumentobjekt verwaltet werden, die genannten Dinge werden oben also innerhalb von *Document.SaveToFile* berücksichtigt. Das Dokumentobjekt veranlasst die Aktualisierung des Fenstertitels bei einem geänderten Dateinamen und enthält auch einen Änderungszähler, den es bei jeder Änderung am Dokument erhöht.

Damit das Dokument auch alle Änderungen kontrollieren kann, deklariert es alle Variablen als *private* und lässt Zugriffe darauf nur über Methoden zu, in denen dann unter anderem auch der Änderungszähler richtig gesetzt wird. Beispiele für solche Methoden gibt Kapitel 6.3.2.

Hinweis: Die Dokumentklasse *TGraphicDoc* ist von der Klasse *TList* abgeleitet. Damit auch in den *TList*-Methoden *Add* und *Delete* der Änderungszähler berücksichtigt wird, muss *TGraphicDoc* die geerbten Methoden *Add* und *Delete* überschreiben und das geerbte Verhalten (ohne Änderungszähler) entsprechend ergänzen. Eigentlich müsste *TGraphicDoc* sogar *alle* den Listeninhalt ändernden Methoden von *TList* überschreiben und durch die Änderungszählung erweitern; im TreeDesigner werden jedoch nur *Add* und *Delete* von außen aufgerufen, und so wurde auf diesen Schritt verzichtet.

Der Dateiname des Dokuments (*TGraphicDoc.FileName*) wird innerhalb der Methoden *SaveToFile* und *LoadFromFile* auf den als Parameter angegebenen Dateinamen gesetzt. Bleibt noch die Frage, wie der Dateiname in den Fenstertitel gelangt. *TGraphicDoc* speichert für das Dokumentfenster einen Zeiger namens *UpdateCaptionProc*, der auf eine Formularmethode weist, die den Titel aktualisiert. Da diese Methode im TreeDesigner sehr stark mit dem Dokument-View-Konzept zusammenhängt, wird sie erst in Kapitel 6.3.1 besprochen.

Lesen von Dateien

Das Lesen von Dateien funktioniert ähnlich simpel wie das Schreiben, als Dialog kommt ein Exemplar von *TOpenDialog* zum Einsatz:

```
procedure TDocumentForm.LoadFromFile(NewFileName: String);
begin
  Document.FreeAll; // alle Objekte löschen
  Document.LoadFromFile(NewFileName); // neue Objekte laden
  Invalidate; // bewirkt die Neuausgabe der Grafik
end;

procedure TDocumentForm.CmLoadClick(Sender: TObject);
{ OnClick für "Öffnen..." }
begin
  if OpenDialog1.Execute then
    LoadFromFile(OpenDialog1.FileName);
end;
```

Schließen

Obwohl man ein Fenster je nach verwendetem Fenstermanager über unterschiedliche astenkombinationen und Mausaktionen schließen kann, verzichtet kaum ein Doku-

mentfenster auf einen eigenen Menüpunkt SCHLIESSEN im Dateimenü. Die zugehörige Methode ist mit sechs Zeichen Inhalt die kürzeste in diesem Buch:

```
procedure TDocumentForm.CmCloseClick(Sender: TObject);
begin
  Close;
end;
```

Der Aufruf von *Close* bewirkt indirekt, dass es zu einer Sicherheitsabfrage kommt, falls die Datei nicht gespeichert wurde. Das damit zusammenhängende *OnCloseQuery*-Ereignis ist ein Thema des nächsten Kapitels.

6.2.4 Typische Formularaufgaben

Um das allgemeine Management des Dokumentformulars abzuschließen, behandelt dieses Kapitel

- die Sicherheitsabfrage vor dem Schließen eines ungespeicherten Dokuments (eine der nahezu unverzichtbaren Funktionen),
- die Anpassung von Menü-Properties zur Laufzeit und
- Popup-Menüs für lokal anwendbare Funktionen.

Die Sicherheitsabfrage *R62*

Durch die marktüblichen Anwendungen ist der Anwender bereits so stark an die Sicherheitsabfrage vor dem Schließen einer ungespeicherten Datei gewöhnt, dass es sich eine Anwendung kaum leisten kann, darauf zu verzichten. In Kylix lässt sich eine solche Sicherheitsabfrage sehr schnell mit Hilfe des *OnCloseQuery*-Events realisieren.

Als Erstes benötigen Sie jedoch eine Variable, in der Sie speichern, ob das aktuelle Dokument bereits verändert wurde. Beim Speichern setzen Sie diese Variable auf *False* zurück, bei jeder Änderung wieder auf *True*. Der TreeDesigner verwendet statt einer booleschen Variable einen Zähler namens *Changed*, der größer als Null ist, falls die Datei seit der letzten Speicherung verändert wurde.

Das *OnCloseQuery*-Ereignis tritt immer auf, wenn der Benutzer versucht, ein Fenster zu schließen, oder wenn dieses Fenster indirekt geschlossen wird, weil die Anwendung beendet wird. Die Methode für das *OnCloseQuery*-Event erhält von der CLX einen booleschen Variablenparameter *CanClose*, der per Voreinstellung mit *True* vorbelegt ist, was bedeutet, dass das Formular geschlossen werden kann.

Falls die Funktion *IsChanged* des Dokuments auf eine ungespeicherte Änderung hinweist, startet die Methode *FormCloseQuery* des TreeDesigners mit der *MessageDlg*-Funktion der *Dialogs*-Unit eine Anfrage, ob die Änderungen gespeichert werden sollen. Der Benutzer hat die Wahl zwischen den Schaltern *mbYes*, *mbNo* und *mbCancel*.

Wählt er *Ja*, so simuliert die Methode den Aufruf des Menüpunkts DATEI | SPEICHERN, indem sie die Methode *CmSaveClick* aufruft:

```
procedure TDocumentForm.FormCloseQuery(Sender: TObject;
            var CanClose: Boolean);
var
  Answer: Integer;
  DocName: string;
begin
  if Document.IsChanged then begin
    { "DocName" gibt im MessageDlg-Text den Dateinamen an: }
    DocName := Document.FileName;
    if DocName = '' then DocName := 'unbenannter Datei';
    Answer := MessageDlg(
      Format('Soll %s gespeichert werden?', [DocName]),
      mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    case Answer of
      mrYes: begin
        try
          CmSaveClick(Self);
        except
          on E: Exception do begin
            Application.HandleException(E);
            CanClose := False;
          end;
        end;
        { testen, ob im Sichern-Dialog Abbruch gedrückt wurde -
          kann an der Variable "Change" erkannt werden: }
        CanClose := (Changed = 0);
      end;
      mrCancel: CanClose := False;
      else CanClose := True;
    end;
  end;
end;
```

FormCloseQuery erlaubt nur dann das Schließen des Fensters, wenn die Datei gar nicht verändert wurde, wenn der Benutzer den Schalter *mbNo* gewählt hat oder wenn er den Schalter *mbYes* gewählt hat *und* die Variable *Changed* durch das Speichern auf Null zurückgesetzt wurde. Ist *Changed* trotz Drückens des Ja-Schalters ungleich Null, so bedeutet das, dass *CmSaveClick* zwar den *Speichern unter*-Dialog aufgerufen hat, dass der Benutzer in diesem jedoch *Abbruch* gedrückt hat.

Schließlich kann es auch noch vorkommen, dass die Speichern-Funktion aufgerufen wurde, es dabei aber zu einer Exception gekommen ist. Die obige Methode fängt alle Exceptions ab und zeigt dem Benutzer über *Application.HandleException* eine Meldung darüber an. Außerdem verhindert sie das Schließen des Fensters, indem Sie *CanClose* auf *False* setzt (dabei wurde angenommen, dass eine Exception meist bedeutet, dass die Datei nicht korrekt gespeichert werden konnte; dies könnte bei Interesse natürlich

über den Typ der Exception genauer abgefragt werden: z.B. kann es zu einer *EFCreateError*-Exception kommen, wenn die Datei in einem Verzeichnis erzeugt wird, für das keine Schreibberechtigung vorliegt).

Ohne diesen Exception-Handler würde die Methode, falls es beim Speichern zu einer Exception kommt, einfach abgebrochen, das Fenster würde geschlossen werden und der Benutzer hätte eventuell seine Änderungen am Dokument verloren.

Menümarkierung und -deaktivierung

Um die Markierung eines Menüpunkts, wie z.B. des Menüpunkts ANSICHT | GERÄTEKOORDINATENSYSTEM, ein- und auszuschalten, müssen Sie zur Laufzeit das Property *Checked* ändern. Den Namen des Menüpunkts, hier *NoMapMode*, erfahren Sie am schnellsten aus dem Objektinspektor, wenn Sie den Menüpunkt im Menüeditor ausgewählt haben. Die folgende Zeile kehrt also den bisherigen Markiert-Status des genannten Menüpunkts um:

```
NoMapMode.Checked := not NoMapMode.Checked;
```

Entsprechend der Deaktivierung von Steuerelementen (siehe Kapitel 3.3.1) sollten auch Menüpunkte manchmal deaktiviert werden, falls sie gerade nicht sinnvoll anzuwenden sind. Auch bei Menüpunkten heißt das entsprechende Property *Enabled*.

Aktionslisten im TreeDesigner *R23*

Im TreeDesigner gibt es nur ein Menü, in dem Menüpunkte deaktiviert werden, das Menü *Ansicht*. Wenn Sie die Option ANSICHT | GERÄTEKOORDINATENSYSTEM einschalten, sollen unter anderem die Menüpunkte zum Vergrößern und Verkleinern sowie der entsprechende Scrollbar in der Werkzeugleiste deaktiviert werden. Deaktiviert werden insbesondere auch die Menüpunkte SEITENANZEIGE und 100%, die in einer Toolbar durch Schalter repräsentiert sind, welche ebenfalls deaktiviert werden müssen.

Zur gleichzeitigen Behandlung von Menüpunkten und zugehörigen Toolbar-Schaltern verwendet der TreeDesigner für das Ansichts-Menü eine Aktionsliste (zum Aufbau von Aktionslisten siehe Kapitel 4.6.4). Statt die *Checked*- und *Enabled*-Properties der Menüpunkte und Schalter direkt zu setzen, setzt er einfach die gleichnamigen Properties der entsprechenden Aktion (mit welcher Menüpunkte und Schalter über das *Action*-Property verknüpft sind). Statt wie in der oben gezeigten Zeile auf *NoMapMode* greift er beispielsweise auf *ANoMapMode*, das zugehörige *TAction*-Objekt der Aktionsliste, zu:

```
procedure TDocumentForm.NoMapModeClick(Sender: TObject);
begin
{ Aktualisierung von Menü UND Toolbars durch Aktionen: }
  { Markierungs-Flag umkehren: }
  ANoMapMode.Checked := not ANoMapMode.Checked;
```

```
    { davon abhängige Aktionen aktivieren oder deaktivieren: }
    ASeitenAnzeige.Enabled := not ANoMapMode.Checked;
    AMaximaleVerkleinerung.Enabled := not ANoMapMode.Checked;
    AN100Prozent.Enabled := not ANoMapMode.Checked;
    AN200Prozent.Enabled := not ANoMapMode.Checked;
    { davon abhängiges Steuerelement aktivieren oder deaktivieren: }
    ZoomFactorBar.Enabled := not NoMapMode.Checked;
    { Durchführen des eigentlichen Menübefehls
      (Gerätekoordinatensystem einschalten): }
    if NoMapMode.Checked then begin
      FZoomFactor := 1;
      ZoomFactorBar.Position := 10;
    end;
    ResizePaintbox;
  end;
```

Diese Vorgehensweise birgt gute Erweiterungsmöglichkeiten in sich: Auch wenn bisher nur zwei der Menüpunkte durch Toolbar-Schalter dupliziert werden, können in Zukunft auch alle anderen Punkte in der Toolbar repräsentiert werden. Wenn die neuen Schalter dann mit den bestehenden Aktionen verknüpft werden, muss kein neuer Code geschrieben werden, um die Schalter zu (de-)aktivieren oder gedrückt darzustellen (ein Schalter für einen markierten Menüpunkt wird gedrückt, also mit *Down*-Property = *True* dargestellt).

Optionen in Popup-Menüs

Popup-Menüs geben dem Anwender einen viel schnelleren Zugriff auf bestimmte Optionen als mehrseitige Dialogboxen oder geschachtelte Hauptmenüs, die erst nach der richtigen Option durchsucht werden müssen. Die Kylix-IDE gibt mit ihren lokalen Menüpunkten HINWEISE ANZEIGEN (für die Komponentenpalette) und IMMER IM VORDERGRUND (für den Objektinspektor) selbst ein Beispiel für Einstellungen, die sehr ützlich sein können, aber nicht über das Hauptmenü vorgenommen werden können.

Da Sie ein Popup-Menü unter Kylix so schnell anlegen können wie ein Hauptmenü und da Popup-Menüs das Programm auch aus der Sicht des Programmierers übersichtlicher werden lassen, steht dem intensiven Gebrauch dieser Menüs eigentlich nichts im Wege.

ügen Sie für die Komponenten Ihres Formulars, die ein Popup-Menü besitzen sollen, jeweils eine *PopupMenu*-Komponente von der ersten Seite der Komponentenpalette in das Formular ein, tragen Sie dieses Popup-Menü im *PopupMenu*-Property der Komponente ein, editieren Sie das Menü im Menüdesigner und verknüpfen Sie seine Punkte durch einen Doppelklick mit Programmcode.

Als kurzes Beispiel soll hier der Menüpunkt SCHRIFT WECHSELN... der Lineale im Dokumentfenster des TreeDesigners genügen. Nach seiner Betätigung wird ein Schriftaus-

wahl-Standarddialog aufgerufen, durch den der Benutzer die Schrift beider Lineale (*VRuler* und *HRuler*) gleichzeitig ändert:

```
procedure TDocumentForm.CmChangeRulerFontClick(Sender: TObject);
begin
  FontDialog1.Font := VRuler.Font;
  if FontDialog1.Execute then begin
    VRuler.Font := FontDialog1.Font;
    HRuler.Font := FontDialog1.Font;
  end;
end;
```

Eine weitere Voraussetzung für den automatischen Aufruf eines Popup-Menüs ist, dass Sie sein *AutoPopup*-Property beim voreingestellten Wert *True* belassen.

Kapitel 6.8.3 beschreibt eine etwas kompliziertere Anwendung von Popup-Menüs: Die Zeichenfläche besitzt nicht nur ein, sondern zwei Popup-Menüs und muss daher auf den automatischen Aufruf durch die CLX verzichten.

6.3 Das Grafikdokument

Die Beispielprogramme der Kapitel 1 bis 5 hatten noch nicht viele Daten zu verwalten. In vielen Fällen beschränkten sich die »Daten« auf den Inhalt einiger Editierfelder und anderer Komponenten. In komplizierteren Fällen waren die Daten mit den Einträgen eines ListViews oder TreeViews verknüpft. In Anwendungen, die eigene Dokumente bearbeiten, ist es vorteilhaft, eine eigene Klasse für ein Dokument zu definieren. Welche Vorteile es hat, diese Klasse vom Formular, das die Daten anzeigen soll, zu trennen, zeigt der erste Teil dieses Abschnitts über das Dokument-View-Konzept.

6.3.1 Das Dokument-View-Konzept

Im Dokument-View-Konzept sorgt eine eigenständige Dokumentklasse dafür, dass alle zum Dokument gehörenden Daten von den Daten des Formulars (dem *View*) deutlich getrennt sind. So gehört beispielsweise die Information über den gerade eingestellten Vergrößerungsfaktor eher zum Formular als zum Dokument und könnte statt in der Dokumentdatei in einer Optionsdatei gespeichert werden. Ein Beispiel dafür gibt die Kylix-IDE selbst: Sie speichert die Projektoptionen weder im Pascal-Quelltext noch in der Projektdatei, sondern in eigenen Optionsdateien mit der Endung `.kof`.

Eine Trennung von Dokument und View wirkt sich nicht nur positiv auf die Verständlichkeit und Wartbarkeit eines Programms aus, sondern schafft auch die Grundlagen für weitere gern gesehene Funktionen von MDI-Anwendungen: So freuen sich Benutzer oft darüber, wenn sie dasselbe Dokument in verschiedenen Fenstern gleichzeitig ansehen können und sie in jedem Fenster eine andere Ansicht erhalten können (z.B. einen ande-

n Vergrößerungsfaktor, eine andere Perspektive, einen anderen Ausschnitt). Wenn jedoch die Daten des Dokuments fest in das Formular integriert sind, ist dem Benutzer diese Möglichkeit ohne programmiertechnische Verrenkungen kaum zu eröffnen.

Das Dokument-View-Konzept im TreeDesigner

In der MDI-Version des TreeDesigners können Sie ein Dokument in mehreren Views, also MDI-Kindfenstern, ansehen. Änderungen am Inhalt des Dokuments wie beispielsweise Farbänderungen von Grafikobjekten oder das Einfügen von neuen Objekten werden sofort in allen geöffneten Views sichtbar. Die Position der Bildlaufleisten und der Vergrößerungsfaktor wird für jedes Fenster getrennt verwaltet, so dass z. B. eine Vergrößerung der Dokumentdarstellung im ersten View nicht automatisch auch eine Vergrößerung im zweiten View verursacht. Dies erlaubt Ihnen, gleichzeitig verschiedene Ausschnitte der Grafik in verschieden Vergrößerungsstufen darzustellen.

Eine erfreuliche Nebenwirkung des Dokument-View-Konzepts ist, dass sich damit ganz leicht ein Übersichtsfenster realisieren lässt, in dem der TreeDesigner eine Gesamtansicht des jeweils aktuellen Dokuments zeigt, zusammen mit einer Andeutung des gerade sichtbaren Ausschnitts (siehe Abbildung 6.1). Dafür muss im Prinzip nur ein fast leeres Übersichtsfenster definiert werden, das dann bei jedem Dokumentwechsel als View zum aktuellen Dokument hinzugefügt wird.

Hinweis: Das MDI-Konzept ist unabhängig vom Dokument-View-Konzept und wird erst in Kapitel 6.7 behandelt. Im Folgenden werden daher die MDI-Kindfenster nur als »Dokumentfenster«, das MDI-Hauptfenster einfach als »Hauptfenster« bezeichnet.

Realisierung des Dokument-View-Konzepts *R45*

Die Grundlage der Trennung von Dokument und View besteht natürlich darin, dass diese beiden Objektarten in getrennten Klassen deklariert werden. Während die Formularklasse für das View zuständig ist, definiert der TreeDesigner die Klasse für das Dokument in einer eigenen Unit namens *GrDoc*. Die Klasse heißt *TGraphicDoc* und wird in Kapitel 6.3.2 detaillierter beschrieben. An dieser Stelle soll es nur um die Dokument-View-Eigenschaften dieser Klasse gehen.

Abbildung 6.3 zeigt die Zusammenhänge zwischen den TreeDesigner-Klassen, die an der Realisierung des Dokument-View-Konzepts beteiligt sind. Wäre der TreeDesigner nach dem Prinzip »ein Fenster – ein Dokument« aufgebaut, genügte es, mit jedem Kindfenster ein eigenes Dokument zu speichern (also jeweils ein Dokument zum Besitz« des Formulars zu machen). Das Dokument-View-Konzept macht es erforderlich, eine von allen Views getrennte Liste von Dokumenten zu verwalten – naheliegenderweise im Hauptfenster.

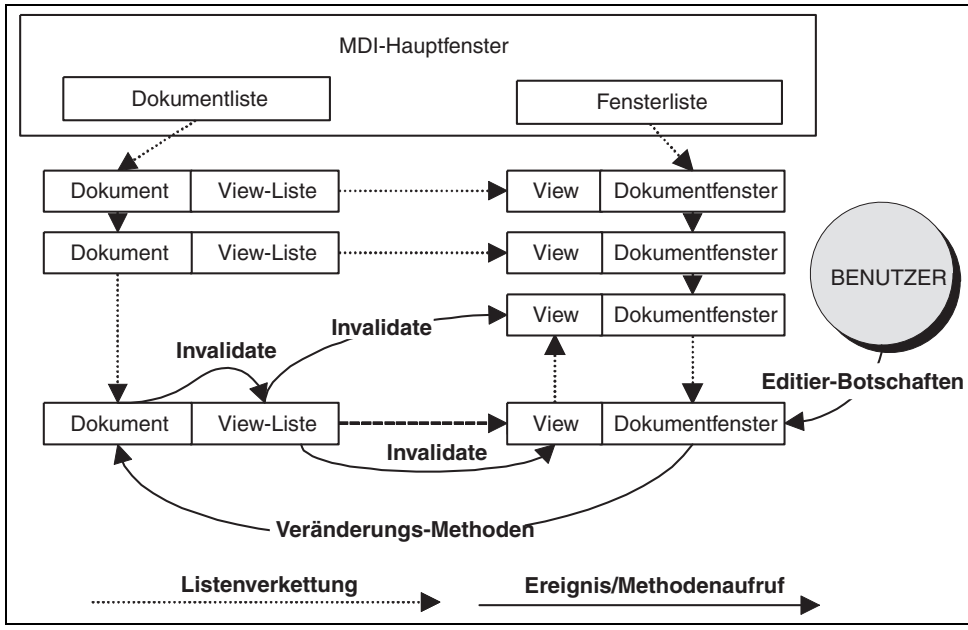


Abbildung 6.3: Eine Editieraktion des Benutzers löst eine Kette von Ereignissen und Methoden aus, an deren Ende erst die sichtbare Veränderung am Bildschirm steht.

Jedes dieser Dokumente verfügt über eine Liste von Views, bei denen es sich prinzipiell um die einzelnen Dokumentfenster handeln könnte. Im TreeDesigner wurde jedoch Wert darauf gelegt, dass die Dokumentklasse möglichst unabhängig von den Formularklassen ist – ein direkter Verweis vom Dokument auf ein Dokumentfenster sollte also vermieden werden. Daher wurde eine dritte Klasse, *TGraphicView* eingeführt, die vom Dokumentfenster initialisiert werden muss und alle View-Daten enthält, die von der Dokumentklasse benötigt werden. *TGraphicView* ist wie *TGraphicDoc* in der Unit *GrDoc* deklariert und unabhängig von der Formularklasse *TDocumentForm*.

Um das Zusammenwirken dieser verschiedenen Objekte und Klassen zu verdeutlichen, zeigt Abbildung 6.3 auch, was beim Editieren des Dokuments durch den Benutzer abläuft. Angenommen, er möchte einem Teil der Grafik eine neue Farbe zuweisen und bedient sich dazu der Technik des Drag&Drop. Für die Anwendung bedeutet dies letztendlich, dass ein Drop-Ereignis im Formular eintrifft (in der Abbildung als *Editier-Botschaft* dargestellt). Das Formular interpretiert diese als Farbänderungswunsch und ruft eine entsprechende Methode des Dokuments auf. Das Dokument ändert daraufhin in seinen Datenstrukturen die Farbe des betroffenen Grafikelements (bzw. der betroffenen Elemente). Jetzt muss noch dafür gesorgt werden, dass diese Änderung auch am Bildschirm sichtbar wird. Dafür muss das Dokument die Liste der mit ihm

verbundenen Views durchgehen und jedes einzelne View zum Neuzeichnen der vernderten Bereiche veranlassen. Die hierzu verwendete Technik des Ungültig-Erklärens *Invalidate*) wird in Kapitel 6.5 näher beschrieben.

Quelltext-Auszüge aus der Haupt-Formularklasse

Wie angekündigt wird die Liste der Dokumente im TreeDesigner in der Klasse des Hauptfensters deklariert. Die folgenden Zeilen zeigen die Deklaration der Liste und die für ihre Verwaltung wichtigen Methoden:

```
TMainForm = class(TForm)
// Ausschnitt: Doc-View-relevante Klassenelemente
  private
    FDocumentList: TList;
  public
    function FindDoc(FileName: String): TGraphicDoc;
    procedure DeleteDoc(Document: TGraphicDoc);
    function GetDoc(FileName: String): TGraphicDoc;
    function NewWindow(Document: TGraphicDoc;
                       ReuseEmptyWindow: Boolean = False): TDocumentForm;
```

Die Elemente der Liste sind die Dokumentobjekte, haben also den Typ *TGraphicDoc* und können mit *FDocumentList.Add(Document)* in die Liste eingefügt werden.

Damit nicht dieselbe Datei in mehreren verschiedenen Dokumentobjekten geöffnet wird, ist es Aufgabe der Methode *FindDoc* festzustellen, ob bereits ein Dokument eines gegebenen Dateinamens existiert:

```
function TMainForm.FindDoc(FileName: String): TGraphicDoc;
var
  i: Integer;
begin
  Result := nil;
  for i := 0 to FDocumentList.Count-1 do
    if TGraphicDoc(FDocumentList[i]).FileName = FileName then begin
      Result := TGraphicDoc(FDocumentList[i]);
      exit;
    end;
end;
```

Aufgabe der Methode *GetDoc* ist es dann, ein Dokumentobjekt zu einem gegebenen Dateinamen zurückzuliefern. Wenn bereits ein solches in der Dokumentliste existiert, wird dieses wiederverwendet, ansonsten wird ein neues erzeugt und aus der Datei geladen (zur Methode *LoadFromFile* siehe Kapitel 6.3.3):

```
function TMainForm.GetDoc(FileName: String): TGraphicDoc;
var
  Document: TGraphicDoc;
begin
  if FileName = '' then Document := nil // leeres Dokument gefordert
```

```
    else Document := FindDoc(FileName);
  if Document = nil then begin        // Dokument noch nicht geladen
    Document := TGraphicDoc.Create;
    if FileName <> '' then begin
      Document.LoadFromFile(FileName);
      // Einfügen in die Liste der zurückliegenden Dokumente
      // (siehe Kapitel 4.6.2, R49):
      FileList.AddString(OpenDialog1.FileName);
    end;
    FDocumentList.Add(Document);
  end;
  Result := Document;
end;
```

GetDoc wird beispielsweise beim Öffnen einer neuen Datei aufgerufen:

```
procedure TMainForm.CMFileOpen(var Msg: TMessage);
var
  Document: TGraphicDoc;
  NewChild: TDocumentForm;
begin
  if OpenDialog1.Execute then begin
    Document := GetDoc(OpenDialog1.FileName);
    NewChild := NewWindow(Document, True);
  end;
end;
```

Die Methode *NewWindow* gibt das gefundene *Document* an den Konstruktor des Dokumentfensters weiter und dieser speichert einen Verweis darauf in den Dokumentfenster-Daten. Eine detaillierte Darstellung dieser Vorgänge würde sich an dieser Stelle nicht lohnen; falls Sie sich dafür interessieren, ist ein schrittweises Verfolgen mit dem Debugger empfehlenswert.

Freigabe der Dokumente

Die Dokumente, die in *GetDoc* zur Liste hinzugefügt werden, müssen natürlich auch irgendwann aus der Liste gelöscht werden. Das Dokumentfenster selbst darf sein Dokument nicht löschen, da es nicht wirklich »sein« Dokument ist, sondern dieses noch in anderen Views dargestellt werden könnte. Vielmehr ist es Aufgabe des Hauptformulars, jedes Mal, wenn ein Kindfenster geschlossen wird, zu überprüfen, ob noch weitere Views an das Dokument angeschlossen sind. Nur wenn das nicht der Fall ist, darf das Hauptfenster das Dokumentobjekt freigeben.

Bei der Beantwortung der Frage, wie das Hauptfenster vom Schließen eines Kindfensters erfährt, soll im TreeDesigner wieder das Prinzip beachtet werden, dass das Kindfenster möglichst unabhängig vom Hauptfenster sein soll. Also ruft es nicht direkt eine Methode des Hauptfensters auf, sondern stellt ein Event zur Verfügung, über welches das Hauptfenster sich bei Bedarf vom Schließen des Fensters informieren lassen kann.

Das Event heißt *OnDestroyDocWin* und das Hauptfenster verknüpft es nach jedem Erzeugen eines Kindfensters in *TMainForm.NewWindow* mit seiner folgenden Methode:

```
procedure TMainForm.ChildWindowDestroy(Sender: TObject);
begin
  // leicht gekürzte Version
  // (zur Verwaltung des Übersichtsfensters siehe CD-ROM)
  with Sender as TDocumentForm do begin
    if Assigned(Document) and (Document.ViewCount = 0) then
      DeleteDoc(Document);
  end;
end;
```

Die Erzeugung dieses Events funktioniert nach dem üblichen Muster (Kapitel 8.3.3) und soll hier nicht weiter erläutert werden.

Die Daten eines Views und deren Initialisierung

Wie schon erwähnt werden alle Daten des Dokumentfensters, die im Grafikdokument benötigt werden, zu der eigenen Klasse *TGraphicView* zusammengefasst:

```
TGraphicView = class
  Form: TForm; // Das Formular, in dem das Dokument dargestellt wird
  PaintBox: TPaintBox; // Zur Darstellung benutzte PaintBox
  ViewNumberCaption: String; // Nummer des Views für die Titelzeile
  // Die folgenden drei Variablen sind Zeiger auf
  // Methoden des Formulars, die vom Dokument oder den darin
  // enthaltenen Grafikelementen aufgerufen werden:
  SetMapModeProc: TSetMapModeProc; // (Kapitel 6.6)
  GetGripSizeFunc: TGetGripSizeFunc; // für die "Rahmengriffe"
  UpdateCaptionProc: TSetFileNameProc;
end;
```

Es ist die Aufgabe des Dokumentformulars, eine solche Struktur zu erzeugen, seine »persönlichen« Daten darin zu speichern und die Struktur schließlich an das Dokumentobjekt weiterzugeben:

```
constructor TDocumentForm.Create(AOwner : TComponent; ...;
                                 ADocument: TGraphicDoc);
begin
  FView := TGraphicView.Create;
  FView.Form := self;
  FView.ViewNumberCaption := '';
  FView.SetMapModeProc := SetMapMode;
  FView.GetGripSizeFunc := GetGripSize;
  FView.UpdateCaptionProc := UpdateCaption;
  Document := ADocument; // Bewirkt den Aufruf von SetDocument
  ...
// Die PaintBox kann erst später beim OnCreate-Ereignis gesetzt werden:
procedure TDocumentForm.FormCreate(Sender: TObject);
begin
```

```
  ...
  FView.PaintBox:=PaintBox;

// Das Property Document wird durch folgende Methode beschrieben:
procedure TDocumentForm.SetDocument(NewDoc: TGraphicDoc);
begin
  ...
  FDocument := NewDoc;
  Document.AddView(FView); // Weitergabe der Struktur an das Dokument
```

Nach diesem *AddView*-Aufruf sind die Verknüpfungen zwischen Dokument und View vollständig aufgebaut und können verwendet werden, wie im nächsten Abschnitt am Beispiel von *TGraphicElement.InvalidateAllViews* gezeigt wird.

Hinweis: Die durch *AddView* hergestellte Verknüpfung wird am Ende innerhalb der Methode *TDocumentForm.FormDestroy* (Ereignis *TForm.OnDestroy*) wieder gelöst. Hierzu wird analog zu *Document.AddView* die Methode *Document.DeleteView* verwendet.

View-Verwaltung im Dokument

Die Dokumentklasse enthält ein *TList*-Objekt zur Speicherung aller mit ihm verbundenen Views. Es wird wie üblich im Konstruktor und Destruktor initialisiert (hier nicht gezeigt) und erhält einige Hilfsmethoden, die ebenfalls so einfach sind, dass ihre Implementierung nicht abgedruckt zu werden braucht:

```
TGraphicDoc = class(TList)
  // Ausschnitt: Doc-View-relevante Klassenelemente
private
  FViews: TList;  { darstellende Views }
  function GetViews(Index: Integer): TGraphicView;
  function GetViewCount: Integer;
public
  procedure AddView(AView: TGraphicView);
  procedure DeleteView(AView: TGraphicView);
  property Views[Index: Integer]: TGraphicView read GetViews;
  property ViewCount: Integer read GetViewCount;
```

Eine beispielhafte Anwendung der View-Liste ist die Methode, die das Neuzeichnen aller mit dem Dokument verbundenen Views bewirkt. Es ist allerdings keine Methode des Grafikdokuments, sondern die eines einzelnen Grafik*elements*, das aber über sein Property *Document* auf das Grafikdokument und dessen View-Liste zugreifen kann:

```
procedure TGraphicElement.InvalidateAllViews;
var
  R : TRect;
  View: TGraphicView;
  i: Integer;
```

```
begin
  for i := 0 to Document.ViewCount-1 do begin
    View := Document.Views[i];
    GetPaintRect(R, View.GetGripSizeFunc);
    InvalidateRect(R, View);
  end;
end;
```

Das Ungültigerklären selbst wird in Kapitel 6.5.3 erläutert.

Setzen der Fenstertitel

Ein weiterer Fall, in dem die View-Liste benötigt wird, tritt ein, sobald die Titelzeilen der Views aktualisiert werden müssen, etwa weil der zweite von vier Views geschlossen wurde und die im Titel angegebenen View-Nummern der letzten beiden Views um eins verringert werden müssen. In jedem Fall muss der Titel angepasst werden, wenn sich der Dateiname des Dokuments ändert. Die Titelanpassung findet daher in der Property-Schreib-Methode für das *FileName*-Property von *TGraphicDoc* statt:

```
procedure TGraphicDoc.SetFileName(const Value: string);
var
  i: Integer;
begin
  FFileName := Value;
  for i:=0 to FViews.Count - 1 do
    with TGraphicView(FViews[i]) do
      if Assigned(UpdateCaptionProc) then
        UpdateCaptionProc(Value, FViews);
end;
```

SetFileName ruft also lediglich für jeden View die in der *TGraphicView*-Struktur gespeicherte Methode zur Aktualisierung des Titels auf. Auch deren Implementation in *TDocumentForm* soll im Folgenden noch kurz betrachtet werden. Sie bekommt nämlich im zweiten Parameter die View-Liste übermittelt, damit sich das Dokumentfenster selbst in dieser Liste finden kann, um seine Nummer darin festzustellen. Diese Nummer wird allerdings nur dann im Titel angezeigt, wenn es mehr als ein Dokumentfenster in der Liste gibt (da auch das Übersichtsfenster als View in die Liste eingetragen wird, muss der Typ von *Form* auf *TDocumentForm* überprüft werden):

```
procedure TDocumentForm.UpdateCaption(NewFileName: string;
                                      ViewList: TList);
var
  i, ViewIndex, ViewCount: Integer;
begin
  ViewIndex := 0;
  ViewCount := 0;
  for i := 0 to ViewList.Count-1 do begin
    with TGraphicView(ViewList.Items[i]) do
      if Form is TDocumentForm then begin
```

```
        inc(ViewCount);
        if Form = self then ViewIndex := ViewCount;
      end;
  end;
  if ViewCount = 1 then FView.ViewNumberCaption := ''
    else FView.ViewNumberCaption := ' ['+IntToStr(ViewIndex)+']';
  // und nun Viewnummer und Dateinamen zusammensetzen:
  if (NewFileName='') then
    FCaptionVorgemerkt := 'unbenannt'+FView.ViewNumberCaption
  else FCaptionVorgemerkt := NewFileName+FView.ViewNumberCaption;
  if Showing then
    Caption := FCaptionVorgemerkt; // sonst kann es hier zu einer
      // Exception kommen, wenn das Formular noch nicht erzeugt ist.
end;
```

Falls noch kein Name für die Datei existiert, setzt diese Methode den Titel auf »unbenannt«, ansonsten auf den Dateinamen. (Bei einer Anwendung, in der das Dokumentfenster gleichzeitig auch das Hauptfenster ist, würde normalerweise vor dem Dokumentnamen noch der Name der Anwendung genannt werden.)

Weitere Funktionen einer Dokument-View-Anwendung

Abschließend sei noch auf zwei weitere Besonderheiten hingewiesen, die im TreeDesigner implementiert sind, hier aber aus Platzgründen nicht detailliert mit Code-Auszügen besprochen werden können:

- Eine LISTE DER GEÖFFNETEN DOKUMENTE können Sie aus dem Menü ANSICHT aufrufen. Ein einfaches Dialogformular mit einer Listbox und eine Schleife, in der das Hauptformular seine *FDocumentList* durchläuft, sind die Basis für diese Funktion.
- Die in Kapitel 6.2.4 gezeigte *OnCloseQuery*-Methode muss etwas erweitert werden, da beim Schließen eines Fensters nur dann eine Sicherheitsabfrage zum Speichern des Dokuments durchgeführt werden muss, wenn kein weiteres Fenster für dasselbe Dokument offen ist. Eine Ausnahme stellt ein *OnCloseQuery* beim Schließen der gesamten Anwendung dar. Hier erhalten *alle* Views eine *OnCloseQuery*-Abfrage, bevor auch nur das erste Fenster geschlossen wird. In diesem Fall muss sichergestellt werden, dass für jedes ungespeicherte Dokument genau ein Fenster eine Sicherheitsabfrage durchführt.

Die im TreeDesigner gewählte Ausarbeitung des Dokument-View-Konzepts ist sicher nicht die einzig mögliche. So gibt es auch Anwendungen, in denen man in verschiedenen Ansichten desselben Dokuments verschiedene Bereiche des Dokuments markieren kann. Die Markierung ist dort also eine Eigenschaft des Views und nicht des Dokuments. Im TreeDesigner wurde die Markierung der Einfachheit halber als Eigenschaft der einzelnen Grafikobjekte und damit als Teil des Dokuments realisiert, weshalb in allen Ansichten desselben Dokuments immer dieselben Objekte markiert sind.

Eine andere Erweiterungsmöglichkeit bestünde darin, aus einem in mehreren Views angezeigten Dokument mehrere verschiedene Dokumente zu machen, wenn das Dokument in einem der Views unter einem anderen Namen gespeichert wird. Die derzeitige Verhaltensweise des TreeDesigners führt nämlich noch dazu, dass bei einer solchen Speicherung einfach der Dokumentname des bestehenden Dokuments geändert wird, und zwar in allen angeschlossenen Views aktualisiert wird (deren Titel schließlich durch die oben abgedruckte *UpdateCaption*-Methode aktualisiert werden).

6.3.2 Eine Klasse für das Grafikdokument

Dieses Kapitel beschreibt das Dokument des TreeDesigners (definiert in der Unit *GrDoc*), zur Vereinfachung allerdings ohne Berücksichtigung der Verbindungslinien zwischen den Objekten. In den folgenden Quelltext-Auszügen wurden also die Zeilen weggelassen, die die Verbindungslinien betreffen. Informationen zu den Verbindungen finden Sie in Kapitel 6.3.4.

Ganz allgemein besteht das Grafikdokument aus einer Datenstruktur, die eine Ansammlung von Grafikobjekten speichert. Ohne selbst eine neue Datenstruktur schreiben zu müssen, haben wir hier die Auswahl zwischen den Arrays der Sprache Pascal und den Listenobjekten der CLX.

Im TreeDesigner fiel die Wahl auf die Listen, da diese im Programm erheblich einfacher zu handhaben sind: Wenn ein Objekt beispielsweise gelöscht wird, verwenden Sie einfach die Methode *TList.Delete* und müssen nicht die Lücken füllen oder verwalten, die bei einer Löschoperation in einem Array entstehen würden.

Ableiten einer spezialisierten Listenklasse

Der TreeDesigner verwendet hierzu also die Klasse *TList*. Das Grafikdokument könnte nun in einer neuen Klasse realisiert werden, die ein *TList*-Objekt enthält:

```
type
  TGraphicDoc = class
    Elements: TList;
    ...
```

Einfacher ist jedoch die im TreeDesigner gewählte Möglichkeit, die Klasse *TGraphicDoc* direkt von der CLX-Klasse *TList* (Kapitel 4.1.3) abzuleiten. *TGraphicDoc* deklariert zwar auch ein paar neue Variablen, diese werden jedoch nur als Hilfsvariablen zur Laufzeit verwendet. Der gesamte Inhalt des Dokuments befindet sich in der von *TList* verwalteten Liste, deren Elemente über das geerbte Property *Items* angesprochen werden können. Dies ändert sich übrigens auch dann nicht, wenn die Verbindungen zwischen den Objekten hinzukommen, denn diese werden als Bestandteil der einzelnen Objekte gespeichert, ohne dass sich die Liste darum kümmern muss. Fast alle von *TGraphicDoc* neu deklarierten Elemente sind daher Properties und Methoden:

```
type
  TGraphicElement = class;
  TShapeType = (stNone, stRect, stEllipse, stLine,
                stHexagon, stRhomb);
  TGraphicDoc = class(TList)
    private
    (* interne Property-Daten *)
      FPaintBox: TPaintBox;
      FWidth, FHeight: integer;
      FFileFormatVersion: char;
      FViews: TList;  { darstellende Views }
      FMarkedElementCount: integer;
      FFileName: string;
      FChangeCount: integer;
    (* Property-Zugriffsmethoden *)
      procedure SelectedSetText(s: string);
      procedure SelectedSetFontName(s: string);
      procedure SelectedSetFontSize(s: Integer);
      procedure SelectedSetFont(F: TFont);
      procedure SelectedSetPenWidth(w: Integer);
      procedure SelectedSetFontColor(c: TColor);
      procedure SelectedSetBrushColor(c: TColor);
      procedure SelectedSetPenColor(c: TColor);
      procedure SelectedSetShape(s: TShapeType);
      function GetGraphicElement(index: Integer): TGraphicElement;
      function GetViews(Index: Integer): TGraphicView;
      function GetViewCount: Integer;
      procedure SetFileName(const Value: string);
    public
    (* Konstruktor und Destruktor *)
      constructor Create;
      destructor Destroy; override;
    (* View-Verwaltung *)
      procedure AddView(AView: TGraphicView);
      procedure DeleteView(AView: TGraphicView);
      property Views[Index: Integer]: TGraphicView read GetViews;
      property ViewCount: Integer read GetViewCount;
    (* Die Markierung betreffende Properties und Methoden *)
      property MarkedElementCount: integer read FMarkedElementCount;
      procedure SelectRect(R: TRect);
      procedure DeleteSelected;
      function AnySelected: Boolean;
      procedure DeselectAll; { alle deselektieren }
    (* Elemente freigeben *)
      procedure FreeElement(Item: TGraphicElement);
      procedure FreeAll;
    (* Input/Output *)
      procedure SaveToStream(Stream: TStream; OnlyMarked: boolean;
                     SaveFormatVersion: char = CurrentFileVersion);
      procedure SaveToFile(Name: string);
      procedure LoadFromStream(Stream: TStream);
```

```
    procedure LoadFromFile(Name: string);
    // weitere Ein- und Ausgabemethoden für XML siehe Kapitel 6.9.3
  (* Properties *)
    property Shape: TShapeType write SelectedSetShape;
    property Text: string write SelectedSetText;
    property FontColor: TColor write SelectedSetFontColor;
    property FontSize: Integer write SelectedSetFontSize;
    property FontName: string write SelectedSetFontName;
    property PenWidth: Integer write SelectedSetPenWidth;
    property PenColor: TColor write SelectedSetPenColor;
    property BrushColor: TColor write SelectedSetBrushColor;
    property Items[index: Integer]: TGraphicElement
                                        read GetGraphicElement;
    property Width: integer read FWidth write FWidth;
    property Height: integer read FHeight write FHeight;
    property FileFormatVersion: char read FFileFormatVersion
                                    write FFileFormatVersion;
    property ChangeCount: integer read FChangeCount;
    property FileName: string read FFileName write SetFileName;
(* weitere Methoden *)
    function GetObjectByPoint(P: TPoint): TGraphicElement; overload;
    function GetObjectByPoint(P: TPoint; GripSize: Integer;
      var rm: TResizeMode) : TGraphicElement; overload;
    procedure PaintAll(Dest: TCanvas);
    procedure GetPaintRect(var R: TRect);
    procedure Scale(B, H: Extended);
  (* Einige in TreeDesigner 2.5 und 3.0 neu hinzugekommene Methoden
     sind hier der Übersicht halber nicht aufgeführt. *)
end;
```

Wie die folgenden Abschnitte zeigen werden, erleichtern die zahlreichen Properties der Klasse *TGraphicDoc* den Umgang mit einem Objekt dieser Klasse (also einem Dokument) erheblich.

Nicht lesbare Properties *R146*

Die öffentlichen Properties von *TGraphicDoc* sind zum großen Teil relativ ungewöhnlich, da sie zwar beschrieben, aber nicht gelesen werden können. Mit dem Schreiben in diese Properties erreichen Sie, dass alle Grafikobjekte des Dokuments, die gerade markiert sind, mit einem neuen Attribut versehen werden. Schreiben Sie z.B. in das Property *BrushColor* des *TGraphicDoc*-Objekts einen Wert, so ändern Sie damit das Property *Brush.Color* aller ausgewählten Grafikelemente.

Im TreeDesigner findet diese Zuweisung beim *OnBrushColorChange*-Event der Farbpalette statt. Das Dokumentformular hat hier nur eine Zeile Code auszuführen:

```
procedure TDocumentForm.BrushColorChange(Sender : TObject; Color: TColor);
begin
  Document.BrushColor := Color;
end;
```

Sehen wir uns als Beispiel die Methode an, die den Schreibzugriff auf dieses Property ausführt:

```
procedure TGraphicDoc.SelectedSetBrushColor(c: TColor);
var
  i: Integer;
begin
  if MarkedElementCount > 0 then begin
    inc(FChangeCount);
    for i := 0 to Count-1 do
      with TGraphicElement(Items[i]) do
        if Marked then
          Brush.Color := c;
  end;
end;
```

Die Methode verwendet zwei von *TList* geerbte Properties: *Count*, das die Anzahl der Listenelemente enthält, und *Items*, das Array, in dem diese gespeichert sind. (Um diese Liste zu erweitern, verwenden andere Teile des Programms entsprechend die ebenfalls von *TList* geerbte Methode *Add*.)

Die anderen in der obigen Methode verwendeten Bezeichner gehören zur Klasse *TGraphicElement*, die in Kürze vorgestellt wird: *Marked* zeigt an, ob ein Element selektiert ist, *Brush* enthält den Pinsel, mit dem sich das Objekt zeichnet, und *Invalidate* bewirkt das Neuzeichnen des Elements.

MarkedElementCount ist ein Property des Dokumentobjekts; es enthält die Zahl der insgesamt markierten Elemente. Auch *FChangeCount* gehört zum Dokument, das darin die Zahl der seit dem letzten Speichern vorgenommenen Änderungen hochzählt.

Ein Typenumwandlungs-Property *R145*

Wie Sie schon am letzten Beispiel sehen konnten, können Sie mit einem *TGraphicDoc*-Objekt so arbeiten wie mit einem *TList*-Objekt, da die Verwaltung der Listenelemente vollständig von *TList* geerbt ist. Die oben gezeigte Methode *TGraphiDoc.SelectedSetBrushColor* zeigt jedoch eine kleine Unbequemlichkeit: Da *TList* nichts über den Typ der enthaltenen Elemente weiß, gibt sie die Elemente als *Pointer* zurück, so dass der Aufruf *Items[i]* zuerst in ein *TGraphicElement* zurückverwandelt werden muss. Aus diesem Grund definiert *TGraphicDoc* das indizierte Property *Items* neu (überschreibt es) und gibt seinen Elementen den Typ *TGraphicElement*:

```
{ Die Deklaration des Properties:
  property Items[index: Integer]: TGraphicElement
                                    read GetGraphicElement; }

function TGraphicDoc.GetGraphicElement(index: Integer)
                           : TGraphicElement;
begin
   { Endlos-Rekursion rekursion vermeiden -> Umwandeln in TList }
  Result := TGraphicElement(TList(self).Items[index]);
end;
```

Die Property-Lesemethode tut also nichts anderes, als den Typ des von *TList* zurückgegebenen Objekts umzuwandeln, so wie es oben die Methode *SelectedSetBrushColor* noch selbst gemacht hat. Wichtig ist dabei jedoch, dass sich das Objekt in dieser Methode erst in den Typ *TList* umwandelt (man könnte auch sagen »zurückverwandelt«, denn eigentlich handelt es sich ja um ein spezialisiertes *TGraphicDoc*-Objekt). Wenn es das nicht täte, würde der Lesezugriff auf das Property *Items* dazu führen, dass sich die Methode selbst aufruft, und zwar in einer endlosen Schleife. Die Umwandlung von sich selbst (*self*) in ein *TList*-Objekt führt dazu, dass der Compiler die *Items*-Lesemethode von *TList* aufruft, die ansonsten von der Neudeklaration des Properties *Items* und dessen Lesemethode *GetGraphicElement* verdeckt wäre.

Die obige Typenumwandlungsmethode wird also bei jedem normalen Lesezugriff auf *TGraphicDoc.Items* angesprochen. Daher war die Typenumwandlung von *Items[i]* in der Methode *SelectedSetBrushColor* unnötig. Die Methode *SelectedSetFontName* verzichtet auf diese Typenumwandlung und nutzt damit die Neudefinition des Properties *Items*:

```
procedure TGraphicDoc.SelectedSetFontName (s: string);
var
  i: Integer;
begin
  if MarkedElementCount > 0 then begin
    inc(FChangeCount);
    for i := 0 to Count-1 do
      with Items[i] do
        if Marked then
          Font.Name := s;
  end;
end;
```

Bemerkenswert ist noch, dass die Methode *GetGraphicElement* keinerlei Überprüfung dessen durchführt, ob das Listenelement wirklich den Typ *TGraphicElement* hat, in den es umgewandelt wird. Hätte es einmal einen anderen Typ, so käme es möglicherweise zu einem nicht leicht lokalisierbaren Programmfehler. Wir können uns aber an dieser Stelle darauf verlassen, dass das Programm nur Objekte des Typs *TGraphicElement* über die geerbte Methode *Add* zur Liste hinzufügt und dass daher die Abfrage des *Items*-Properties immer ein solches Objekt zu Tage fördert (Zugriffe außerhalb der Listengrenzen führen nicht zu einem *nil*-Wert, sondern zu einer Exception).

Weitere Klassenelemente

Die Schreibmethoden für die Properties *Shape*, *Text*, *FontColor*, *FontSize*, *FontName*, *PenColor*, *PenWidth* und *BrushColor* funktionieren ähnlich wie die Methode *SelectedSetFontName*, weshalb sie nicht abgedruckt zu werden brauchen. Ihre Bedeutung stimmt mit den entsprechenden Eigenschaften der Klasse *TGraphicElement* überein, die im nächsten Kapitel erläutert wird.

Die Properties *Height* und *Width* geben an, wie viele Einheiten die Zeichenfläche beinhalten sollte, auf der das Dokument dargestellt wird. Diese Größenangabe wird in keiner der Methoden von *TGraphicDoc*, sondern nur in den Formularmethoden berücksichtigt. Das Property *FileFormatVersion* wird während des Ladens einer Datei verwendet, um die Version des Dateiformats zu speichern (zur *Versionskennung* siehe Kapitel 6.3.3). Die Properties zur View-Verwaltung und *FileName* wurden bereits in Kapitel 6.3.1 vorgestellt.

Was bleibt, sind die folgenden Methoden von *TGraphicDoc*:

Element	**Bedeutung**
constructor Create	initialisiert *FWidth*, *FHeight* und *FViews* (ruft zuerst natürlich den geerbten Konstruktor auf).
destructor Destroy	gibt alle Grafikelemente und die Liste *FViews* frei (die Freigabe der Grafikelemente wird nicht automatisch durch den geerbten Listen-Destruktor erledigt, da *TList* die Listenelemente nicht als Objekte mit virtuellem *Destroy*-Destruktor, sondern nur als Zeiger speichert). Wird beim *OnDestroy*-Ereignis des Dokumentformulars aufgerufen.
procedure SelectRect	wählt alle Elemente, die sich innerhalb des als Parameter übergebenen Rechtecks befinden, aus (setzt deren *Marked*-Property auf *True*).
procedure DeleteSelected	löscht alle ausgewählten Elemente.
function AnySelected	überprüft, ob irgendwelche Elemente selektiert sind.
procedure DeselectAll	hebt die Auswahl aller Elemente auf.
procedure FreeElement	entfernt ein Element mit *TList.Remove* aus der Liste und gibt es mit *Free* frei.
procedure FreeAll	leert die Liste und gibt alle Grafikelemente mit *Free* frei.
procedure SaveToFile	erzeugt eine neue Datei oder überschreibt eine bestehende Datei und speichert darin den gesamten Dokumentinhalt, benutzt dafür die Methode *SaveToStream*.
procedure SaveToStream	speichert den gesamten Dokumentinhalt oder nur die selektierten Grafikelemente in einem Stream, der eine Datei sein kann oder ein MemoryStream (für das Kopieren der Daten in die Zwischenablage)
procedure LoadFromFile	liest das Dokument aus einer bestehenden Datei ein. Zuvor werden alle bestehenden Elemente mit *FreeAll* gelöscht.

Element	Bedeutung
ocedure LoadFromStream	lädt Grafikelemente aus einem Stream.
function GetObjectByPoint	findet das angeklickte Objekt und den angeklickten Markierungspunkt heraus (siehe Kapitel 6.4.2).
ocedure PaintAll	ruft die *Paint*-Methode aller Grafikelemente auf.
ocedure Scale	skaliert die gesamte Grafik, wobei die Faktoren für horizontale und vertikale Skalierung in den beiden Parametern angegeben werden.
ocedure GetPaintRect	liefert die Koordinaten des kleinsten Rechtecks, mit dem alle Grafikelemente des Dokuments überdeckt werden könnten. Diese Methode wird sowohl beim Exportieren des Dokuments als Metadatei als auch beim Drucken dazu benötigt, einen optimalen Ausschnitt aus der möglicherweise viel zu großen Zeichenfläche zu wählen.

Bevor wir nun die neu gezeichneten Objekte mit *TList.Add* hinzufügen können, benötigen wir eine zweite Klasse, in der diese Objekte definiert werden.

6.3.3 Eine Klasse für die Grafikobjekte

In einer ganz einfachen Anwendung, in der Sie beispielsweise nur Ellipsen und Kreise zeichnen können, deren Farbe und Umrandung vorgegeben ist, würde es genügen, für jedes Grafikobjekt einfach nur eine *TRect*-Struktur zu verwenden, in der das Rechteck gespeichert ist, das die Ellipse bzw. den Kreis umgibt.

Die Objekte im TreeDesigner sollen jedoch auch farbig gestaltbar sein und verschiedene Formen sowie eine Beschriftung haben können, daher benötigen wir hier mindestens eine eigene Datenstruktur, besser natürlich eine eigene Klasse. Diese Klasse heißt *TGraphicElement*, sie definiert alle zu einem Grafikelement gehörenden Daten, darunter auch eine *TRect*-Struktur für die Koordinaten.

berlegungen zur Klassenstruktur

or der Implementierung der verschiedenen Grafiktypen gilt es, eine wichtige Entscheidung zur Programmstruktur zu treffen. Es konkurrieren hauptsächlich zwei Methoden, die unterschiedlichen Grafiktypen (Rechteck, Kreis etc.) in *TGraphicElement* zu vereinbaren:

- Für jeden Grafiktyp könnte eine eigene Klasse angelegt werden. Dies ist auf den ersten Blick die natürlichere Vorgehensweise. Jede der von *TGraphicElement* abgeleiteten Klassen könnte dann durch Überschreiben der Methode *Paint* große Freiheiten erreichen.
- Die zweite Möglichkeit ist, alle Grafiktypen bereits in der Klasse *TGraphicElement* abzudecken. Um welchen Grafiktyp es sich handelt, müsste dann in einer Variablen

gespeichert werden. Dies scheint auf den ersten Blick ein nicht ganz OOP-gemäßer Trick zu sein, da er die Klasse *TGraphicElement* mit der Arbeit vieler einzelner Klassen (Typen grafischer Elemente) belastet.

Indem Sie eine solche *TGraphicElement*-Klasse jedoch nicht ansehen als eine Klasse, die die Arbeit verschiedener speziellen Klassen übernimmt, sondern als eine Klasse, die eine rechteckige Fläche mit verschiedenen Formen auszufüllen vermag, kommen Sie jedoch wieder mit der OOP-Philosophie ins Reine. (Tatsächlich können Sie die Form der Grafikobjekte ja auch zur Laufzeit ändern, indem Sie eine andere Form von den Schaltern der Werkzeugleiste auf das zu ändernde Element ziehen.)

Der größte Vorteil der zuletzt genannten Möglichkeit, die im TreeDesigner auch realisiert ist, kommt erst in abgeleiteten Klassen zum Tragen: Angenommen, Sie wollten aus irgendeinem Grund sowohl Rechtecke als auch Ellipsen und die anderen Grafiktypen um eine besondere Funktionalität erweitern, indem Sie eine neue Klasse ableiten. Hätte jeder Grafiktyp seine eigene Klasse, so müssten Sie von jeder dieser Klassen eine neue Klasse ableiten und in dieser jeweils die neue (identische) Funktionalität implementieren, was durch die enorme Redundanz eine völlig unpraktikable Lösung wäre. Sind aber alle Grafiktypen in einer Klasse vereint, würde es genügen, eine einzige neue Klasse davon abzuleiten.

Dies soll selbstverständlich nicht heißen, dass Klassen immer mit möglichst viel Funktionalität vollgestopft sein sollten. Da sich die Unterschiede zwischen den Grafikobjekten jedoch nur auf die Form und damit die Variable *ShapeType* beschränken, lohnt sich die Zusammenfassung zu einer Klasse in diesem Fall.

Hinweis: Der TreeDesigner definiert selbst bereits zwei von *TGraphicElement* abgeleitete Klassen, die jedoch hier schon aus Platzgründen nicht weiter beschrieben werden können: *TGradientObject* definiert die Farbverlauf-Grafikelemente, die Sie durch den untersten Schalter der Werkzeugleiste am linken Fensterrand erhalten; die Klasse *TMultiLineElement* ist für die mehrzeilige Textausgabe zuständig, die in der Darstellung einer XML-Datenstruktur verwendet wird.

Übersicht über TGraphicElement

Ohne die in Kapitel 6.3.4 hinzukommenden Verbindungen zu anderen Objekten und ohne einige Erweiterungen der TreeDesigner-Versionen 2.5-3.5 sieht die Deklaration der Klasse *TGraphicElement* folgendermaßen aus:

```
type
  TShapeType = (stNone, stRect, stEllipse, stLine,
                stHexagon, stRhomb, stRoundRect);
  { In der Markierung eines Rechtecks werden acht Größenänderungspunkte
    hervorgehoben, die den folgenden Größenänderungs-Modi entsprechen: }
```

```
  TResizeMode = (rmNone, rmRB, rmLB, rmRT, rmLT, rmR, rmL, rmT, rmB);
  TGrObjectText = string[50]; // "historische" Beschränkung auf 50 Zeichen
    // (könnte auch auf einen normalen String variabler Länge erweitert
    // werden, dieser würde jedoch ein neues Dateiformat erfordern.)
  TGraphicElement = class(TPersistent)
    { Die Klasse zieht zurzeit keinen Vorteil aus ihrer
      Vorgängerklasse TPersistent. }
  private
  (* persistente Daten (werden in Datei gespeichert) *)
    Points: TRect;
    FShapeType: TShapeType;
    FText: TGrObjectText;
    Content: TElementContent; { Die Klasse TElementContent kapselt einen
       eventuellen zusätzlichen Inhalt des Grafikelements, beispielsweise
       eine Bitmap oder eine Metadatei (zur Anwendung siehe Kap. 6.1.3).
       Wird in den folgenden Quelltextauszügen nicht weiter beachtet. }
    (* Dies sind noch nicht alle persistenten Daten: Auch einige Attribute
       der öffentlichen Objekte Pen, Brush und Font werden gespeichert! *)
  (* Laufzeitvariablen *)
    FMarked: Boolean;
    Document: TGraphicDoc;
  (* Property-Zugriffsmethoden *)
    procedure SetMarked(val: Boolean);
    function GetMarked: Boolean;
    procedure SetShape(s: TShapeType);
    function GetCoord(Index: Integer): Integer;
    procedure SetText(s: TGrObjectText);
    procedure GraphicsObjectChanged(Sender: TObject);
  public
    Pen: TPen;
    Brush: TBrush;
    Font: TFont;

    property ShapeType: TShapeType read FShapeType write SetShape;
    property Text: TGrObjectText read FText write SetText;
    property Marked: Boolean read GetMarked write SetMarked;

    property Left: Integer index 1 read GetCoord;
    property Right: Integer index 2 read GetCoord;
    property Top: Integer index 3 read GetCoord;
    property Bottom: Integer index 4 read GetCoord;

  (* Konstruktoren, Destruktoren, Speichern und Laden: *)
    constructor Create(Doc: TGraphicDoc; InitRect: TRect;
                       SType: TShapeType); virtual;
    destructor Destroy; override;
    procedure LoadFromStream(Stream: TStream);
    procedure StoreToStream(Stream: TStream; Index: Integer);
  (* Veränderung der Koordinaten und Verschieben des Objekts *)
    procedure SetNewRect(NewRect: TRect);
    procedure Move(XTranslate, YTranslate: Integer);
```

```
  (* Methoden im Zusammenhang mit dem Zeichnen *)
    procedure Paint(CanvasRef: TCanvas); virtual;
    procedure PaintMark(CanvasRef: TCanvas);
    procedure Invalidate(View: TGraphicView);
    procedure InvalidateRect(var R : TRect; View: TGraphicView);
    procedure InvalidateAllViews;
   (* Zeichnen während einer Drag-Operation *)
    procedure XorPaint(CanvasRef: TCanvas;
       XorRect: TRect); virtual;
   (* Informationsmethoden über die verbrauchte Fläche *)
    procedure GetOriginalRect(var R: TRect);
    procedure GetNormalizedRect(var R: TRect);
    procedure GetPaintRect(var R: TRect);
    procedure GetInnerRect(var R: TRect);
    function PointInShape(TestPoint: TPoint): Boolean; overload;
    function PointInShape(TestPoint: TPoint;
                          var rm: TResizeMode): Boolean; overload;
    function ContainedIn(R: TRect): Boolean;
  end;
```

Die Klassenelemente bestehen aus verschiedenen Gruppen:

- Private Daten: Zu diesen gehören die Property-Variablen *FMarked*, *FShapeType* und *FText*, auf die von außen nur über die Properties zugegriffen wird, die Variable *Document*, die auf das *TGraphicDoc*-Dokument weist, in dem das Objekt enthalten ist, und *Points*, eine *TRect*-Struktur, die die Koordinaten des Objekts enthält. Letztere beiden Variablen werden bereits im Konstruktor initialisiert. Während *Document* normalerweise nur einmal gesetzt wird, können Sie die *Points* durch *SetNewRect* (siehe Tabelle) jederzeit ändern.
- Private Methoden: Diese dienen nur dem Zugriff auf die Properties und brauchen nicht einzeln erläutert zu werden.
- Grafikattribute: Zwei der Grafikattribute werden über Properties angesprochen: die grafische Form des Elements (*ShapeType*) und der Beschriftungstext (*Text*). Die Farben, die Liniendicke und die Schriftart werden in eigenen Objekten gespeichert: *Pen*, *Brush* und *Font*. Diese sind nicht als Properties implementiert, da sie nicht als Ganzes neu gesetzt werden. Vielmehr werden immer nur einzelne Attribute dieser Objekte gesetzt, wie oben z.B. in der Methode *TGraphicDoc.SelectedSetFontName*.
- Weitere Properties: Das Property *Marked* gibt an, ob das Element markiert ist oder nicht. *Left*, *Right*, *Top* und *Bottom* erlauben den Lesezugriff auf die einzelnen Koordinatenbestandteile. Sie sind ein Beispiel für Properties mit gemeinsamer Zugriffsmethode (trotz des Wortes *index* in ihrer Deklaration nicht zu verwechseln mit den indizierten Properties, bei denen es sich um die Array-Properties handelt). Ändern lassen sich die vier Werte nur gemeinsam über die Methode *SetNewRect*.

Die öffentlichen Methoden sind in der folgenden Tabelle zusammengefasst.

Methode	Beschreibung
Create	Konstruktor, überschreibt den geerbten Konstruktor, initialisiert die Koordinaten, die Form und den Zeiger auf das *TGraphicDoc*-Dokument.
Destroy	Destruktor, gibt den dynamisch belegten Speicher frei.
LoadFromStream	liest die persistenten Daten des Objekts aus einem Stream.
veToStream	speichert die persistenten Daten in einem Stream.
XorPaint	übernimmt die Ausgabe des Objekts bei Ziehoperationen mit der Maus.
Paint	zeichnet das komplette Grafikelement in das übergebene *TCanvas*-Objekt.
PaintMark	zeigt die Markierung des Elements am Bildschirm an.
validateRect	weist ein View an, einen bestimmten Bereich neu zu zeichnen (Hilfsmethode für *InvalidateAllViews*).
validateAllViews	bewirkt den Aufruf von *InvalidateRect* für alle Views, die an das übergeordnete Dokument angeschlossen sind (ein Verweis auf dieses Dokument befindet sich ja in der privaten Variablen *Document*).
Move	verschiebt das Element um bestimmte Beträge in horizontaler und vertikaler Richtung.
SetNewRect	ändert die Koordinaten und sorgt für eine Neuausgabe der Grafik an den veränderten Stellen.
GetOriginalRect	liefert die unveränderten Koordinaten, die in *Points* angegeben sind.
GetNormalizedRect	liefert das Originalrechteck, vertauscht die Koordinaten aber so, dass *Bottom>Top* und *Right>Left*.
GetPaintRect	liefert das Rechteck, das durch das Objekt belegt wird; zum Original-Rechteck kommen hier evtl. überstehende Umrandungslinien hinzu (wenn der Stift breiter als 1 Pixel ist).
GetInnerRect	liefert das innere Rechteck ohne die Umrandungslinien; dieses Rechteck entspricht dem inneren Bereich des Objekts, falls *ShapeType=stRect*.
ointInShape	entscheidet, ob der übergebene Punkt im Bereich des Grafikobjekts ist, sieht das Grafikobjekt jedoch immer als Rechteck an. Zur richtigen Behandlung von Linien und anderen Formen wären einige Erweiterungen dieser Methode notwendig. Die zweite (überladene) Version dieser Methode mit zusätzlichem *TResizeMode*-Parameter prüft auch, ob der Punkt in einer Randmarkierung zur Änderung der Größe liegt, und, wenn ja, welche Art der Größenänderung dies bedeutet.
ContainedIn	entscheidet, ob das Grafikelement innerhalb eines Rechtecks liegt – diese Methode funktioniert für alle geometrischen Formen gleichermaßen gut.

Einige dieser Methoden könnten übrigens durchaus auch als private Methoden deklariert werden, denn sie werden von außen nicht aufgerufen, so z. B. *PaintHexagon* und *PaintRhomb*.

Änderung der Zeichenattribute

Dadurch, dass die meisten Attribute von *TGraphicElement* als Properties implementiert sind, können die Methoden von *TGraphicDoc* diese Attribute durch einfache Zuweisung eines neuen Wertes ändern (z.B. durch die *TGraphicDoc.SelectedSetFontName* in Kapitel 6.3.2). Die Schreibmethode für diese Properties führt das erforderliche Neuzeichnen automatisch durch (wie *Invalidate* funktioniert, lesen Sie in Kapitel 6.5.1):

```
procedure TGraphicElement.SetShape(s: TShapeType);
begin
  FShapeType := s;
  InvalidateAllViews;
end;
```

Der TreeDesigner ermöglicht es dem Benutzer über die Werkzeugleiste lediglich, Einfluss auf die Properties *Pen.Width*, *Brush.Color*, *Pen.Color*, *Font.Color*, *Font.Size* und *Font.Name* zu nehmen, die anderen Properties von *Pen*, *Brush* und *Font* bleiben ungenutzt.

Allerdings sind die Objekte *Pen*, *Brush* und *Font* keine Properties. Dies wäre hier auch nicht sinnvoll, da nicht Stift, Pinsel und Schrift als Ganzes, sondern nur einzelne ihrer Attribute geändert werden. Die Frage ist nun, wie eine Änderung dieser Attribute zu einem automatischen Neuzeichnen führen kann wie in der obigen Methode *SetShape*. Offenbar sind wir hier an den Grenzen der Properties angelangt, da durch die Zeile

```
Objekt.Brush.Color := clLightRed;
```

zwar das *Brush*-Objekt selbst über die Änderung informiert wird, das *TGraphicElement* jedoch keine Möglichkeit hat, das Setzen des Properties mit einer eigenen Methode zu verknüpfen.

Die OnChange-Ereignisse nicht-visueller Objekte

Borland hat dies bei der Entwicklung der CLX berücksichtigt und vielen nicht-visuellen Objekten ein *OnChange*-Ereignis mit auf den Weg gegeben. So lösen Stifte, Pinsel und Schriftobjekte immer dann ein *OnChange*-Ereignis aus, wenn eines ihrer Properties mit einem neuen Wert beschrieben wird. Die Klasse *TCanvas* verwendet diese Ereignisse beispielsweise, um ihr eigenes *OnChange*-Ereignis auszulösen.

Ähnlich wie *TCanvas* enthält auch *TGraphicElement* drei Objekte *Pen*, *Brush* und *Font*. Damit das Grafikobjekt neu ausgegeben wird, wenn sich eines der Attribute ändert, verknüpft es die folgende einfache Methode mit den *OnChange*-Ereignissen aller drei Objekte:

```
procedure TGraphicElement.GraphicsObjectChanged(Sender: TObject);
begin
  Document.IncChangeCount; // erhöht privaten Zähler Document.FChangeCount
```

```
    InvalidateAllViews;
    ElementChangeNotify;
  end;
```

Die Verknüpfung findet im Konstruktor statt, dem wir uns als Nächstes zuwenden werden.

onstruktor, Destruktor und dynamische Objekte *R155*

Die Vorgänge im Konstruktor der Klasse *TGraphicElement* sind ein gutes Beispiel für den dynamischen Aufbau von Objekten:

```
constructor TGraphicElement.Create(AList: TGraphicDoc;
                        InitRect: TRect; SType: TShapeType);
begin
  inherited Create;
  Document := Doc;
  FShapeType := SType;
  Points := InitRect;
(* Konstruktion der dynamischen Objekte: *)
  Pen := TPen.Create;
  Brush := TBrush.Create;
  Font := TFont.Create;
  Pen.OnChange := GraphicsObjectChanged;
  Brush.OnChange := GraphicsObjectChanged;
  Font.OnChange := GraphicsObjectChanged;
{ für die von diesem Objekt ausgehenden Verbindungen kommt
  später noch hinzu: }
{  Edges := TList.Create;   }
  { die anderen Daten werden automatisch auf 0 gesetzt. }
end;
```

Der Code des Konstruktors *Create* erfüllt drei verschiedene Aufgaben:

- Er ruft den geerbten Konstruktor auf, damit sich die Basisklasse initialisieren kann.
- Da alle Variablen bereits automatisch mit Nullen initialisiert werden, braucht er nur einige Werte zu setzen, deren Voreinstellung von 0 verschieden ist, wie *FShapeType*, *Document* und *Points*, also genau die Werte, die als Parameter im Aufruf des Konstruktors enthalten sind.
- Er erzeugt durch weitere Konstruktoraufrufe die dynamischen Objekte, die im Grafikelement enthalten sind, und initialisiert diese (explizit initialisiert werden müssen lediglich deren Verknüpfungen zu den *OnChange*-Ereignissen – wie im letzten Abschnitt besprochen).

Die wichtigste Aufgabe des Konstruktors, das Reservieren des Speichers für das *TGraphicElement*-Objekt selbst, wird automatisch ausgeführt, und zwar schon bevor die erste Zeile des Konstruktors erreicht wird (zu weiteren Details von Konstruktoren siehe Kapitel 2.3.1).

Der dynamische Aufbau eines Grafikobjekts beginnt damit, dass die in Kapitel 6.4.1 noch folgende Methode zum Event *OnMouseDown* ein neues *TGraphicElement* erstellt:

```
MouseObject := TGraphicElement.Create(
         Document, { so heißt das TGraphicDoc-Objekt des Formulars }
         Rect(x, y, x, y),  { das Anfangsrechteck ist ein Punkt   }
         CurShapeType);  { CurShapeType speichert die Auswahl des
                         Benutzers in den Schaltern der Mauspalette }
```

Was jetzt noch fehlt, ist der Destruktor. Er muss lediglich die dynamischen Objekte freigeben und den geerbten Destruktor aufrufen:

```
destructor TGraphicElement.Destroy;
begin
  Pen.Free;
  Brush.Free;
  Font.Free;
{ später kommt hinzu: }
{  Edges.Free; }
  inherited Destroy;
end;
```

Speicherverbrauch

Der Aufruf von *TGraphicElement.Create* belegt in der fertigen Version (mit den Verbindungen zwischen den Objekten) bereits 132 Bytes für die Variablen, die in *TGraphicElement* deklariert sind (dieser Wert wurde über die Methode *InstanceSize* ermittelt). Der Konstruktor erzeugt dann die Stift-, Pinsel- und Schriftobjekte, die weitere 36, 40 und 48 Bytes benötigen, sowie die Liste der Verbindungen, die ähnlich viel Platz verbraucht. Nicht mitgerechnet sind allerdings Speicherbereiche, die in den Klassen *TList*, *TBrush*, *TFont* und *TPen* dynamisch reserviert werden.

Eine effektivere Möglichkeit, den Speicherbedarf zu reduzieren, bestünde darin, die Beschriftung der Objekte dynamisch zu verwalten (z.B. mit den neuen langen Strings) und die Farben, die Liniendicke und den Namen der Schrift in einzelnen Variablen zu speichern statt innerhalb der Objekte *Pen*, *Brush* und *Font*:

```
BColor, PColor, FColor: TColor;
PWidth: Integer;
FontName: string;
```

Solange der TreeDesigner weniger als 1000 oder 10000 Objekte verwalten soll, ist der dadurch entstehende Zusatzaufwand jedoch kaum notwendig.

Dateioperationen *R1I4*

Die nächste wichtige Aufgabe, die *TGraphicElement* zu erfüllen hat, ist, seine Daten in einen Stream zu schreiben, und zwar so, dass sie später genauso wieder zurückgelesen werden können.

Hinweis: Die tatsächlichen Methoden des TreeDesigners weichen in Teilen von den hier gezeigten Methoden ab, um das Dateiformat kompatibel zur 16-Bit-Version zu halten. Auf diese Kompatibilitäts- und Portierungsfragen geht die Datei D1und2.rtf auf der CD näher ein.

Zuständig für die einzelnen Objekte sind die *TGraphicElement*-Methoden *SaveToStream* und *LoadFromStream* (zu den einzelnen Methoden von *TStream* siehe Kapitel 4.3):

```
procedure TGraphicElement.SaveToStream(Stream: TStream;
     Index: Integer); (* Index wird für das Speichern der Verbindungen
           benötigt, siehe "Speichern von Zeigern" in Kapitel 6.3.4. *)
  procedure WriteColorToStream(s: TStream; c: TColor);
  begin
    S.Write(c, sizeof(c));
  end;

var
  i: Integer;
  s: string[LF_FACESIZE - 1];
begin
  { einfach zu schreibende Daten }
  Stream.Write(Points, sizeof(Points));
  Stream.Write(FShapeType, sizeof(FShapeType));
  Stream.Write(FText, length(FText)+1);
  { Farben }
  WriteColorToStream(stream, Pen.Color);
  WriteColorToStream(stream, Brush.Color);
  WriteColorToStream(stream, Font.Color);
  { umständlich zu schreibende Daten, weil es sich nicht
    um eigene Properties handelt: }
  i := Pen.Width;
  Stream.Write(i, sizeof(i));
  i := Font.Size;
  Stream.Write(i, sizeof(i));
  s := Font.Name;
  Stream.Write(s, length(s)+1);

  FileIndex := Index; { Für Speicherung der Verbindungen merken }
end;

procedure TGraphicElement.LoadFromStream(Stream: TStream);
var
  C: TColor;
```

```
    i: Integer;
    s: string[LF_FACESIZE - 1];;
  begin
    FMarked := False;
    Visited := False;
    { einfach zu lesende Daten }
    Stream.Read(Points, sizeof(Points));
    Stream.Read(FShapeType, sizeof(FShapeType));
    Stream.Read(FText[0], 1);
    Stream.Read(FText[1], Byte(FText[0]));
    { Farben }
    Stream.Read(C, sizeof(TColor));
    Pen.Color := C;
    Stream.Read(C, sizeof(TColor));
    Brush.Color := C;
    Stream.Read(C, sizeof(TColor));
    Font.Color := C;
    { umständlich zu lesende Daten: }
    Stream.Read(i, sizeof(i));
    Pen.Width := i;
    Stream.Read(i, sizeof(i));
    Font.Size := i;
    Stream.Read(s[0], 1);
    Stream.Read(s[1], Byte(s[0]));
    Font.Name := s;
  end;
```

SaveToStream und *LoadFromStream* schreiben und lesen nur die wichtigen Daten, speichern also nicht die kompletten Objekte *Pen*, *Brush* und *Font*, sondern wählen aus diesen die verwendeten Daten aus.

Die Properties zeigen sich in diesen Methoden von ihrer problematischen Seite, denn sie sind nicht als Variablenparameter verwendbar – und die Stream-Methoden *Read* und *Write* benötigen einen solchen Variablenparameter. Die beiden obigen Methoden müssen daher Hilfsvariablen bereitstellen, in denen sie den Wert des Properties zwischenspeichern und die sie an die Stream-Methoden übergeben können.

Die Methode *SaveToStream* geht bei den Farben einen etwas anderen Weg. Sie verwendet die lokale Prozedur *WriteColorToStream,* die keine Variablenparameter erwartet. Wenn *SaveToStream* ihr die Properties direkt übergibt, wird durch den Prozeduraufruf eine Kopie der Property-Werte auf dem Stack angefertigt, so dass *WriteColorToStream* diese Kopie als Variablenparameter an den Stream weitergeben kann.

Strings

Weitere Umstände bereitet der String zur Beschriftung. Da die obigen Methoden aus Gründen der Effektivität keine *TFiler*-Objekte verwenden, müssen sie den String per Hand speichern. Da es außerdem sehr platzverschwenderisch wäre, für jeden String

eine feste Zahl von Zeichen zu speichern, schreibt *SaveToStream* nur die tatsächlich existierenden Zeichen des Strings in die Datei. Als Erstes ist daher die Speicherung der Zeichenzahl erforderlich, die sich immer im nullten Zeichen eines kurzen Pascal-Strings befindet. Dementsprechend liest die Lesemethode zuerst die Länge des Strings, um dann im zweiten Schritt die richtige Bytemenge für alle Zeichen des Strings lesen zu können.

Dateioperationen für das Dokument

In der Klasse des Dokuments *TGraphicDoc* sorgen die Methoden *SaveToFile* und *LoadFromFile* für das Speichern der gesamten Elementliste. Um die Lese- und Schreibmethoden nicht an die Verwendung von Dateien zu binden, sondern sie auf jeden abstrakten *TStream* anzuwenden (wichtig z.B. für das Kopieren von Grafikobjekten in die Zwischenablage, siehe ***R75***), werden die eigentlichen Lese- und Schreiboperationen in die separaten Methoden *ReadFromStream* und *WriteToStream* ausgelagert, die dann von *SaveToFile* und *LoadFromFile* aufgerufen werden:

```
const
  NullRect: TRect = (Left:0; Top:0; Right:0; Bottom:0);
  CurrentFileFormatVersion = '2'; { Dateiformats-Version }
  TVFSignature: string = 'EWTD-V1'+CurrentFileFormatVersion;

procedure TGraphicDoc.SaveToFile(Stream: TStream);
var
  ObjCount: Integer;
  i: Integer;
begin
  Stream.Write(TVFSignature[1], 8); { Dateikennung schreiben }
  { Anzahl der Objekte vermerken }
  ObjCount := Count;
  Stream.Write(ObjCount, sizeof(ObjCount));
  { Schreiben der einzelnen Objekte }
  for i := 0 to Count-1 do
    TGraphicElement(Items[i]).SaveToStream(Stream, i);
end;

procedure TGraphicDoc.SaveToFile(Name : string);
var
  Stream : TFileStream;
begin
  Stream := TFileStream.Create(Name, fmCreate);
  WriteToFile(Stream, false);
  Stream.Free;
  FileName := Name;
  FChangeCount := 0;
end;

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

```
var
  ObjCount: Integer;
  i: Integer;
  NextObject: TGraphicElement;
  TestSig: string[8];
begin
  { Dateikennung und -Version überprüfen }
  Stream.Read(TestSig[1],8);
  TestSig[0] := #7;
  TVFSignature[0] := #7;
  if not ( (TestSig = TVFSignature) and  { Kennung prüfen }
           (TestSig[8] = '0'))           { Version prüfen }
    then raise EFileFormatError.Create('Fehler im TVF-Dateiformat');
  FFileFormatVersion := TestSig[8]; { Formatversion kann von den
             Lademethoden der Grafikelemente abgefragt werden }
  { Zahl der Objekte lesen }
  Stream.Read(ObjCount, sizeof(ObjCount));
  { Objekte lesen }
  for i := 1 to ObjCount do begin
    NextObject := TGraphicElement.Create(self, NullRect, stRect);
    NextObject.LoadFromStream(Stream);
    Add(NextObject);
  end;
end;

procedure TGraphicDoc.LoadFromFile(Name : string);
var
  Stream : TFileStream;
begin
  FreeAll;
  Stream := TFileStream.Create(name, fmOpenRead);
  LoadFromStream(Stream);
  DeselectAll;
  Stream.Free;
  FileName := Name;
  FChangeCount := 0;
end;
```

Versionskennung

Um ein Dokument als das eigene zu erkennen, versieht eine Anwendung eine Datei üblicherweise mit einer Kennung, die meistens noch Angaben zur verwendeten Programmversion enthält. Wenn sich die Dateiformate so von Version zu Version ändern, ist es einer neueren Version über diese Kennung möglich, das alte Format automatisch in das neue umzuwandeln. Die Kennung für die TreeDesigner-Dateien ist im String *TVFSignature* angegeben, wobei die beiden letzten Zeichen desselben die Versionsnummer angeben.

Hinweis: In den TreeDesigner-Versionen 2.5 und 3.5 wurden neue Dateiformat-Versionen eingeführt – zuletzt, um die Grafikelemente verschiedener Klassen unter Nutzung der Polymporphie zu speichern. Dateien aus TreeDesigner-Versionen vor 2.5 (bei denen die Versionskennung mit »V10« endet), können weiterhin eingelesen werden, denn die vollständige *LoadFromStream*-Methode auf der CD berücksichtigt auch das alte Format. Beim Speichern werden die Dateien quasi automatisch in das Format »V11« konvertiert und können nicht mehr mit alten TreeDesigner-Versionen gelesen werden! Das neue Format »V12« wird auch von der aktuellen TreeDesigner-Version nur erzeugt, wenn das Grafikdokument Elemente enthält, die dieses Format erfordern (Farbverlaufs- oder Mehrzeilentext-Elemente). Für eine Behandlung in diesem Buch wären diese Erweiterungen zu umfangreich, daher beschränkt sich der im Buch abgedruckte Code auf die Daten, die schon seit der Dateiformats-Version des TreeDesigner 1.0 in die Datei geschrieben werden. Wenn Sie sich für die Nutzung der Polymorphie bei der Speicherung interessieren, sei auf R115 Seite 427 verwiesen, der TreeDesigner geht nach dem dort beschriebenen Prinzip vor.

Schreiben der Objektliste

Das Schreiben der Objektliste ist aus der Sicht des Dokuments kein Problem. Damit die Lesemethode weiß, wie viele Objekte zu lesen sind, schreibt *SaveToFile* zuerst die Zahl der Objekte in die Datei (wobei das Property *Count* wieder nicht direkt als Variablenparameter an den Stream übergeben werden kann). Für jedes Objekt muss *SaveToFile* dann lediglich die Schreibmethode des Objekts aufrufen. *ReadToFile* verwendet entsprechend die Lesemethode, muss aber zuvor per Konstruktoraufruf ein neues Objekt erzeugen.

6.3.4 Baumstrukturen

Dieses Kapitel beschreibt die Erweiterungen, die die oben beschriebenen Klassen erfahren, um die Baumstruktur speichern zu können. Aus Platzgründen kann neben einer Übersicht über die neue Klasse *TEdge* und über die Erweiterungen der Klasse *TGraphicElement* nur eine vollständige Methode zum Anordnen des Baums besprochen werden.

Bedienung

Wie schon in der Einführung erwähnt, können Sie im TreeDesigner zwei Objekte verbinden, indem Sie ein Objekt markieren und das andere Objekt in Verbindung mit den Tasten ⇧ und Strg anklicken. Das zuerst markierte Objekt wird dadurch zum *Nachfolger* des als Zweites angeklickten Objekts. Diese Reihenfolge widerspricht zwar der Intuition, derzufolge der Vorgänger *vor* dem Nachfolger angeklickt werden müsste, hat aber zum Vorteil, dass Sie *mehrere* Nachfolger markieren und sie mit einem Vorgänger verbinden können. (Jedes Objekt darf beliebig viele Nachfolger, aber nur einen Vorgänger haben – so wie die Klassen in Object Pascal).

Weitere Möglichkeiten, die Grafik automatisch anzuordnen oder Teilbäume ein- und auszublenden, finden Sie im lokalen Menü der Grafikelemente. Klicken Sie dazu ein grafisches Element direkt mit der rechten Maustaste an und wählen Sie den gewünschten Menüpunkt. Das lokale Menü eines Grafikelements enthält auch Optionen, mit denen Sie die eingehenden und ausgehenden Verbindungslinien lösen können.

Knoten und Kanten

Wie jede Baumstruktur setzt sich auch die Baumstruktur des TreeDesigners aus *Knoten* und *Kanten* zusammen. Ein Knoten (engl. *Node*) ist in diesem Fall einfach eines der Grafikelemente und wird somit durch die Klasse *TGraphicElement* repräsentiert. Eine Kante (engl. *Edge*) ist die Verbindung zwischen zwei Knoten. Für sie definiert die Unit *GrDoc* eine neue Klasse:

```
TEdge = class
  StartNode, EndNode: TGraphicElement;
public
  procedure SaveToStream(Stream: TStream);
  constructor LoadFromStream(Stream: TStream; List: TList);
  constructor StandardConnection(n1, n2: TGraphicElement);
  function GetEndNode: TGraphicElement;
end;
```

Die einzigen Daten einer Kante sind also die beiden Knoten, die verbunden werden sollen. Mit dieser allgemeinen Struktur ließen sich die verschiedensten in der Informatik verwendeten Typen von Graphen definieren, im TreeDesigner müssen diese theoretischen Möglichkeiten aus Gründen der Komplexität jedoch etwas eingeschränkt werden:

- Zunächst einmal behandelt der TreeDesigner den Graphen als *gerichteten Graphen* – das heißt, dass in jeder Verbindung ein Knoten der Ausgangsknoten (*StartNode*) und der andere der Endknoten (*EndNode*) ist. Bei der Darstellung der Kassenhierarchie ist die Basisklasse jeweils der Anfangsknoten, die davon ausgehenden Verbindungen führen in *Richtung* der abgeleiteten Klassen, die die Endknoten bilden.
- Als zweite Beschränkung darf jeder Knoten nur einen Vorgängerknoten haben, so wie jede Klasse in Object Pascal nur eine Vorfahrklasse haben kann. Die Zahl der Nachfolger ist nur durch die Kapazität der CLX-Klasse *TList* beschränkt.

An diesen beiden Beschränkungen führt im TreeDesigner kein Weg vorbei. Falls Sie versuchen, einem Knoten, der schon einen Vorgängerknoten hat, einen weiteren Vorgängerknoten zuzuweisen, erzeugt die Klasse *TEdge* eine Exception.

Für die automatische Anordnung der verknüpften Elemente gelten weitere Regeln: Es werden jeweils nur die Elemente angeordnet, die von einem bestimmten, von Ihnen ausgewählten Wurzel-Element aus erreichbar sind. Darüber hinaus sollte die Grafik

keine Zyklen haben, d.h. dass kein Grafikelement sein eigener Vorgänger sein sollte. Dies ist immer sichergestellt, wenn die Grafik aus einem normalen Hierarchiebaum besteht. Da jeder Knoten nur einen Vorgänger haben darf, könnte in einem solchen Baum nur ein Zyklus entstehen, wenn Sie die Wurzel an einen ihrer Nachfolger anhängen (bei der Klassenhierarchie würde das bedeuten, die Klasse *TObject* beispielsweise von *TForm* abzuleiten).

Wenn Sie Grafikelemente manuell verknüpfen, überprüft der TreeDesigner *nicht*, ob dadurch Zyklen entstehen. Bei der Darstellung der Baumstruktur spielt es keine Rolle, ob Zyklen vorhanden sind (da die Darstellung nicht rekursiv erfolgt). Rekursiv arbeiten allerdings die Methoden zum Anordnen des Baums. Sie vermeiden eine Endlosschleife durch Verwendung der Variable *Visited* und ignorieren einen Zyklus stillschweigend.

Die Erweiterungen von TGraphicElement

Die wichtigsten neuen Datenelemente von *TGraphicElement* sind die Liste der Verbindungen zu den Nachfolgerknoten (*Edges*) und der Zeiger auf den Vorgängerknoten (Property *PrevNode*). Die Liste *Edges* enthält Objekte der Klasse *TEdge*, um also an einen Nachfolgerknoten zu kommen, müssen Sie aus einem dieser *TEdge*-Objekte das Element *EndNode* auslesen:

```
EndNode := TEdge(Edges[i]).EndNode;
```

Diese scheinbar umständliche Speicherung der Verbindungsdaten ermöglicht es, später weitere Daten mit jeder Verbindung zu speichern. So könnte die Klasse *TEdge* beispielsweise um Daten wie die Farbe der Verbindungslinie oder die Option, einen Pfeil statt einer Linie zu zeichnen, erweitert werden.

Damit man auch ohne den oben gezeigten komplizierten Ausdruck sofort an einen Nachfolgerknoten gelangen kann, definiert *TGraphicElement* außerdem das Array-Property *ChildNodes*, zu finden im folgenden Listing:

```
TGraphicElement = class(TPersistent)
 { Auszug der Elemente, die für die Baumstruktur
   hinzugefügt wurden }
private
  Edges: TList; { Liste der TEdges, die hiermit verknüpft sind }
  PreviousNode: TGraphicElement;
                { ansprechbar unter dem Property PrevNode }
  Visited: Boolean; { Endlosschleifen-Blocker }
  FCollapsed: Boolean; { True, wenn Verbindung zu
           Nachfolgerknoten nicht gezeichnet wird }
(* Methoden für Properties *)
  procedure SetPreviousNode(N: TGraphicElement);
  function GetChildNode(Index: Integer): TGraphicElement;
public
```

```
  procedure AddNode(n: TEdge);
  property PrevNode: TGraphicElement read PreviousNode
              write SetPreviousNode; { Vorgängerknoten }
  property Collapsed: Boolean read FCollapsed write FCollapsed;
  { Zahl der Nachfolgerknoten }
  property OutCount: Integer read GetChildCount;
  { Array der Nachfolgerknoten }
  property ChildNodes[Index: Integer]: TGraphicElement
                                       read GetChildNode;
(* Speichern und Laden der Verbindungsdaten *)
  procedure SaveEdgesToStream(Stream: TStream);
  procedure LoadEdgesFromStream(Stream: TStream; Items: TList);
(* Funktionen zum Anordnen *)
  function ArrangeTree(var t: TTreeFormat;
     RangeStart, RangeEnd, Depth: Integer): Boolean;
   {  TTreeFormat ist ein Record, der das Format des Baums angibt:
      horizontal/vertikal, Ebenenabstand, Ebenentiefe und
      Knotenbreite }
  procedure LocalExpand;
  procedure MoveTree(XTranslate, YTranslate: Integer);
  { Anzahl der Blätter dieses Teilbaums zählen: }
  function CountLeafs: Integer;
(* Verbindungs-Abbau *)
  procedure DisconnectInEdge;
  procedure DeleteInEdges;
  procedure DeleteOutEdges;
  procedure DeleteAllEdges;
  procedure DisconnectFrom(Item: TGraphicElement);
end;
```

Verbindungsaufbau

Alle Variablen, die Daten der Verbindungen speichern (*TEdge.Start/EndNode*, *TGraphicElement.PreviousNode* und *TGraphicElement.Edges*), sind als privat deklariert, so dass Sie eine neue Verbindung nur über den *TEdge*-Konstruktor *StandardConnection* aufbauen können, wobei Sie diesem die zu verbindenden Objekte als Parameter übergeben:

```
{Connection := }TEdge.StandardConnection(Vorgaenger, Nachfolger);
```

Das neu erstellte *TEdge*-Objekt *Connection* ist hier auskommentiert, weil es nicht extra gespeichert zu werden braucht: Es trägt sich bereits selbst in der *TEdge*-Liste des Vorgängerknotens ein. Falls *Nachfolger* bereits einen Vorgänger hat, kommt es in der obigen Anweisung zu einer *ETreeTypeError*-Exception und das *TEdge*-Objekt wird automatisch wieder freigegeben.

An diesem Verbindungsaufbau sind die folgenden beiden Methoden beteiligt:

```
procedure TGraphicElement.SetPreviousNode(N: TGraphicElement);
begin
  if Assigned(PreviousNode) then
```

```
    raise ETreeTypeError.Create('Eingangsknoten schon vorhanden');
  PreviousNode := N;
end;

constructor TEdge.StandardConnection(n1, n2: TGraphicElement);
begin
  { folgende Zuweisung kann eine Exception auslösen:
    Falls das der Fall ist, wird der Rest des Konstruktors
    übersprungen inklusive des Aufrufs StartNode.AddNode.
    Es ist kein try..finally-Block notwendig. Das Objekt self
    wird im Fehlerfall automatisch wieder freigegeben. }
  n2.PrevNode := n1; { SetPreviousNode wird aufgerufen }

  StartNode := n1;
  EndNode := n2;
  StartNode.AddNode(self);
end;
```

Anordnen

Als Beispiel für die Handhabung der Baumstruktur soll hier die Methode *LocalExpand* dienen, die alle Nachfolgerknoten des Grafikelements links neben diesem in gleichmäigen Abständen anordnet (Abbildung 6.4). Sie können diese zur Laufzeit über das Popup-Menü der Grafikelemente aufrufen.

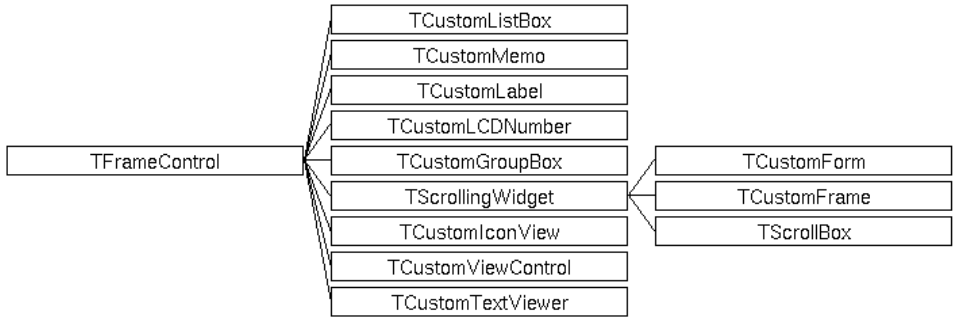


Abbildung 6.4: Das Ergebnis eines zweifachen Aufrufs der Methode LocalExpand

LocalExpand muss neben einigen Berechnungen im Wesentlichen nur alle *Edges.Count* Nachfolgerknoten durchlaufen, die sie über das Property *ChildNodes* einfach ansprechen kann. Jeder Knoten erhält mit *SetNewRect* seine neue Position und Größe.

```
procedure TGraphicElement.LocalExpand;
{ nur für vertikal angeordnete Bäume }
var
  Middle, { vertikale Mitte von Self }
```

```
    RangeStart, RangeEnd, { vertikaler Bereich für die Nachfolger }
    CellHeight, CellStart, CellEnd, { Bereich für einen einzelnen
                 Nachfolger, mit Zwischenraum zwischen den Rechtecken}
    RectHeight, RectWidth, { Daten des Nachfolger-Rechtecks }
    RectDist, { Zwischenraum zwischen Bereichsgrenze und Rechteck }
    i: Integer;
  begin
    FCollapsed := False;
   { vorbereitende Berechnungen }
    Middle := Top+(Bottom-Top) div 2;
    CellHeight := trunc((Bottom-Top)*1.2);
    RangeStart := Middle-Edges.Count*CellHeight div 2;
    RangeEnd := Middle+Edges.Count*CellHeight div 2;
    { eigene Breite und Höhe für den Nachfolgerknoten übernehmen: }
    RectHeight := Bottom-Top;
    RectWidth := Right-Left;
    RectDist := (CellHeight-RectHeight) div 2;
   { Durchlaufen aller Nachfolgerknoten }
    for i := 0 to Edges.Count-1 do begin
      CellStart := RangeStart+i*CellHeight;
      CellEnd := RangeStart+(i+1)*CellHeight;
      { Nachfolger des Nachfolgers verstecken: }
      ChildNodes[i].FCollapsed := True;
      { Nachfolgerknoten anordnen }
      ChildNodes[i].SetNewRect(
        Rect(Right+20, CellStart+RectDist,
             Right+20+RectWidth, CellEnd-RectDist));
    end;
  end;
```

LocalExpand berechnet zu Beginn den vertikalen Platzbedarf der Nachfolgerknoten. Die Knoten werden vertikal innerhalb des Bereichs *RangeStart* bis *RangeEnd* angeordnet, wobei *LocalExpand* für jeden Nachfolgerknoten das 1,2-fache seiner eigenen Höhe berechnet (*CellHeight*). Darin ist bereits der vertikale Zwischenraum zwischen den Nachfolgerknoten enthalten (der halbe Zwischenraum nach oben und nach unten beträgt *RectDist* Einheiten).

Zeiger speichern *R116*

Problematisch ist es, die Baumstruktur auch in einer Datei zu speichern, denn die Daten innerhalb der *TEdge*-Objekte weisen auf *TGraphicElement*-Objekte, die bereits innerhalb der Objektliste von *TGraphicDoc* gespeichert werden. Um eine doppelte Speicherung zu vermeiden, muss statt des Objektinhalts eine Art von Adresse des Objekts gespeichert werden. Da die Speicheradresse eines Objekts jedoch nicht vorhersehbar ist, brauchen wir hier eine sicherere Adresse. Als solche bietet sich der Index des Objekts innerhalb der Liste *TGraphicDoc* an, der sich beim Einlesen nicht ändert, da das Einlesen in derselben Reihenfolge stattfindet und die Elemente mit *Add* immer am Schluss der Liste angehängt werden (Methode *TGraphicDocument.LoadFromFile*).

Wenn nun das Element mit dem Index 10 gespeichert wird und eine Verbindung zum Element an Position 20 vorhanden ist, kann diese Verbindung beim Lesen nicht sofort hergestellt werden, weil Element 20 zum Lesezeitpunkt von Nummer 10 noch gar nicht eingelesen wurde. Daher findet das Lesen in zwei Durchläufen statt: Zuerst werden die Grafikelemente ohne die Verbindungsdaten geschrieben. Dann folgt ein zweiter Durchlauf durch alle Grafikelemente, allerdings werden nun nur die Verbindungsdaten gespeichert:

```
{ Ausschnitt aus TGraphicDoc.WriteToStream }
  for i := 0 to count-1 do
    TGraphicElement(Items[i]).SaveToStream(Stream, i);
  for i := 0 to count-1 do
    TGraphicElement(Items[i]).SaveEdgesToStream(Stream);
```

SaveEdgesToStream bewirkt, dass jede Ecke der *Edges*-Liste sich mit ihrer *SaveToStream*-Methode selbst im Stream speichert. Diese Speicherung sieht so aus, dass statt des Zeigers auf die beiden verbundenen Knoten die beiden Indizes dieser Knoten in den Stream geschrieben werden. Dies ist auch der Grund dafür, dass die Methode *TGraphicElement.SaveToStream* immer einen Index als Parameter erhält, den sie sich dann in der Variablen *FileIndex* merkt.

Zwar stimmt dieser Index beim Speichern einer Datei immer mit dem Index des Grafikelements in der *TList*-Listenstruktur überein, müsste also in diesem Fall gar nicht in einer extra Variablen abgelegt werden. Sollten aber einmal nicht alle Objekte des Dokuments in die Datei geschrieben werden, sondern z.B. nur die markierten Objekte, so weichen Dateiindex und Listenindex voneinander ab, daher wird hier die Variable *FileIndex* benötigt. Im TreeDesigner tritt dieser Fall ein, wenn die markierten Objekte in die Zwischenablage kopiert werden (hierfür werden die markierten Objekte mitsamt ihrer erbindungen in einen MemoryStream geschrieben).

6.4 Mausaktionen und Zeichnen

Dieses Kapitel beschreibt die Durchführung von Mausziehaktionen, bei denen es nicht um Drag&Drop geht, sondern um das Zeichnen eines Objekts oder einer Grafik. Derartige Mausaktionen kommen nicht nur in Grafik- und Malprogrammen, sondern beispielsweise auch in der Kylix-IDE beim Einzeichnen der Komponenten in das Formular vor.

Da das Zeichnen mit der Maus eine etwas komplexere Aktion als das Betätigen eines Schalters ist, müssen Sie drei verschiedene Ereignisse bearbeiten, um den gesamten organg kontrollieren zu können.

Die Ereignisse des Mauszieh-Vorgangs *R87*

In einem Zeichenprogramm, bei dem ein neues Objekt erst dann erstellt wird, wenn der Benutzer die Maustaste loslässt, können Sie die drei Ereignisse wie folgt bearbeiten:

- Wenn Sie beim Ereignis *OnMouseDown* die Position der Maus speichern, können Sie diese beim Abschluss der Aktion als Startpunkt für ein neu erstelltes Grafikobjekt verwenden.
- Während der Mausbewegung erhält der Benutzer normalerweise eine visuelle Rückmeldung über die Größe des eingezeichneten Objekts. Hierfür können Sie bei jedem *OnMouseMove*-Event einen invertierten Rahmen, ein invertiertes Rechteck oder einen anderen grafischen Hinweis im Formular ausgeben.
- Wenn schließlich das *OnMouseUp*-Ereignis anzeigt, dass die Maustaste wieder in ihren Ausgangszustand zurückgekehrt ist, können Sie die provisorische Größenanzeige, die bei den *OnMouseMove*-Events ausgegeben wurde, wieder entfernen und durch die endgültige Grafik ersetzen.

Etwas anders verhält es sich beim Freihandzeichnen in einem Malprogramm, denn dort wird normalerweise bei jeder Mausbewegung eine neue endgültige Linie gezeichnet.

MouseCapture

Normalerweise tritt das *OnMouseMove*-Ereignis nur bei dem Fenster auf, über dem sich die Maus bewegt hat. Da ein Zeichenvorgang jedoch nicht dadurch unterbrochen werden soll, dass der Benutzer die Maus über die Fenstergrenze hinwegbewegt, wird die Maus während des Zeichenvorgangs üblicherweise für das Fenster, in dem die Zeichnung stattfindet, reserviert.

Die CLX erledigt diese Reservierung automatisch bei jedem *OnMouseDown*-Event und hebt sie beim *OnMouseUp*-Event wieder auf. Ob die Mausnachrichten gerade reserviert sind, können Sie über das Property *MouseCapture* des Formulars in Erfahrung bringen.

6.4.1 Die Methoden im TreeDesigner

Nachdem in Kapitel 6.3 die TreeDesigner-Unit *GrDoc* im Mittelpunkt des Interesses stand, wenden wir uns jetzt der Unit *DocForm* zu. Die folgenden Variablen der Formularklasse *TDocumentForm* sind am Ziehvorgang beteiligt:

- *DragStartX* und *DragStartY* speichern die Position, bei der der Mauszieh-Vorgang begonnen hat.
- *LastX* und *LastY* speichern die letzte Position der Maus im Laufe der Bewegung.

- *MouseObject* enthält das Grafikobjekt, das gerade erzeugt wird (bzw. später das Objekt, das von der Maus angeklickt wurde).
- *DragAction* gibt an, ob eine für den TreeDesigner interessante Ziehoperation aktiv ist.
- *DragMode* enthält eine Konstante, die die Art des Ziehvorgangs bezeichnet: *dmCreate* (Erzeugen eines neuen Elements), *dmMark* (Aufziehen eines Rahmens zur Markierung von Elementen), *dmMove* (Verschieben eines einzelnen Elements), *dmMoveTree* (Verschieben eines Elements mit allen als Kind verbundenen Elementen) oder *dmResize* (Änderung der Größe eines Elements).
- *ResizeMode* gibt im Fall von *DragMode=dmResize* an, welcher Punkt des Grafikelement-Rahmens von der Maus zur Änderung der Größe gegriffen wurde.

Da im TreeDesigner nicht direkt auf dem Formular, sondern auf einer *TPaintBox*-Komponente namens *PaintBox* gezeichnet wird, werden nur die drei Mausereignisse dieser Komponente bearbeitet.

aintBoxMouseDown

Die insgesamt sehr umfangreichen Methoden von *TDocumentForm* sind an dieser Stelle auf die Teile beschränkt, die für das Einzeichnen neuer Objekte verantwortlich sind. Das hier beschriebene Einzeichnen eines neuen Objekts wird nur dann ausgeführt, wenn in der Werkzeugleiste eine der grafischen Formen gewählt ist bzw. wenn der Zeigerschalter nicht gedrückt ist (*SpeedBtnArrow.Down=False*). Das Gerüst der Methode ür das *OnMouseDown*-Ereignis sieht wie folgt aus:

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  // Die CLX   // Einstellung des virtuellen Koordinatensystems und Umrechnen
  // der Mausposition, siehe Kapitel 6.6.2
  SetMapMode(PaintBox.Canvas, PaintBox);
  MapMousePos(x, y);
  // Speichern der Mausposition für OnMouseMove/OnMouseUp
  DragStartX := x;
  DragStartY := y;
  LastX := x;
  LastY := y;
  if Button = mbRight then begin
    { Code für den Klick mit der rechten Maustaste, siehe CD }
  end else
    DragAction := True;

  if SpeedBtnArrow.Down then begin
    { Code für den Markierungsmodus, folgt später }
  end else if DragAction begin
```

```
      { Hier startet das Neuzeichnen eines Grafikelements }
      DragMode := dmCreate;
      Document.DeselectAll;
      ShapeText.Text := ''; { Platz machen für neue Beschriftung }
      MouseObject :=
        TGraphicElement.Create(Document, Rect(x, y, x, y), CurShapeType);
      SetAttributes(MouseObject);
      MouseObject.Text := ShapeText.Text;
      MouseObject.Marked := True;
    end;
    if DragAction then begin
      MouseObject.XorPaint(Paintbox.Canvas,
                      Rect(DragStartX, DragStartY, X, Y));
    end;
  end;
```

Die Methode hält sich an den oben beschriebenen Ablauf, speichert also die Mausposition in den Variablen *DragStartX* und *DragStartY*. Sie erstellt bereits ein neues Grafikobjekt und stellt dessen Attribute ein, wie in Kapitel 6.6.2 beschrieben.

Zeichnen beim Ziehen der Maus *R93*

Die Methode *XorPaint* der Grafikelemente zeichnet eine besondere Version des Objekts in der Art, dass sich zwei gleiche Aufrufe dieser Methode auf der Zeichenfläche gegenseitig aufheben. Damit sich zwei gleiche Zeichenaktionen gegenseitig aufheben, müssen Sie lediglich das Property *Canvas.Pen.Mode* auf *pmNotXor* einstellen. Dies wirkt sich dann auch auf die mit dem *Canvas.Brush* gezeichneten Bereiche aus (so z.B. die Füllung eines Rechtecks).

XorPaint setzt also die Zeichenattribute des *Canvas*-Objekts auf die Attribute des Grafikelements und passt *Pen.Mode* an. Dann ruft es die Prozedur *PaintShape* auf, die die grafische Form (ohne Text) ausgibt. Wie die anderen Zeichenmethoden von *TGraphicElement* benötigt auch *XorPaint* einen Parameter *CanvasRef*, in dem die Zeichenfläche angegeben wird, auf der das Element ausgegeben werden soll:

```
procedure TGraphicElement.XorPaint(CanvasRef : TCanvas; XorRect : TRect);
var
  OldPenMode: TPenMode;
begin
  { Pen.Mode muss am Schluss wiederhergestellt werden,
    damit es bei anderen Zeichenvorgängen nicht zu Störungen kommt. }
  OldPenMode := CanvasRef.Pen.Mode
  { Vorbereitungen, damit sich zwei Zeichnungen gegenseitig aufheben: }
  CanvasRef.Brush := Brush;
  CanvasRef.Pen := Pen;
  CanvasRef.Pen.Mode := pmNotXor;
  { Ausgabe der Grafik, siehe Kapitel 6.5.2: }
  PaintShape(CanvasRef, ShapeType, XorRect, Document.DrawEnhanced);
```

```
  { Canvas-Attribute wieder zurücksetzen: }
  CanvasRef.Pen.Mode := OldPenMode;
end;
```

Hinweis: Um die Füllung der Objekte während des Ziehvorgangs abzuschalten, können Sie *Brush.Style* auch auf *bsClear* setzen.

Bewegung der Maus

PaintBoxMouseMove sorgt bei jeder Mausbewegung für die schon beschriebene Aktualisierung der von *XorPaint* gezeichneten Grafik und speichert die neue Mausposition in *LastX/LastY*:

```
procedure TDocumentForm.PaintBoxMouseMove
  (Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin { vereinfachte Version }
  (... Einstellung des virtuellen Koordinatensystems weggelassen ...)
  MapMousePos(x, y);
  // Nebenbei: Anzeige der aktuellen Koordinaten in der Statuszeile:
  Paintbox.Hint := Format('%d : %d', [x, y]);
  if DragAction then begin
    { Aufheben des alten Rechtecks }
    MouseObject.XorPaint(Paintbox.Canvas, Rect
                  (DragStartX, DragStartY, LastX, LastY));
    { Zeichnen des neuen Rechtecks }
    MouseObject.XorPaint(Paintbox.Canvas, Rect
                  (DragStartX, DragStartY, X, Y));
    end;
    LastX := X;
    LastY := Y;
  end;
end;
```

Hinweis: In der vollständigen Version ist *PaintBoxMouseMove* auch dafür zuständig, die Mauszeigerform anzupassen, wenn die Maus sich über einem Größenänderungsgriff eines Objektrahmens befindet; ferner sorgt sie beim Verschieben mehrerer gleichzeitig markierter Objekte für eine Vorschau all dieser Objekte (*XorPaint* muss dabei für jedes verschobene Objekt aufgerufen werden). Eine vollständige Vorschau beim Verschieben von Teilbäumen ist allerdings noch nicht implementiert.

ertigstellung des Objekts

Schließlich kann *PaintBoxMouseUp* dem Grafikobjekt mit *SetNewRect* seine endgültige Größe zuweisen. *SetNewRect* sorgt auf dem Weg der Ungültig-Erklärung (siehe Kapitel 6.5.3) dafür, dass das Objekt sich in seiner endgültigen Gestalt ausgibt.

```
procedure TDocumentForm.PaintBoxMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin { vereinfachte Version }
  (... Einstellung des virtuellen Koordinatensystems weggelassen ...)
  if DragAction then begin
    MapMousePos(x, y); { nur für virtuelle Koordinatensysteme }
    { letzte XOR-Zeichnung aufgeben: }
    MouseObject.XorPaint(PaintBox.Canvas, Rect
                  (DragStartX, DragStartY, LastX, LastY));
    MouseObject.SetNewRect(Rect(DragStartX, DragStartY, X, Y));
    if DragMode = dmCreate then
      Document.Add(MouseObject);
    DragAction := False;
  end;
end;
```

6.4.2 Shift und Variationen eines Ziehvorgangs

Die im letzten Abschnitt gezeigten Methoden werden jetzt noch etwas komplizierter, denn auch die Mausaktionen zum Objekt-Verschieben und zum Ändern der Größe hängen mit den drei schon bearbeiteten Ereignissen zusammen.

Außerdem kommt ein weiterer *SpeedButton* ins Spiel, und zwar der mit dem Zeigersymbol versehene (Name der Komponente: *SpeedBtnArrow*). Wenn dieser selektiert ist, wird die Maus zum Manipulationswerkzeug bestehender Objekte, während sie bei allen anderen Schaltern als Zeichengerät funktioniert (ähnlich wie die Schalter in der Komponentenpalette der Kylix-IDE mit dem Unterschied, dass Kylix normalerweise nach jedem neu gezeichneten Objekt wieder zurück zum Zeigersymbol schaltet).

Die folgende Tabelle zeigt, wie Sie die Mausaktion im Manipulationsmodus mit zusätzlichen Tasten der Tastatur beeinflussen können. (Der TreeDesigner beachtet nur, welche Tasten beim Drücken der Maustaste gedrückt sind. Wenn Sie während des Ziehvorgangs weitere Tasten drücken oder loslassen, so hat das keinen Einfluss mehr.)

Tasten	Bedeutung	Ziehvorgang
keine	Wenn *MouseDown* innerhalb eines Objekts stattfindet: Einzelnes Objekt markieren und mitsamt den verbundenen Nachfolger-Objekten verschieben. Wenn *MouseDown* auf der Randmarkierung eines Objekts stattfindet: Größe des Objekts ändern. Wenn *MouseDown* außerhalb aller Objekte stattfindet: Einen Bereich zum Markieren mehrerer Objekte aufziehen.	ja
⇧	Mehrere Objekte markieren und angeklicktes Objekt verschieben, ohne die verbundenen Nachfolger-Objekte mitzubewegen.	ja
Strg	das angeklickte Objekt mit allen bisher markierten Objekten verbinden.	nein

Da die Taste Alt für Systemfunktionen vorgesehen ist (ihre Betätigung lässt den Tastaturfokus zum System- oder Hauptmenü wandern), kann sie sich übrigens nicht an diesen Variationen beteiligen, aber ⇧ und Strg sind ja für die Zwecke des TreeDesigners bereits ausreichend.

Shift und TShiftState

Bei der Umsetzung der obigen Pläne in lauffähigen Quelltext hilft der Parameter *Shift*, der bei jedem der drei Mausereignisse an die Bearbeitungsmethode gesendet wird. Er gibt den Status der Tasten ⇧, Strg und Alt in Form einer Menge des Typs *TShiftState* an. Um zu überprüfen, ob die Tasten ⇧ und Strg beim Drücken der Maustaste ebenfalls gedrückt waren, testen die Ereignismethoden, ob die Konstanten *ssShift* bzw. *ssCtrl* in dieser Menge enthalten sind (neben diesen Werten und *ssAlt* kann *Shift* auch noch die Flags *ssLeft*, *ssRight* und *ssMiddle* enthalten, die besagen, welche der Maustasten gedrückt sind).

Hinweis: In der ersten CLX-Version werden *ssLeft*, *ssRight* und *ssMiddle* nicht immer korrekt gesetzt. Abhilfe schaffen zwei zusätzliche Zeilen wie am Anfang der oben gezeigten Methode *TDocumentForm.PaintBoxMouseDown*.

Angeklicktes Objekt herausfinden *R88*

Im Manipulationsmodus ist es besonders wichtig, herauszufinden, welchem Grafikobjekt der Mausklick gegolten hat, denn dieses Grafikobjekt wird in den meisten Fällen markiert und in der folgenden Aktion womöglich noch verschoben oder anderweitig manipuliert. *TGraphicDoc* stellt dazu die Methode *GetObjectByPoint* zur Verfügung, die herausfindet, welches Objekt sich an einer gegebenen Position befindet:

```
function TGraphicDoc.GetObjectByPoint(P: TPoint): TGraphicElement;
var
  i: Integer;
begin
  Result := nil;
  { von oben nach unten durchsuchen }
  for i := Count-1 downto 0 do
    if Items[i].PointInShape(P) then begin
      Result := Items[i];
      exit;
    end;
end;
```

Da die Objekte am Anfang der Liste eventuell durch nachfolgende Objekte verdeckt erden, durchsucht *GetObjectByPoint* die Liste der Objekte von hinten nach vorne. Sie findet so das Objekt, das von allen an der Position *P* befindlichen Objekten zu oberst liegt.

Um herauszufinden, ob ein Objekt den Punkt *P* enthält, bedient sich *GetObjectByPoint* einer weiteren *TGraphicElement*-Methode: *PointInShape*. Diese wiederum verwendet die in der Unit *Types* definierte Funktion *PtInRect*, um die gestellte Frage zu beantworten:

```
function TGraphicElement.PointInShape(TestPoint : TPoint) : Boolean;
{ Stellt fest, ob TestPoint im Grafikelement liegt, wird
  verwendet, um das passende Objekt zu einem Mausklick zu finden. }
var
  TestRect : TRect;
begin
  GetNormalizedRect(TestRect); { Reihenfolge von Left/Right
                                 und Top/Bottom überprüfen }
  Result := PtInRect(TestRect, TestPoint);
end;
```

Markierungspunkte zur Größenänderung *R89*

Aus dem Listing der Klassendeklarationen von *TGraphicElement* und *TGraphicDoc* ging bereits hervor, dass es von *GetObjectByPoint* noch eine zweite Version gibt:

```
function GetObjectByPoint(P: TPoint; GripSize: Integer;
                     var rm: TResizeMode) : TGraphicElement; overload;
```

Diese Variante gibt im dritten Parameter zurück, welcher der Markierungspunkte zur Größenänderung angeklickt wurde (oder *rmNone*, falls kein Punkt angeklickt wurde), und benötigt dafür als Eingabe im Parameter *GripSize* die Größe eines Markierungspunkts. Zur Erinnerung noch einmal die Deklaration von *TResizeMode*:

```
type TResizeMode = (rmNone, rmRB, rmLB, rmRT, rmLT, rmR, rmL, rmT, rmB);
                                    // RLBT = Right/Left/Bottom/Top
```

Die mit diesen Markierungspunkten zusammenhängenden Programmteile können hier aus Platzgründen nicht abgedruckt werden, die folgenden Zeilen sollen jedoch eine Idee von der groben Funktionsweise vermitteln. Sie stammen aus einer zweiten Version von *PointInShape*, wo sie direkt auf die beiden Anweisungen folgen, die oben schon für die einfache Version von *PointInShape* gezeigt wurden:

```
if Marked then begin // Markierungspunkte nur sichtbar, wenn markiert
  for GripIterator := TResizeMode(1) to high(TResizeMode) do begin
    if PtInRect(GetGripRect(GripIterator, GripSize), TestPoint) then begin
      rm := GripIterator;
      Result := true;
      exit;
    end;
  end;
end;
```

Wichtigstes Element ist die Hilfsfunktion *GetGripRect*, die für einen übergebenen *TResizeMode*-Wert und eine Größenangabe für die Markierungspunkte die Koordinaten für

das entsprechende Markierungsrechteck berechnet. Diese Koordinaten werden beim Zeichnen der Markierung (siehe CD) oder in diesem Fall für das Überprüfen der Mausposition verwendet.

Der TreeDesigner-Quelltext

Da die vollständigen Methoden etwas umfangreich sind, soll hier ein Auszug aus der *OnMouseDown*-Bearbeitung genügen. Das angeklickte Objekt wird dabei in *MouseObject* gespeichert:

```
{ Aus TDocumentForm.PaintBoxMouseDown: }
{ Mausposition in logische Koordinaten umwandeln }
MapMousePos(x, y);
{ Objekt an logischer Klick-Position feststellen }
MouseObject := Document.GetObjectByPoint(Point(X, Y), GetGripSize,
                                         ResizeHandle);
...
if SpeedBtnArrow.Down then begin
  if MouseObject = nil then begin
    { Es wurde auf die leere Zeichenfläche geklickt. }
    if not (ssShift in Shift) then
      Document.DeselectAll; { alle Objekte deselektieren }
    DragMode := dmMark; { Markierung mehrerer Objekte durch
                          Aufziehen eines Rechtecks. }
  end
  else if Shift = [ssLeft] then begin
    { ein Objekt (MouseObject) wurde mit links angeklickt }
    if ResizeHandle <> rmNone then begin
      { ein Markierungspunkt zur Größenänderung wurde "getroffen" }
      Document.DeselectAll;
      DragMode := dmResize;
      ResizeMode := ResizeHandle;
    end else begin
      { das Innere eines Grafikelements wurde getroffen }
      if not MouseObject.Marked then begin
        { Objekt ist nicht markiert -> als Unterbaum verschieben }
        Document.DeselectAll;
        MouseObject.Marked := True;
        DragMode := dmMoveTree;
        LoadAttributes(MouseObject);
        Document.Exchange(Document.Count-1,
                          Document.IndexOf(MouseObject));
      end else
        { Anklicken eines bereits markierten Objekts -> Alle markierten
          Objekte verschieben }
        DragMode := dmMove;
    end;
  end else if Shift = [ssLeft, ssShift] then begin
    { Shift gedrückt -> kumulativ markieren und evtl. verschieben }
    MouseObject.Marked := not MouseObject.Marked;
```

```
      DragMode := dmMove;
      { weitere Anweisungen, die sicherstellen, dass bei einem markierten
        Objeke 8 Markierungspunkte zur Größenänderung, bei mehreren
        markierten Objekten jeweils nur 4 Punkte gezeichnet werden (wie
        im Formulardesigner der Kylix-IDE) siehe CD }
    end else if Shift = [ssLeft, ssCtrl] then begin
... { Strg-Taste -> Verbindung definieren }
      ...
    end else begin
   { SpeedBtnArrow.Down = False -> der TreeDesigner befindet sich im
     Zeichenmodus. Das Neuzeichnen eines Objekts wurde bereits
     in Kapitel 6.5.2 gezeigt... }
    ...
```

6.5 Grafikausgabe und Scrolling

Dieses Kapitel setzt die in Kapitel 4.4 beschriebenen Grundlagen der Grafikausgabe und der Klasse *TCanvas* voraus und beschreibt die Realisierung einer zunächst nur scrollbaren Grafik am Beispiel des TreeDesigners. Im folgenden Kapitel 6.6 wird die scrollbare Zeichenfläche durch ein virtuelles Koordinatensystem erweitert, mit dessen Hilfe sich die Grafik vergrößern und verkleinern lässt.

6.5.1 Ereignisgesteuerte Grafikausgabe

Unter X Window und anderen GUI-Systemen findet auch die Ausgabe von Grafik ereignisgesteuert statt, wobei die Grafik normalerweise nicht sofort ausgegeben wird, sobald eine Änderung eingetreten ist, sondern erst dann, wenn ein Paint-Ereignis die Anwendung explizit zum Zeichnen der Grafik auffordert. Die folgenden Situationen bzw. Ereignisse machen die Neuausgabe der Grafik erforderlich:

- Mausereignisse, die auftreten, wenn Sie die Grafik direkt mit der Maus verändern, im TreeDesigner also entweder neue Objekte einzeichnen oder bestehende manipulieren.
- Befehlsereignisse, die die Anwendung dazu veranlassen, die Grafik zu verändern. So bieten z. B. Bildverarbeitungsprogramme verschiedene Filter und Effekte an, und Textverarbeitungsprogramme stellen automatische Formatierfunktionen zur Verfügung. Die vergleichbaren Funktionen des TreeDesigners sind die Funktionen zum Anordnen der verbundenen Grafikelemente in einer gleichmäßigen Baumdarstellung.
- Zeichenereignisse, die dann auftreten, wenn Teile der Grafik sichtbar werden, die bisher von einem anderen Fenster verdeckt waren oder am Rand des übergeordneten Fensters bzw. des Bildschirms abgeschnitten wurden. Da X Window diese verdeckten Teile nicht speichert, muss die Anwendung die Grafik neu ausgeben, auch wenn diese sich nicht geändert hat.

Es genügt daher nicht, die Grafik nur bei den zuerst genannten Maus- und Befehlsereignissen anzuzeigen; der Aufbau der Grafik muss jederzeit rekonstruierbar sein. Bei dokumentbasierten Anwendungen beispielsweise wird die Grafik aus den Daten des Dokuments und gegebenenfalls aus weiteren Daten rekonstruiert. Zu diesen weiteren Daten gehören in einem Editor beispielsweise die Anfangs- und Endmarken der Textmarkierung, die sich in der grafischen Ausgabe in einer Hervorhebung der entsprechenden Zeilen und Zeichen auswirken.

Diese Aufzählung erhebt zwar keinen Anspruch auf allgemeine Vollständigkeit, deckt aber zumindest die im TreeDesigner vorkommenden Anlässe zur Grafikausgabe ab.

Die Ungültigerklärung *R99*

Sie können die Grafik zwar entsprechend der obigen Aufzählung in den Methoden für die Maus- und Befehlsereignisse (*OnClick*, *OnMouseDown*, etc.) direkt ausgeben, meistens ist es jedoch wesentlich effektiver, die Grafikausgabe nur in einer einzigen, mit dem Ereignis *OnPaint* verknüpften Methode zu implementieren. Bei Maus- und Befehlsereignissen, die die Grafik verändern, genügt es dann, indirekt ein solches *OnPaint*-Ereignis hervorzurufen, indem Sie den veränderten Bereich der Grafik für ungültig erklären. Hierzu gibt es zwei Möglichkeiten:

- Die Methode *TWidgetControl.Invalidate* erklärt den gesamten Bereich des Formulars bzw. der Komponente für erneuerungswürdig, was zur vollständigen Neuzeichnung führt und in vielen Fällen ein unvertretbar hoher Aufwand ist.
- Die Methode TWidgetControl.InvalidateRect erklärt nur einen Teil des Fensters für ungültig. InvalidateRect erwartet als Parameter das Handle des Canvas-Objekts und eine TRect-Struktur, die den ungültigen Bereich angibt.

Nach dem Aufruf von *Invalidate* oder *InvalidateRect* prüft das GUI-System, ob der ungültige Bereich überhaupt sichtbar ist. Wenn das der Fall ist, sendet X Window ein *Expose*-Event an die Kylix-Anwendung, in der es von der Qt-Bibliothek angenommen und als *QPaintEvent* an die CLX weitergegeben wird. Diese schließlich wandelt das *QPaintEvent* in ein *OnPaint*-Ereignis um. Kapitel 6.5.3 beschreibt den Aufruf von *InvalidateRect* im TreeDesigner.

Grafikausgabe außerhalb der *OnPaint*-Bearbeitung ist selten notwendig. Im TreeDesigner kommt sie nur während der Mausziehoperationen aus Kapitel 6.4 vor.

6.5.2 OnPaint und das Zeichnen von Objekten

Im TreeDesigner sind das Dokumentfenster, das *TGraphicDoc*-Objekt und die einzelnen Grafikobjekte an der Grafikausgabe beteiligt. Durch diese Aufteilung wird die Aufgabe für alle Beteiligten leicht überschaubar. Da die Grafikausgabe im TreeDesigner

nicht auf der Fläche des Formulars, sondern in der Paintbox-Komponente stattfindet, bearbeitet die Formularklasse *TDocumentForm* nicht das eigene *OnPaint*-Ereignis, sondern das der Paintbox-Komponente.

Die Grafikausgabe beginnt also mit der Methode *PaintBoxPaint* der Klasse *TDocumentForm*. Diese muss zunächst das Koordinatensystem einstellen (siehe Kapitel 6.6.2) und teilt dann den weiteren Vorgang auf drei Methoden auf: *PaintBackground* (zeichnet den schattierten Umriss und die Fläche des Zeichenblattes), *PaintAll* (zeichnet die Objekte selbst, also den eigentlichen Grafikinhalt, wie er auch gedruckt wird) und *PaintAllMarks* (zeichnet nur die Markierungen von Objekten und soll im Folgenden, wie auch *PaintBackground*, nicht weiter beachtet werden):

```
procedure TDocumentForm.PaintBoxPaint(Sender: TObject);
var
  VCanvas: TVirtualCanvas;
  ClipRectLP: TRect;
begin
  PaintBox.Canvas.Start;
  VCanvas := TVirtualCanvas.Create(PaintBox.Canvas);
  Document.PaintBackground(VCanvas);
  Document.PaintAll(VCanvas, ClipRectLP);
  Document.PaintAllMarks(VCanvas, GetGripSize);
  PaintBox.Canvas.Stop;
  VCanvas.Free;
end;
```

PaintBoxPaint rahmt zunächst die gesamte Grafikausgabe durch die Aufrufe von *TCanvas.Start* und *TCanvas.Stop* ein. Dies ist nicht für die einfache Grafikausgabe mit CLX-Methoden, sondern für die im TreeDesigner notwendigen direkten Aufrufe von Qt-Methoden erforderlich, die sonst unter Umständen nicht die erwünschte Wirkung haben (ein anderes Beispiel dazu mit näherer Erläuterung finden Sie in Kapitel 5.4.3).

Erweiterung von TCanvas

Als Nächstes konstruiert *PaintBoxPaint* ein Objekt der Klasse *TVirtualCanvas*. Diese ist nicht Teil der CLX, sondern in der Unit *VCanvas* auf der CD definiert und dient dazu, die von *TCanvas* bekannten und zusätzlich alle weiteren im TreeDesigner benötigten Grafikoperationen durch einfache Methodenaufrufe bereitzustellen. Gleichzeitig ist sie auch eine Schicht, die diese *TCanvas*-Erweiterungen bibliotheksunabhängig zur Verfügung stellt. Die Plattformunabhängigkeit wäre alleine durch die Qt-Bibliothek schon gewährleistet, jedoch soll der TreeDesigner sich unter Windows auch mit der VCL ohne die Qt kompilieren lassen. Daher verwendet *TVirtualCanvas* unter Delphi Windows-API-Funktionen, unter Kylix Methoden von Qt.

Sie können dieses *TVirtualCanvas*-Objekt im Folgenden einfach als »die Zeichenfläche« ansehen, als Ersatz des vorgegebenen *TCanvas*-Objekts. Alle Methoden von *TCanvas* stehen auch in *TVirtualCanvas* zur Verfügung. Weitere Informationen zu dieser Klasse finden Sie in Kapitel 6.6.1.

PaintBoxPaint muss den drei *Document*-Methoden die Zeichenfläche, also das *VirtualCanvas*-Objekt übergeben, auf dem sich das Dokument zeichnen soll, denn da die Klasse *TGraphicElement* (wie die gesamte Unit *GrDoc*) völlig unabhängig von jeglichen Formular-Units ist, kann sie natürlich nicht wissen, wo sich die Ausgabefläche befindet.

Zeichnen der Grafikelemente

Kommen wir nun wieder zurück zur Zeichenmethode. Da für *PaintBackground* und *PaintAllMarks* bereits auf die CD verwiesen wurde, bleibt noch die Methode *PaintAll*. Sie ruft lediglich die *Paint*-Methoden der einzelnen Grafikelemente auf:

```
procedure TGraphicDoc.PaintAll(Dest: TCanvas);
var
  i: Integer;
begin
  for i := 0 to Count-1 do
    Items[i].Paint(Dest);
end;
```

In *TGraphicElement.Paint* findet eine weitere Untergliederung in einzelne Teile der Grafikausgabe statt:

- Für alle Elemente werden zunächst die Pinsel- und Stiftattribute eingestellt.
- *PaintContent* mit dem zweiten Parameter *psBeforeFill* zeigt einen eventuellen zusätzlichen Inhalt des Grafikelements an. Wie in Kapitel 6.1.3 erwähnt, können Sie über das Popup-Menü des Grafikelements Bilddateien in das Element laden. Die Implementierung dieser Funktion soll hier nicht weiter untersucht werden, wichtigster Aufruf für die Grafikausgabe ist der Aufruf von *StretchDraw*, um die Bilddatei genau in die Umrisse des Elements einzupassen.
- Die Methode *PaintShape* gibt die eigentliche grafische Form aus, bestehend aus dem Umriss, der mit dem Stift, sowie der Füllung, die mit dem Pinsel gezeichnet wird. Dafür werden die in Kapitel 4.4.3 aufgeführten *TCanvas*-Methoden zum Zeichnen von Linien, (abgerundeten) Rechtecken und Ellipsen verwendet, so dass *PaintShape* für dieses Kapitel ebenfalls keine neue Herausforderung darstellen würde.
- *SetTextClipRegion* stellt eine Clipping-Region ein, durch die die Ausgabe des Textes auf den Bereich des Elements begrenzt wird. Dies wird Thema von Kapitel 6.5.5 sein.

- *PaintText* nimmt die eigentliche Textausgabe vor, diese Methode wird weiter unten abgedruckt.
- Mit *ClippingOff* (eine Methode der erweiterten *TVirtualCanvas*-Klasse) wird das vorher eingestellte Clipping wieder abgeschaltet. Es ist wichtig, dass dieser Aufruf in jedem Fall vor dem Ende der *Paint*-Methode stattfindet, weil sonst darauf folgende Zeichenmethoden an dem alten Clipping-Bereich abgeschnitten werden und somit wirkungslos bleiben. *ClippingOff* wird daher durch einen *try…finally*-Abschnitt gesichert.
- Der zweite Aufruf von *PaintContent* dient dem gleichen Zweck wie der erste, allerdings wird hier durch den Parameter *psAfterFill* angezeigt, dass dieser Aufruf nach dem Ausfüllen der Elementfläche durch die »reguläre« Grafik (*PaintShape* und *PaintText*) stattfindet. Grafikausgabe von *PaintContent* an dieser Stelle wird also die vorherigen Ausgaben übermalen. (Zurzeit zeichnet *PaintContent* übrigens nur den Zeitpunkt *psAfterFill*, der erste Aufruf von *PaintContent* mit *psBeforeFill* als Parameter dient lediglich als Ansatzpunkt für spätere Erweiterungen.)
- *PaintEdges* schließlich zeichnet die Verbindungslinien des Elements zu anderen Elementen. Kapitel 6.3.4 gibt eine kurze Einführung in diese Verbindungen, für den Code der Methode *PaintEdges* muss hier aber auf die CD verwiesen werden.

```
procedure TGraphicElement.Paint(CanvasRef : TVirtualCanvas);
begin
  CanvasRef.Brush := Brush;
  CanvasRef.Pen := Pen;

  PaintContent(CanvasRef, psBeforeFill);
  PaintShape(CanvasRef, ShapeType, Points);
  SetTextClipRegion(CanvasRef);
  try
    PaintText(CanvasRef);
  finally
    CanvasRef.ClippingOff;
  end;
  PaintContent(CanvasRef, psAfterFill);
  PaintEdges(CanvasRef);
end;
```

Polygone *R92*

Da das Qt-API für Dreiecke und alle Formen mit mehr als vier Ecken keine speziellen Ausgabefunktionen hat, müssen die spezielleren Formen des TreeDesigners in etwas aufwändiger Weise berechnet werden. *PaintShape* übergibt diese Arbeit den Prozeduren *PaintRhomb* und *PaintHexagon*. Als Beispiel sei hier die Methode *PaintHexagon* gezeigt, die ein Punkte-Array für das Sechseck berechnet und dieses mit der Methode *Polygon* ausgibt:

```
procedure PaintHexagon(CanvasRef: TCanvas; Points: TRect);
var
  dpoints: array[0..5] of TPoint; { Koordinaten des Sechsecks }
begin
{ Die Reihenfolge der Punkte:

       ___
      /0  1\
     5      2
      \4__3/
}
  dpoints[0].x := points.left+(points.right-points.left) div 3;
  dpoints[1].x := points.right-(points.right-points.left) div 3;
  dpoints[2].x := points.right;
  dpoints[3].x := dpoints[1].x;
  dpoints[4].x := dpoints[0].x;
  dpoints[5].x := points.left;

  dpoints[0].y := points.top;
  dpoints[1].y := points.top;
  dpoints[2].y := points.top+(points.bottom-points.top) div 2;
  dpoints[3].y := points.bottom;
  dpoints[4].y := points.bottom;
  dpoints[5].y := dpoints[2].y;
  CanvasRef.Polygon(dpoints);
end;
```

Die weiteren Methoden finden Sie auf der CD. Als Erweiterung wäre hier zu empfehlen, Formen mit beliebig vielen (gleichmäßig verteilten) Ecken zu implementieren, wobei die Eckenzahl beispielsweise in der Werkzeugleiste eingestellt werden könnte.

extausrichtung *R96*

Nun wenden wir uns der Beschriftung der Grafikelement zu, die durch die Methode *PaintText* vorgenommen wird.

```
procedure TGraphicElement.PaintText(CanvasRef: TVirtualCanvas);
var
  R: TRect;
begin
  CanvasRef.SetTextAlign(TA_CENTER or TA_VCENTER);
  CanvasRef.Font := Font;
  CanvasRef.Font.Height := Font.Height;
  GetInnerRect(R);
  CanvasRef.Brush.Style := bsClear;
  CanvasRef.TextRect(R, Text);
end;
```

Die *TCanvas*-Textausgabemethode *TextOut* erwartet als Parameter die Koordinaten der linken oberen Ecke des Textes, was zur zentrierten Beschriftung des Rechtecks noch nicht ideal ist. Dies entspricht der Ausrichtung *taTop*. Über das Property *TCanvas.*

TextAlign können Sie außerdem eine Ausrichtung von *taBottom* einstellen, damit die Unterkante des Textes an der angegebenen Y-Koordinate platziert wird. Auch dies ist für den TreeDesigner nicht besonders günstig, da hier der Text genau in der Mitte platziert werden soll.

Damit hätten wir den ersten Anlass zu einer kleinen Erweiterung von *TCanvas* erreicht. Der Methode *SetTextAlign* können Sie die folgenden Werte übergeben, um die Ausrichtung zu bestimmen:

```
// aus: VCanvas.pas, TreeDesigner-Quelltextverzeichnis
const
  ta_Center = integer(Qt.AlignmentFlags_AlignHCenter);
  ta_VCenter = integer(Qt.AlignmentFlags_AlignVCenter);
  ta_Top = integer(Qt.AlignmentFlags_AlignTop);
  ta_Right = integer(Qt.AlignmentFlags_AlignRight);
  ta_Left = integer(Qt.AlignmentFlags_AlignLeft);
```

Dabei stehen *ta_Left* und *ta_Right* für die horizontale Ausrichtung des Textes, in der Standardeinstellung *ta_Left* wird die linke Seite des Textes an die Textausgabeposition gesetzt, bei *ta_Right* entsprechend die rechte Seite. *Ta_Top* und *ta_Bottom* entsprechen den auch von der CLX angebotenen Ausrichtungsoptionen. Für den TreeDesigner werden die Optionen *ta_Center* und *ta_VCenter* benötigt, die die horizontale und die vertikale Mitte des Textes an die gewünschte Position bringen.

Alle Ausgabeflags werden unter Linux wie oben gezeigt auf die *AlignmentFlags*-Konstanten von Qt zurückgeführt. Wird der Code für Windows kompiliert, kommen andere als die oben gedruckten Zeilen zum Einsatz.

Die folgende Implementierung der Methode *TextRect* wurde ebenfalls um die Windows-Teile sowie um einigen Verwaltungsaufwand gekürzt. Es soll hier nur gezeigt werden, wie der Text nun tatsächlich ausgegeben wird. Und zwar wird dazu die Methode *QPainter::drawText* verwendet, die über die in *Qt.pas* definierte Prozedur *QPainter_drawText* aufgerufen werden muss. Die Variable *FVCLCanvas* speichert in *TVirtualCanvas* das *TCanvas*-Objekt, das bei der Initialisierung in *PaintBoxPaint* übergeben wurde, und in *FTextAlign* die Ausrichtungs-Optionen, die oben mit *SetTextAlign* eingestellt wurden:

```
procedure TVirtualCanvas.TextRect(Rect: TRect; const Text: string);
var
  WideText: WideString;
begin
  WideText := Text;
  QPainter_drawText(FVCLCanvas.Handle, addr(Rect), FTextAlign,
    addr(WideText), length(Text), nil, nil);
end;
```

Hinweis: Die oben aufgelisteten *AlignmentFlags* von Qt können Sie auch an die von *TCanvas* definierte Methode *TextRect* übergeben. Der TreeDesigner verwendet jedoch lieber *TVirtualCanvas.TextRect*, da diese Methode die Unterschiede zur Windows-Schnittstelle kapselt und der Aufruf von *TextRect* somit unabhängig davon erfolgen kann, ob CLX+Linux oder VCL+Windows verwendet wird.

6.5.3 Effiziente Grafikausgabe

Wenn eine Grafikanwendung wie der TreeDesigner bei jeder kleinen Änderung der Grafik, beispielsweise beim Verschieben eines Objekts, das gesamte Fenster mit *Invalidate* für ungültig erklären würde, hätte das zwei Nachteile:

- Solange wenig Grafikobjekte vorhanden sind und das Neuzeichnen sehr schnell geht, würde das ständige Neuzeichnen zu einem Flackern des Bildes führen.
- Falls sehr viele Objekte zu zeichnen sind, würde das komplette Neuzeichnen unter Umständen zu viel Zeit in Anspruch nehmen.

Wie schon erwähnt, sollten Sie der Methode *Invalidate* in einem solchen Fall die Methode *InvalidateRect* vorziehen.

Der TreeDesigner verwendet diese Funktion bei jeder geringfügigen Änderung der Grafik. Bevor wir zum konkreten Aufruf von *InvalidateRect* kommen, sehen Sie im Folgenden den allgemeinen Ablauf des Neuzeichnens.

Falls ein Objekt beispielsweise verschoben wird, müssen sowohl der Bereich, der vorher vom Objekt verdeckt war, als auch der gegenwärtig belegte Bereich neu gezeichnet werden. Dazu erklärt *TGraphicElement* zuerst das alte und dann das neue Rechteck für ungültig. Dieser Vorgang sieht in der Methode *SetNewRect* wie folgt aus:

```
procedure TGraphicElement.SetNewRect(NewRect: TRect);
begin
  InvalidateAllViews;  { alten Bereich für ungültig erklären }
  Points := NewRect;   { Koordinaten neu setzen }
  InvalidateAllViews;  { neuen Bereich für ungültig erklären }
end;
```

InvalidateAllViews ist eine Methode der Klasse *TGraphicElement* und erklärt den durch das Element abgedeckten Fensterbereich in allen Views (Dokumentfenstern), in denen das Element angezeigt wird, für ungültig. Wichtig ist hierbei, dass das Neuzeichnen erst nach Abschluss der Methode *SetNewRect* stattfindet. Würde es bereits im Aufruf von *InvalidateAllViews* stattfinden, würden ja das Element erneut mit seinen alten Koordinaten gezeichnet werden, da die neuen Koordinaten erst nach *InvalidateAllViews* gesetzt werden.

Hinweis: Das Neuzeichnen findet nicht sofort nach der Beendigung der *SetNewRect*-Methode statt, sondern erst, wenn die Bearbeitung des Ereignisses, das letztendlich den Aufruf von *SetNewRect* ausgelöst hat, abgeschlossen ist. Die CLX wird dann schrittweise die nächsten Nachrichten aus der Nachrichten-Warteschlange abrufen und darunter eine Aufforderung zum Neuzeichnen finden, die sie dann letztlich durch ein *OnPaint*-Ereignis an das Formular weitergibt.

Aufrufen von InvalidateRect

Während Sie den Quelltext der Methode *InvalidateAllViews* in Kapitel 6.3.1 bei der Besprechung des Dokument-View-Konzepts finden, geht es jetzt um die Methode *InvalidateRect*. Wenn Sie kein virtuelles Koordinatensystem verwenden, können Sie die Koordinaten des ungültigen Rechtecks direkt an *InvalidateRect* übergeben. Da *InvalidateRect* aber Gerätekoordinaten erwartet, müssen virtuelle Koordinaten, wie sie im Tree-Designer verwendet werden, vor dem Aufruf mit *LPtoDP* in Gerätekoordinaten umgewandelt werden:

```
procedure TGraphicElement.
              InvalidateRect(var R: TRect; View: TGraphicView);
var
  Control: TWidgetControl;
  PaintBox: TPaintBox;
  I: Integer;
  VCanvas: TVirtualCanvas;
begin
  PaintBox := View.PaintBox; { siehe nächster Abschnitt }
  if Assigned(View.SetMapModeProc) then { Koordinatensystem einstellen }
    VCanvas := View.SetMapModeProc(PaintBox.Canvas, PaintBox);
  { die PaintBox besitzt kein Window-Handle, daher wird auch
    das Elternfenster der Paintbox benötigt: }
  Control := PaintBox.Parent as TWidgetControl;
  { virtuelle Koordinaten in Gerätekoordinaten,
    nur bei virtuellem Koordinatensystem notwendig: }
      LPtoDP(PaintBox.Canvas.Handle, R, 2);
  { Rechteck in die richtige Richtung drehen: }
  NormalizeRect(R);
  { Bereich für ungültig erklären: }
  VCanvas.InvalidateRect(Control, R);
end;
```

Wichtig ist an dieser Stelle nur der Aufruf der Methode *InvalidateRect*, der auch mit *TCanvas* genauso erfolgen kann wie oben mit *TVirtualCanvas*. Hierfür wird zunächst eine *TWidgetControl*-Komponente benötigt, in der sich der ungültig zu erklärende Bereich befindet. Die Paintbox selbst ist kein *TWidgetControl*, daher muss die Elternkomponente der Paintbox verwendet werden. Die Paintbox lässt sich aus den *View-*

Daten ermitteln (siehe *TGraphicView* in Kapitel 6.3.1). Der zweite Parameter von *SetWindowRect* ist das ungültig zu erklärende Rechteck, welches in Gerätekoordinaten angegeben wird und daher eventuell zuerst berechnet werden muss.

Rechteck-Orientierung

Ein letzter Punkt, der im TreeDesigner beachtet werden muss, ist folgender: Beim Einzeichnen der Grafikelemente übernimmt der TreeDesigner die Rechteck-Koordinaten, wie der Benutzer sie einzeichnet: Die Position, an der die Maus geklickt wurde, wird zur Ecke (*Left*, *Top*), die Position des Loslassens wird zu (*Right*, *Bottom*). Wenn auf diese Weise ein Rechteck entsteht, bei dem *Left* > *Right* oder *Top* > *Bottom*, dann macht das bei der Ausgabe der Grafik mit *Rectangle* und *Ellipse* nichts aus. Für Linien-Objekte ist es sogar wichtig, dass die Koordinaten auch in diesen Verhältnissen vorliegen können, denn sonst würden alle Linien-Objekte von links oben nach rechts unten gezeichnet[1].

Einige Funktionen wie *InvalidateRect* erwarten jedoch Rechtecke, bei denen sich durch die Berechnung von *Bottom-Top* und *Right-Left* positive Werte ergeben. Andere Rechtecke werden je nach Funktion als nicht vorhanden oder als nach einer Seite unbegrenzt etc. interpretiert. Für diese Funktionen müssen die Rechteck-Koordinaten zuerst in die richtige Reihenfolge gebracht werden, weshalb in der oben gezeigten Methode die Prozedur *NormalizeRect* aufgerufen wird, die wie folgt arbeitet:

```
procedure NormalizeRect(var R: TRect);
  procedure Swap(var x, y: Integer);
  var h: Integer;
  begin
    h := x;  x := y;  y := h;
  end;
begin
  with R do begin
    if right < left then Swap(right,left);
    if bottom < top then Swap(bottom,top);
  end;
end;
```

Unabhängigkeit des Grafikdokuments vom Fenster

In den bisher beschriebenen Funktionen hat *TGraphicElement* die Zeichenfläche des Fensters, auf der es sich ausgeben sollte, immer in einem Parameter namens *CanvasRef* erhalten. Zum Ungültig-Erklären des Rechtecks wird die Zeichenfläche ebenfalls benötigt, da mit ihrer Hilfe die virtuellen in die logischen Koordinaten umgewandelt werden.

---

[1] Das betrifft natürlich nicht die automatisch gezeichneten Verbindungslinien *zwischen* den Grafikobjekten.

Da es sehr umständlich wäre, jeder *TGraphicElement*-Methode, in der das Grafikelement neu gezeichnet werden muss, einen Parameter für die Zeichenfläche zu geben (die Properties *ShapeText* und *Text* könnten so überhaupt nicht mehr geändert werden, da deren Schreibmethoden gar keine zusätzlichen Parameter haben dürfen), schlägt die Unit *GrDoc* einen anderen Weg ein, bei dem die Klassen *TGraphicDoc* und *TGraphicElement* weiterhin unabhängig von jeglichen Formular-Units bleiben.

Und zwar verlangt sie von jedem Fenster, das ein Grafikdokument des Typs *TGraphicDoc* darstellen und verändern will, einen *TGraphicView*-Record, in dem sich unter anderem die oben benötigten Daten *PaintBox* und *SetMapModeProc* befinden. Die Deklaration von *TGraphicView*, die Konstruktion dieses Records und die Weitergabe an das *TGraphicDoc*-Objekt wurden bereits in Kapitel 6.3.1 im Zusammenhang mit dem Dokument-View-Konzept gezeigt.

Das Neuzeichnen in OnPaint

Es stellt sich nun noch die Frage, was passiert, wenn nur ein Teil des Fensters für ungültig erklärt wird. Schließlich haben wir die *PaintboxPaint*-Methode nicht geändert, so dass diese nach wie vor *alle* Objekte zeichnet, selbst wenn nur ein kleiner Bereich der Grafik verändert wurde.

Das Neuzeichnen aller Objekte bei jeder kleinen Änderung ist natürlich für den Benutzer auffällig und selbst wenn es sehr schnell geht, wird noch ein Flimmern des Bildes sichtbar bleiben. Unter Windows gibt es dieses Problem übrigens nicht, denn Windows begrenzt diese Neuausgabe automatisch auf den ungültigen Bereich. Dies wäre auch unter Linux mit Hilfe der Clipping-Funktionen von Qt zumindest weitgehend nachahmbar, wenn Sie die Clipping-Regionen bzw. die Klasse *QRegion* verwenden. Aufgrund des sowieso schon hohen Portierungsaufwandes konnte dies für die aktuelle Version des TreeDesigners aus Zeitgründen noch nicht berücksichtigt werden. Auch die CLX hat noch etwas nachzuholen, was die richtige Behandlung des neu zu zeichnenden Bereichs betrifft, wie der nächste Abschnitt zeigen wird. Wenn Borland diese Funktionen in der CLX nachrüstet, wird die CLX im besten Fall auch die automatische Beschränkung der Neuausgabe übernehmen, so dass sich das Problem des Flackerns von selbst lösen würde.

Vermeidung unnötigen Zeichnens *R97*

Eine Optimierung muss aber auf jeden Fall noch im TreeDesigner berücksichtigt werden: Die Neuausgabe aller Objekte in *PaintAll* ist unter Umständen sehr zeitaufwändig, vor allem, wenn viele Objekte mit mehrzeiligem Text, etwa zur Darstellung einer XML-Datenstruktur, gezeichnet werden sollen. Der Benutzer soll z. B. nach jeder Verschiebung eines Grafikobjekts nicht zwei Sekunden auf das Neuzeichnen warten müssen.

Daher zeichnet der TreeDesigner nur die Objekte, die sich im sichtbaren Bereich befinden. Der sichtbare Bereich wird wie folgt ermittelt: Zunächst gibt *Scrollbox.ClientRect* die Begrenzung der Scrollbox an, innerhalb der die Paintbox sichtbar ist. Die linke obere Koordinate von *ClientRect* ist dabei (0, 0). Das *ClientRect* kann nun mit *DPtoLP* in logische Koordinaten relativ zum virtuellen Koordinatensystem in der Paintbox umgewandelt werden, wobei von *DPtoLP* auch berücksichtigt wird, dass die Paintbox eventuell innerhalb der Scrollbox verschoben wurde. Nach der Umwandlung gibt *ClientRectLP* im folgenden Listing daher den Ausschnitt des virtuellen Koordinatensystems an, der im Fenster sichtbar ist. Dieser Bereich wird nun als zusätzlicher Parameter an *PaintAll* übergeben:

```
procedure TDocumentForm.PaintBoxPaint(Sender: TObject);
var
  ClipRectLP: TRect;
begin
  PaintBox.Canvas.Start;
  ClipRectLP := Scrollbox.ClientRect;
  VCanvas.DPtoLP(ClipRectLP);
  Document.PaintAll(VCanvas, ClipRectLP); // erweitert um 2. Parameter
  ...
```

PaintAll überprüft nun, bevor es ein einzelnes Grafikelement mit *Paint* ausgibt, ob dieses ganz oder teilweise innerhalb *ClipRect* liegt. Hierzu wird die Funktion *IntersectRect*, die in der Borland-Unit *Types* definiert ist, verwendet. Wichtig ist, dass in beiden an *IntersectRect* übergebenen Rechtecken die Koordinaten (*Right, Bottom*) jeweils größer sind als (*Left, Top*), daher wird auch hier, wie schon vor dem Aufruf von *InvalidateRect*, die bereits gezeigte *NormalizeRect*-Prozedur aufgerufen:

```
procedure TGraphicDoc.PaintAll(Dest: TVirtualCanvas; ClipRect: TRect);
var
  i: Integer;
  DummyRect, // Rechteck der Überschneidung, wird nicht benötigt
  PaintRect: TRect;
begin
  for i:=0 to Count-1 do begin
    Items[i].GetPaintRect(PaintRect, 0);
    NormalizeRect(PaintRect);
    NormalizeRect(ClipRect);
    if IntersectRect(DummyRect, PaintRect, ClipRect) then
      Items[i].Paint(Dest);
  end;
end;
```

Hinweis: Diese Maßnahme sollte nicht mit der vorher beschriebenen Idee verwechselt werden, das Neuzeichnen durch Clipping auf den ungültigen Bereich zu beschränken: Bei aktiviertem Clipping können theoretisch alle Grafikobjekte ihre Grafikausgabebefehle an die Zeichenfläche senden. Am Bildschirm sichtbar sind dann jedoch durch das Clipping nur die Zeichenoperationen innerhalb des ungültigen Bereichs, so dass ein Flackern des nicht ungültigen Bereichs verhindert wird. Die oben gezeigte Abfrage in *PaintAll* kann ein Flackern nicht verhindern, wohl aber, dass Objekte ihre Grafikausgabebefehle an die Zeichenfläche senden, die gar nicht sichtbar sind.

Optimal wäre es, nur die Objekte zu zeichnen, die sich im ungültigen Bereich befinden. Tatsächlich besitzt *TCanvas* sogar ein Property namens *UpdateRect*, das die Angabe dieses ungültigen Bereichs verspricht. In *PaintBoxPaint* müsste nur *Scrollbox.ClientRect* durch *Paintbox.Canvas.UpdateRect* ausgetauscht werden, um davon Gebrauch zu machen. Jedoch gibt es zwei Nachteile:

- In der ersten CLX-Version wird *UpdateRect* immer auf den Bereich der gesamten Komponente, hier der gesamten Paintbox gesetzt, hat also letztlich noch gar keinen Effekt. Es ist zu hoffen, dass dies in einer zukünftigen Version der CLX verbessert wird.
- Selbst wenn *UpdateRect* einen korrekten Wert enthielte, wäre es nicht nützlich, die ausgegebenen Objekte auf diesen Bereich zu beschränken, wenn nicht gleichzeitig das Clipping auf diesen Bereich eingeschränkt werden würde. Denn durch das fehlende Clipping wird durch die Methode *PaintBackground* immer der gesamte Hintergrund des Dokumentfensters gezeichnet. Dadurch werden auch alle außerhalb von *UpdateRect* noch sichtbaren Grafikelemente gelöscht, müssen danach also wieder neu gezeichnet werden.

Hinweis: Wenn der zweite Nachteil in Ihrer Anwendung keine Rolle spielen sollte oder Sie selbst für das notwendige Clipping sorgen, können Sie ein korrektes *UpdateRect* an der CLX vorbei auch direkt von Qt erfragen, indem Sie das *QPaintEvent* abfangen und mit *QPaintEvent_rect* das zugehörige Update-Rechteck abfragen. Auf der CD finden Sie dies in der Methode *TScrollboxEx.EventFilter*, die das Rechteck im Property *TScrollboxEx.UpdateRect* verfügbar macht, wo es auch vom TreeDesigner genutzt werden könnte.

6.5.4 Scrolling

Scrolling von Grafiken wird dann erforderlich, wenn die Grafik größer ist als die auf dem Bildschirm zur Verfügung stehende Fläche. Zwar ist es meistens möglich, die Grafik so zu verkleinern, dass sie ganz in das Fenster passt, jedoch ist bei einer solchen Übersichtsansicht oft kein Text mehr lesbar.

Dieses Kapitel stellt allgemeine Methoden des Scrollings sowie die im TreeDesigner gewählte Möglichkeit vor. Ein Ziel beim Scrollen ist in jedem Fall, dass die schon geschriebenen Grafikausgabemethoden nicht geändert werden müssen, und zwar nicht aus Bequemlichkeit, sondern um ihre Unabhängigkeit vom Scrolling zu gewährleisten.

Die CLX stellt einen Komponententyp namens *TScrollBox* zur Verfügung, der das Scrollen vereinfacht, wenn Sie nach dem unten beschriebenen Muster vorgehen.

eeDesigner-Versionen

Dieses Kapitel beschreibt das Verhalten des TreeDesigners ohne die Verwendung des erst in Kapitel 6.6 beschriebenen virtuellen Koordinatensystems. Die fertigen TreeDesigner-Versionen auf der CD arbeiten per Voreinstellung zwar mit eingeschaltetem virtuellen Koordinatensystem, Sie können dieses jedoch mit dem Menüpunkt ANSICHT | ERÄTEKOORDINATENSYSTEM aus- und wieder einschalten. Allerdings dreht sich die Grafik dadurch nahezu komplett um, denn die Y-Koordinaten sind im virtuellen Koordinatensystem so eingestellt, dass sie von unten nach oben ansteigen im Gegensatz zum voreingestellten Fensterkoordinatensystem. Das Einzige, was sich nicht umdreht, ist der zentrierte Beschriftungstext der Grafikobjekte.

ür den Programmcode auf der CD bedeutet das, dass Sie an den Code-Stellen, die beim später eingeführten virtuellen Koordinatensystem anders funktionieren als bei dem in diesem Kapitel vorausgesetzten Gerätekoordinatensystem, eine Fallunterscheidung der folgenden Art finden:

```
if NoMapMode.Checked then
  { Code bei Verwendung des Gerätekoordinatensystems }
else ...
```

Dabei ist *NoMapMode* der Name des Menüpunkts ANSICHT | GERÄTEKOORDINATENSYSTEM. Es kann aber schon verraten werden, dass es nicht viele solcher Stellen gibt; im Prinzip wird in Kapitel 6.6 einfach das virtuelle Koordinatensystem hinzukommen und alles andere funktioniert weiter wie gehabt.

Funktionsweise des Scrollings

Damit die eigentlichen Grafikdaten beim Verschieben der Bildlaufleisten nicht verändert werden müssen, ist es erforderlich, in zwei verschiedenen Koordinatensystemen zu arbeiten: Im Koordinatensystem der Grafik (virtuelles, logisches oder Weltkoordinatensystem) und in dem des Fensters. Wenn die Grafikausgabefunktionen die Weltkoordinaten automatisch in Fensterkoordinaten umrechnen, können Sie diese Funktionen immer mit den gleichen Originalkoordinaten der Grafik aufrufen, unabhängig davon,

welcher Grafikausschnitt gerade durch die Bildlaufleisten eingestellt ist. Verschiebt der Benutzer die Bildlaufleisten, so muss lediglich der Ursprung des Fensterkoordinatensystems relativ zum Weltkoordinatensystem verschoben werden (Abbildung 6.5a).

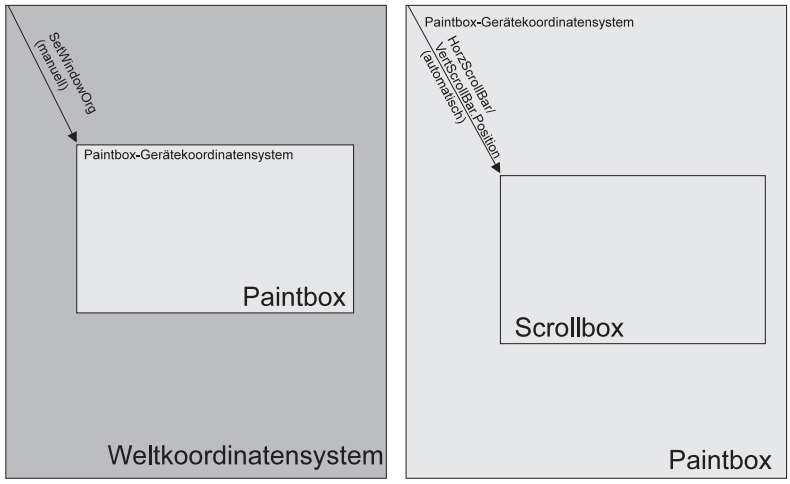


Abbildung 6.5: Der Fensteraufbau für zwei Arten des Scrollens

Praktische Umsetzung

Im Folgenden sind drei verschiedene Realisierungen des Scrollings kurz vorgestellt:

- Die Methode mit den wenigsten Beschränkungen ist die, dass Sie die Weltkoordinaten selbst in Fensterkoordinaten umrechnen. Auf diese Weise sind Sie nicht auf eventuelle Beschränkungen des Grafiksystems angewiesen (16-Bit-Koordinaten, wie sie etwa unter Windows 98 noch anzutreffen sind, sind zwar unter Linux kein Thema, aber es könnte ja auch sein, dass Ihnen 32-Bit-Koordinaten nicht ausreichen). Die PaintBox, in der die Grafik ausgegeben wird, braucht dann nicht größer als der gesamte zur Verfügung stehende Bereich des Formulars zu sein, allerdings könnte sie dann nicht mehr ohne weiteres mit Hilfe einer ScrollBox gescrollt werden.
- Da Qt bereits Unterstützung für logische Koordinaten enthält, könnten Sie diese zur Umrechnung der Weltkoordinaten in Fensterkoordinaten verwenden. Beim Scrollen der Grafik müsste dann nur das so genannte *Window* verschoben werden. Auch bei dieser Vorgehensweise genügt eine Paintbox, die durch *alClient*-Ausrichtung den gesamten Bereich füllt.

- Die von *TScrollBox* begünstigte und damit bequemste Möglichkeit besteht darin, die Paintbox so groß zu machen, dass die gesamte Grafik hineinpasst (also so groß wie das Weltkoordinatensystem). Wenn die Paintbox der Scrollbox untergeordnet ist, aber größer als diese ist, wird sie automatisch am Rand der Scrollbox abgeschnitten, und Sie können mit den Bildlaufleisten der Scrollbox jeden Bereich der Paintbox sichtbar machen. Der grundlegende Unterschied zu den beiden vorher genannten Methoden ist, dass die Weltkoordinaten immer mit den Fensterkoordinaten der Paintbox übereinstimmen (solange keine zusätzlichen Koordinatentransformationen verwendet werden; Abbildung 6.5b).

Da *TScrollBox* die Verschiebung der Paintbox automatisch übernimmt, nehmen wir dieses Angebot dankend an.

Der visuelle Entwurf *R4*

Um zur Entwurfszeit eine solche scrollbare Paintbox zu erhalten, fügen Sie Komponenten der Typen *TScrollBox* und *TPaintBox* in das Formular ein, wobei letztere Komponente Kindfenster der *ScrollBox* ist. Gehen Sie also so vor wie beim Einfügen von Komponenten in Panels, indem Sie die *PaintBox* direkt in die *ScrollBox* einzeichnen.

Damit die ScrollBox genau den Bereich füllt, der nach Abzug der Werkzeugleiste noch brig bleibt, und die Größe automatisch mit der des Formulars verändert, ist sie im reeDesigner mit der Ausrichtung *alClient* versehen.

Die *TPaintBox*-Komponente darf jedoch nicht auf diese Weise ausgerichtet sein, denn ihr Kennzeichen ist es ja gerade, dass sie viel größer ist als die Scrollbox – womit es schwierig wird, sie mit der Maus zu zeichnen. Entweder stellen Sie die Werte *Width* und *Height* also im Objektinspektor ein, oder Sie berechnen Sie zur Laufzeit. Im TreeDesigner werden Breite und Höhe zur Laufzeit bei der Initialisierung des Fensters in der Methode *ResizePaintbox* eingestellt, allerdings wurden die Werte *Left* und *Top* bereits zur Entwurfszeit endgültig auf 0 und 0 festgelegt (diese Position wird an verschiedenen Stellen des TreeDesigners vorausgesetzt, z. B. in *TDocumentForm.SetMapMode* und beim Einstellen der Lineale).

Ereignisse

Welche Ereignisse müssen nun noch bearbeitet werden? Erfreulicherweise keine. *TScrollBox* verschiebt den Zeichenbereich von alleine und die Koordinaten werden in beiden Richtungen automatisch umgewandelt:

- Vom Programm »empfangene« Koordinaten wie die Positionen von Mausklicks in den Methoden für die Mausereignisse werden automatisch relativ zur linken oberen Ecke PaintBox angegeben, die eventuell weit außerhalb des Bildschirms liegt, und

- vom Programm »gesendete« Koordinaten wie die Parameter für die Grafikausgabemethoden beziehen sich ebenfalls auf das PaintBox-System; alle Umrechnungen laufen automatisch ab.

Auch die Neuausgabe der Grafik, wenn sich die Position der Bildlaufleisten ändert, läuft automatisch ab. Dabei wird der Bereich, der sichtbar bleibt, innerhalb des Fensters verschoben und nur der vorher nicht sichtbare Bereich wird neu gezeichnet, so dass es lediglich an den Rändern zu einem Flackern des Bildes kommen kann.

Fließendes Scrolling

Wenn der Benutzer die Bildlaufleisten der Scrollbar mit der Maus verschiebt, kann die Aktualisierung der Grafik auf zwei verschiedene Weisen erfolgen: entweder nur einmal, wenn der Benutzer die Maustaste loslässt, oder fortlaufend, während die Maus bewegt wird. Die letztgenannte Option, das »fließende Scrolling«, erhalten Sie dadurch, dass Sie Properties *HorzScrollBar.Tracking* und *VertScrollBar.Tracking* der ScrollBox einschalten. Im Falle des TreeDesigners ist das aber noch nicht ausreichend, da nicht nur der Inhalt der Zeichenfläche verschoben, sondern auch die Positionsangabe in den Linealen während des Scrollens angepasst werden soll.

Der TreeDesigner verwendet daher die Klasse *TScrollBoxEx* aus Kapitel 8.5.2, die das im nächsten Absatz erläuterte *OnScroll*-Ereignis zur Verfügung stellt. Zum fließenden Scrollen selbst sei noch angemerkt, dass ein angenehmer Ablauf dieses Scrollens natürlich voraussetzt, dass die Grafik schnell genug neu aufgebaut werden kann. Für den TreeDesigner gilt hier, dass das Verschieben der Grafik nicht so zeitkritisch ist wie das fließende Zoomen mit dem Scrollbar der Mauspalette (siehe Kapitel 6.6.2), denn beim Verschieben müssen nur die Teile, die vorher unsichtbar waren, neu gezeichnet werden, während bei einer Änderung des Zoomfaktors die gesamte Zeichenfläche aktualisiert werden muss.

Lineale

Die Klasse *TScrollBoxEx* erzeugt bei jedem Scrollvorgang ein *OnScroll*-Ereignis, das der TreeDesigner dazu nutzen kann, die beiden Lineal-Komponenten anzupassen. Zwar wäre es auch möglich, ohne dieses Ereignis die Lineale auszugeben, und zwar immer dann, wenn auch das Ereignis *OnPaint* auftritt. Dies würde jedoch zu einer Reihe unnötiger Wiederholungen des Lineal-Zeichnens führen (unter anderem jedes Mal dann, wenn die Grafik verändert wird).

Die folgende Version der Methode gilt nur, solange noch keine Veränderung des Koordinatensystems durchgeführt wird (die vollständige Version finden Sie auf der CD). Sie gibt auch ein Beispiel dafür, wie die im Komponenten-Package des Buchs enthaltene *TRuler*-Komponente verwendet wird:

```
procedure TDocumentForm.ScrollBoxScroll(Sender: TObject; Code: Word;
  Horizontal: Boolean);
begin { Einstellen der durch die Lineale gezeichneten Grenzen }
  HRuler.LeftVal := ScrollBox.HorzScrollBar.Position;
  HRuler.RightVal := ScrollBox.HorzScrollBar.Position
                     +ScrollBox.ClientWidth;
  VRuler.LeftVal := ScrollBox.VertScrollBar.Position;
  VRuler.RightVal := ScrollBox.VertScrollBar.Position
                      +ScrollBox.ClientHeight;
end;
```

6.5.5 Clipping

Ein wichtiger Funktionsbereich der Qt-Bibliothek, der nicht von der CLX abgedeckt wird, ist das *Clipping*. Mit dem Clipping können Sie die Grafikausgabe, die bereits standardmäßig auf den Bereich des Fensters beschnitten wird, weiter einschränken (Abbildung 6.6). Wenn Sie beispielsweise einen Kreis als Clipping-Bereich festlegen und die gesamte Zeichenfläche mit dem Aufruf der Methode *Rectangle* zu füllen versuchen, wird stattdessen nur der Kreis ausgefüllt, den Sie als Clipping-Bereich angegeben haben.

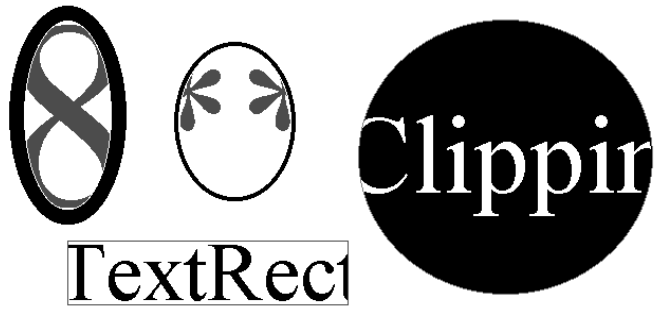


Abbildung 6.6: Text-Clipping im TreeDesigner-Dokument

Erstellen eines einfachen Clipping-Bereichs *R100*

Auf Qt-Ebene ist die Klasse *QPainter* für das Clipping zuständig. Sie besitzt sowohl *setClipRect*-Methoden mit verschiedenen Aufrufformaten, mit denen Sie schnell ein Rechteck für das Clipping definieren können, als auch eine ganz allgemeine Methode *setClipRegion* der Sie ein *QRegion*-Objekt übergeben. *QRegion* kann ellipsen- und rechteckförmige Bereiche sowie alle Bereiche, die sich durch Schnitte, Vereinigungen und Differenzen von solchen einfachen Regionen ergeben, enthalten.

Das folgende Beispiel zeigt die Definition einer einfachen elliptischen Region, wie sie im TreeDesigner benötigt wird, wenn die Grafikform *stEllipse* gewählt wurde. Die Region wird mit *QRegion_create* erzeugt und dann an das *QPainter*-Objekt übergeben. Dieses fertigt sich intern eine Kopie dieser Region an, so dass sie danach gleich wieder mit *QRegion_destroy* freigegeben werden kann.

Diese Aktionen sind in der Klasse *TVirtualCanvas* zu einer einfachen Methode namens *SetClippingEllipse* zusammengefasst, welche ein Rechteck in logischen Koordinaten als Parameter erwartet. Da Qt für das Clipping jedoch Gerätekoordinaten erwartet, ist hier wieder ein Aufruf von *LPtoDP* erforderlich. Sie brauchen diese Umwandlung nicht, wenn Sie keine virtuellen Koordinatensysteme verwenden.

```
procedure TVirtualCanvas.SetClippingEllipse(R: TRect);
var
  Rgn: QRegionH;
begin
  if Assigned(FVCLCanvas) then begin
    LPtoDP(R);
    Rgn := QRegion_create(@R, QRegionRegionType_Ellipse);
    QPainter_setClipRegion(FVCLCanvas.Handle, Rgn);
    QRegion_destroy(Rgn);
  end
  else raise ENoVCLCanvas.Create(NoVCLCanvasError);
end;
```

Entsprechend arbeitet auch die Methode *TVirtualCanvas.SetClippingRectangle*, die nur eine andere *RegionType*-Konstante an *QRegion_create* übergibt. Beide Methoden werden von *TGraphicElement.SetTextClipRegion* aufgerufen, je nach der verwendeten Grafikform (der Einfachheit halber und auch aus Mangel an Alternativen wird bei allen anderen Formen außer Rechteck und Ellipse einfach ein Rechteck verwendet).

```
procedure TGraphicElement.SetTextClipRegion(CanvasRef : TVirtualCanvas);
var
  R: TRect;
begin
  GetInnerRect(R);
  NormalizeRect(R);
  if (ShapeType=stEllipse) then
    CanvasRef.SetClippingEllipse(R)
  else
    CanvasRef.SetClippingRectangle(R);
end;
```

SetTextClipRegion wird als Teil der Methode *TGraphicElement.Paint* aufgerufen, die Sie in Kapitel 6.5.2 abgedruckt finden. Dort wurde auch bereits eine Methode *ClippingOff* aufgerufen, die das Clipping wieder abschaltet. Das folgende Listing zeigt, wie diese Methode durch einen einfachen Aufruf einer Qt-Methode implementiert ist:

```
procedure TVirtualCanvas.ClippingOff;
begin
  if Assigned(FVCLCanvas) then begin
    QPainter_setClipping(FVCLCanvas.Handle, False);
  end
  else raise ENoVCLCanvas.Create(NoVCLCanvasError);
end;
```

6.6 Skalierung, virtuelle Koordinatensysteme und Druckerausgabe

irtuelle Koordinatensysteme erleichtern die Ausgabe von geräteunabhängiger Grafik erheblich und erlauben eine Vergrößerung der Grafik auf eine einfache Weise, die grundsätzlich so funktioniert: Sie haben auf der einen Seite ein virtuelles *Weltkoordinatensystem*, das mehrere tausend Einheiten durchmisst und in dem sich Ihre Grafik befindet, auf der anderen Seite haben Sie ein Fenster des Bildschirms. Nun sagen Sie dem Grafiksystem, dass es im beispielsweise 1000*500 Pixel großen Fensterbereich einen 100*50 virtuelle Einheiten großen Ausschnitt des Weltkoordinatensystems darstellen soll. In diesem Beispiel erhalten Sie eine zehnfache Vergrößerung, denn jede virtuelle Einheit ist auf dem Bildschirm 10 Pixel groß.

Kapitel 6.6.2 zeigt den praktischen Einsatz eines solchen Koordinatensystems am Beispiel des TreeDesigners. Einen noch größeren Nutzen ergeben die virtuellen Koordinatensysteme jedoch auf einem anderen Gebiet: der Geräteunabhängigkeit. Wie Sie diese ür die Druckerausgabe nutzen können, erfahren Sie in Kapitel 6.6.3. Das Kapitel 6.6.1 beginnt zunächst mit den Grundlagen.

6.6.1 Window- und Viewport-Transformation

Außer den in Kapitel 5.4.3 beschriebenen Funktionen zur Koordinatentransformation bietet Qt noch zwei weitere Transformationsstufen, bei denen die Koordinaten an zwei Rechtecken umgerechnet werden, die *Window* und *Viewport* genannt werden. Der Begriff »Window« hat nichts mit den Fenstern der Benutzerschnittstelle zu tun, sondern meint einen Ausschnitt aus dem Weltkoordinatensystem. Der Viewport befindet sich hingegen auf der physikalischen Abbildungsfläche (meistens das Bildschirmfenster oder der Drucker), ist aber nicht mit dieser gleichzusetzen. Die Transformation läuft so ab, dass der durch das *Window* definierte Bereich auf den den Bereich des *Viewports* projiziert wird. So können Sie beispielsweise bei einem festen Viewport verschiedene Abbildungsmaßstäbe der Grafik erhalten, indem Sie die Größe des Windows variieren. Abbildung 6.7 veranschaulicht diesen Vorgang.

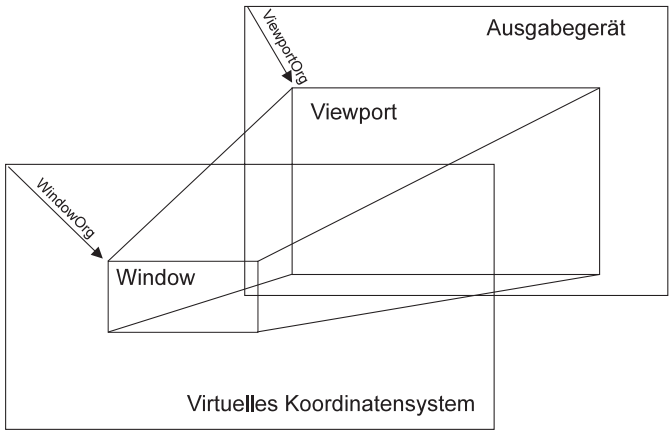


Abbildung 6.7: Window, Viewport und Gerätekoordinatensystem

In Abbildung 6.8 sehen Sie außerdem ein Testprogramm, das Sie unter dem Namen *CoordinateSystemTestForm* auf der CD-ROM finden. Es ermöglicht Ihnen eine interaktive Einstellung der Abbildungsmodi und deren Parameter und eignet sich somit gut zum Experimentieren mit den im Folgenden besprochenen Parametern. (Das Programm verknüpft alle *OnChange*-Events der Eingabeelemente mit einer Ungültig-Erklärung des gesamten Fensters durch *Invalidate*. Bei jedem *OnPaint*-Ereignis stellt es den Abbildungsmodus gemäß den Eingaben neu ein und gibt das Testbild neu aus.)

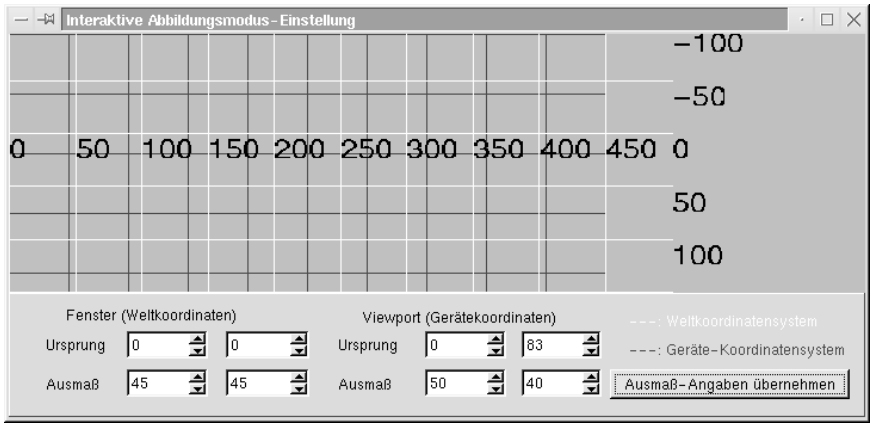


Abbildung 6.8: Ein selbst definiertes Koordinatensystem mit nach unten verschobenem Ursprung, Dehnung in X-Richtung und Stauchung in Y-Richtung

QPainter-Methoden

In Qt ist die Klasse *QPainter* für Window und Viewport zuständig. Sie besitzt *setWindow*- und *setViewport*-Methoden, mit denen Sie das Rechteck ganz einfach definieren önnen. Mit der Methode *setViewXForm* können Sie die Verwendung von Window und iewport an- und abschalten und mit den Funktionen *hasViewXForm*, *window* und *viewport* die gegenwärtigen Einstellungen abfragen. In Object Pascal werden diese Methoden wie üblich über die C-Schnittstelle angesprochen, in der noch das Präfix *QPainter_*« und ein erster Handle-Parameter zur Methodenschnittstelle hinzukommt.

Auch wenn in diesem Kapitel die *QPainter*-Methoden nicht direkt verwendet werden, önnen Sie die hier beschriebenen Vorgänge leicht auf *QPainter* übertragen: Wo im Folgenden die Methoden *SetWindowOrg* und *SetWindowExt* aufgerufen werden, verwenden Sie in *QPainter* die Methode *setWindow*, die die Funktion der beiden vorher genannten Methoden vereinigt. Entsprechendes gilt für die Methoden *SetViewportOrg* und *SetViewportExt*, die durch die *QPainter*-Methode *setViewport* ersetzt werden. Die im Folgenden verwendeten Abbildungsmodi (auch *mmLoMetric*) können Sie mit Qt nachbilden, indem Sie Window und Viewport passend einstellen. Wie diese Einstellung für *mmLoMetric* abläuft, ist am Ende von Kapitel 6.6.3 gezeigt (R112).

Der Aufruf der Qt-Methoden verläuft nach den in Kapitel 5.4.3 beschriebenen Prinzipien. Ein Beispiel für den direkten Aufruf von *setWindow* und *setViewport* finden Sie auf der CD im erwähnten Testprogramm *CoordinateSystemTestForm*.

TVirtualCanvas – Erweiterung für TCanvas

Da der TreeDesigner einige Grafikfunktionen benutzt, die nicht von *TCanvas* bereitgestellt werden, verwenden die meisten Zeichenmethoden des TreeDesigners eine andere Klasse: *TVirtualCanvas*, die auf der CD in der Unit *VCanvas* zu finden ist. *TVirtualCanvas* hat drei wichtige Aufgaben:

- Alle von *TCanvas* angebotenen öffentlichen Methoden und Properties zur Verfügung zu stellen, damit ein *TVirtualCanvas*-Objekt anstelle eines *TCanvas*-Objekts verwendet werden kann.
- Methoden und Properties für alle zusätzlichen vom TreeDesigner benötigten Grafikfunktionen zu definieren (virtuelle Koordinatensysteme, Clipping, alle von Qt bekannten Ausrichtungsoptionen für Text und weitere Funktionen, die für den Zeilenumbruch der mehrzeiligen Grafikelemente benötigt werden).
- Alle zusätzlichen Methoden und Properties sowohl unter Windows als auch unter Linux zu implementieren. Die Unit *VCanvas* verwendet dafür zahlreiche Blöcke, die bedingt mit *$ifdef Windows* und *$ifdef Linux* kompiliert werden. Der Code, der *TVirtualCanvas* benutzt, sollte sowohl unter Linux mit der CLX als auch unter Windows mit der VCL (ohne Qt!) die erweiterten Grafikfunktionen wir Clipping, virtuelle Koordinatensysteme etc. verwenden können.

Verwendung von TVirtualCanvas *R105*

Da die CLX grundsätzlich nur ihre eigenen *TCanvas*-Objekte erzeugt, müssen wir das *TVirtualCanvas*-Objekt selbst erzeugen und es mit dem vorgegebenen *TCanvas*-Objekt verbinden, damit es mit diesem zusammenarbeitet[2]. Für die Verbindung mit dem *Canvas* muss das *VCLCanvas*-Property gesetzt werden (bei der Verwendung der CLX ist damit eigentlich *CLXCanvas* gemeint):

```
VCanvas := TVirtualCanvas.Create;     // Erzeugung des Objekts
VCanvas.VCLCanvas := Paintbox.Canvas;  // Verbindung mit Canvas-Objekt
VCanvas.Rectangle(...)                // Aufruf von TCanvas-Methoden
VCanvas.SetClippingEllipse(...)        // Aufruf von neuen Methoden
```

Wenn das *VCanvas*-Objekt nicht mehr benötigt wird, können Sie es mit *VCanvas.Free* freigegeben (optimal wäre es natürlich, alles, was dem *Create*-Aufruf folgt, in einen *try...finally*-Block einzuschließen und *Free* im *finally*-Teil aufzurufen). Die Freigabe des *TVirtualCanvas*-Objekts hat im Übrigen keinen Einfluss auf das damit verbundene *TCanvas*-Objekt der CLX).

Das Dokumentformular des TreeDesigners geht auf eine besondere Weise mit *TVirtualCanvas* um: Da hier praktisch ununterbrochen (bei jeder Mausbewegung und jeder Zeichenoperation) ein Objekt dieser Klasse benötigt wird, wird eine *TVirtualCanvas*-Variable in der Formularklasse gespeichert, auf die über ein Property zugegriffen werden kann:

```
TDocumentForm = class(TForm)
private
  FPaintboxVCanvas: TVirtualCanvas;
  function GetPaintboxVCanvas: TVirtualCanvas;
public
  property PaintBoxVCanvas: TVirtualCanvas read GetPaintboxVCanvas;
```

Die Property-Lese-Methode erzeugt das Objekt, falls noch keines existiert:

```
function TDocumentForm.GetPaintboxVCanvas: TVirtualCanvas;
begin
  if FPaintBoxVCanvas = nil then
    FPaintboxVCanvas := TVirtualCanvas.Create;
  Result := FPaintboxVCanvas;
end;
```

Da das Objekt erst beim Schließen des Formulars freigegeben wird, findet diese Erzeugung für jedes Dokumentfenster nur einmal statt.

2. Dies ist auch der Grund dafür, dass *TVirtualCanvas* nicht von *TCanvas* abgeleitet ist: Die Ableitung wäre nur sinnvoll, wenn *TVirtualCanvas* das bestehende *TCanvas*-Objekt *ersetzen* könnte. Da die CLX in jedem Fall ihr eigenes *TCanvas*-Objekt erzeugt, muss *TVirtualCanvas* mit diesem kooperieren und braucht selber nicht von *TCanvas* abgeleitet zu sein.

Hinweis: Ein solches Property, bei dessen erstem Lesezugriff ein dahinter liegendes Objekt erzeugt wird, eignet sich hervorragend zur Initialisierung von Objekten, für die nicht bekannt ist, wann genau sie zum ersten Mal benötigt werden. Muss *PaintBoxVCanvas* schon vor der *OnCreate*-Methode initialisiert sein oder erst danach? Wann treten die ersten *OnMouseMove* und *OnPaint*-Ereignisse auf, die das *TVirtualCanvas*-Objekt benötigen? Die Property-Lesemethode stellt sicher, dass das Objekt keine Sekunde zu früh und keine Sekunde zu spät erzeugt wird (was natürlich eher ein ideeler Gewinn ist, denn das Objekt verbraucht nur wenige Bytes an Speicher).

In Kapitel 6.5.2 wurde bereits gezeigt, wie die *OnPaint*-Methode ein *TVirtualCanvas*-Objekt konstruiert. Dies war jedoch eine Vereinfachung, die nicht der Realität im TreeDesigner entspricht. Der tatsächliche Code beginnt wie folgt:

```
procedure TDocumentForm.PaintBoxPaint(Sender: TObject);
var
  VCanvas: TVirtualCanvas;
  ClipRectLP: TRect;
begin
  PaintBox.Canvas.Start;
  VCanvas := SetMapMode(PaintBox.Canvas, PaintBox);
  Document.PaintBackground(VCanvas); // (weiter ohne Änderung)
  ...
```

Letztlich entscheidet also die Methode *SetMapMode*, die das Koordinatensystem einstellt, welches *TVirtualCanvas*-Objekt erzeugt wird. Sie verwendet lediglich das erläuterte *PaintBoxVCanvas*-Property:

```
function TDocumentForm.SetMapMode(DestCanvas: TCanvas;
  DestControl: TControl): TVirtualCanvas;
var
 R: TRect;
begin
  Result := PaintboxVCanvas;
  Result.VCLCanvas := DestCanvas;
```

Im Rest dieses Kapitels geht es nun um die eigentliche Aufgabe von *SetMapMode* – der Einstellung des virtuellen Koordinatensystems.

Die Koordinatensystem-Schnittstelle von TVirtualCanvas *R106*

Die Methodennamen von *TVirtualCanvas* sind an den Koordinatensystem-Funktionen des Windows-API orientiert, denn *TVirtualCanvas* sollte ja die Portierung des TreeDesigners nach Linux vereinfachen.

Zusätzlich zu den von Qt definierten Eigenschaften bietet *TVirtualCanvas* noch die Wahl eines voreingestellten Abbildungsmodus, in dem eine logische Einheit der Abmessung von 0,1 mm auf dem Ausgabegerät entspricht. (Dieser Abbildungsmodus

trägt unter Windows den Namen *mm_LoMetric* und wurde schon in früheren Versionen des TreeDesigners verwendet, weshalb er nun von *TVirtualCanvas* in einer plattformunabhängigen Weise bereitgestellt wird.)

Im Folgenden sind die in diesem Kapitel benötigten *TVirtualCanvas*-Methoden aufgeführt:

- *SetViewportOrg* setzt den Ursprung des Viewport-Rechtecks,
- *SetViewportExt* setzt dessen Ausdehnung.
- *SetWindowOrg* und *SetWindowExt* definieren auf entsprechende Weise das Window-Rechteck.
- *SetMapMode* setzt einen von vier Abbildungsmodi, die den ähnlich benannten Mapping Modes des Windows-API entsprechen.

Die vier möglichen Abbildungsmodi, die Sie an *SetMapMode* übergeben können, sind:

- *mmText* stellt das normale Gerätekoordinatensystem dar – ohne Window- und Viewport-Transformation (es entspricht also dem, was Sie unter Qt mit *setViewXForm (False)* einstellen können).
- *mmAnIsoTropic* entspricht dem Modus, den Sie unter Qt mit *setViewXForm(True)* erhalten. Sie können Window und Viewport frei definieren und die Koordinaten werden auf die Ausgabefläche projiziert, wie in der Einleitung beschrieben.
- *mmIsoTropic* ist eine kleine Modifikation von *mmAnIsoTropic*, bei der *TVirtualCanvas* automatisch sicherstellt, dass die Skalierung in X-Richtung genauso groß ist wie die Skalierung in Y-Richtung. Quadratische Pixel vorausgesetzt erhalten Sie so immer korrekte Kreise, wenn Sie in diesem Modus mit Window- und Viewport arbeiten. Dieser Modus wird bei der Bildschirmausgabe des TreeDesigners verwendet (dabei stellt der TreeDesigner die Seitenverhältnisse von Window und Viewport so ein, dass auch der Modus *mmAnIsoTropic* verwendet werden könnte).
- *mmLoMetric* führt zu einer automatischen Einstellung zu Window und Viewport, so dass eine logische Einheit der Abmessung von 0,1 Millimetern entspricht. Dieser Modus wird im TreeDesigner für die Druckerausgabe verwendet. Zu beachten ist jedoch, dass die Richtung der Y-Achse in diesem Modus umgedreht ist (für die Kompatibilität mit Windows): Die Y-Koordinaten steigen nach oben an.

Die folgenden Abschnitte geben zunächst einige allgemeine Anwendungsbeispiele für die Koordinatensystem-Methoden von *TVirtualCanvas*.

Hinweis: Die Klasse *TVirtualCanvas* wurde geschrieben, bevor bekannt war, dass Delphi für Linux auf einer Bibliothek mit Window- und Viewport-Unterstützung basieren würde. Daher macht *TVirtualCanvas* intern gar keinen Gebrauch von Qt-Window und -Viewport, sondern nimmt unter Linux alle Koordinatentransformationen selbst vor. Dafür redefiniert die Klasse alle Grafikausgabemethoden von *TCanvas*, weshalb es wichtig ist, dass jegliche Grafikausgabe über Methodenaufrufe von *TVirtualCanvas* stattfindet und nicht über *TCanvas*-Methoden oder Qt-Aufrufe, es sei denn, Sie berechnen vorher die Koordinaten für die *TCanvas/Qt*-Methoden mit *LPtoDP* in Gerätekoordinaten um.

Der Ursprung

Mit den Methoden *SetViewportOrg* und *SetWindowOrg*, die Sie bei allen Abbildungsmodi verwenden können, verschieben Sie das virtuelle Koordinatensystem relativ zur Ausgabefläche. Per Voreinstellung befindet sich die logische Koordinate (0, 0) immer in der linken oberen Ecke der Ausgabefläche des Fensters oder Druckers, unabhängig davon, ob die Y-Koordinaten nach oben oder unten hin ansteigen.

- Das Haupteinsatzgebiet von *SetViewportOrg* ist eine feste Einstellung der Ausgabefläche, die sich zur Programmlaufzeit normalerweise nicht ändert: *SetViewportOrg* setzt den Ursprung des Viewports, an dem sich per Voreinstellung die logische Koordinate (0, 0) befindet, an eine andere Position des Ausgabegeräts. Häufig soll beispielsweise nicht die linke obere, sondern die rechte untere Ecke der Ausgabefläche den Ursprung beinhalten. Angenommen, die Größe der Fensterfläche befindet sich im Property *TPaintBox.ClientHeight*. Um den Ursprung auf dem Bildschirm in die linke untere Ecke zu verschieben, müssten Sie *SetViewPortOrg* wie folgt aufrufen:

  ```
  SetViewportOrg(0, PaintBox.ClientHeight);
  ```

- Während *SetViewportOrg* in Geräteeinheiten arbeitet, verwendet *SetWindowOrg* logische Einheiten. Mit dieser Funktion können Sie eine andere logische Koordinate angeben, die statt (0,0) am Ursprung des Viewports erscheinen soll. Hat der Benutzer beispielsweise die beiden Bildlaufleisten eines Fensters auf die Position (450, 560) eingestellt, so können Sie diesen Punkt mit

  ```
  SetWindowOrg(450, 560, nil)
  ```

  in die linke obere Ecke des Fensters bzw. dahin positionieren, wo Sie den Viewport mit *SetViewportOrgEx* gebracht haben. (Um Verwechslungen zu vermeiden: Der TreeDesigner geht beim Scrollen nicht auf diese Weise vor.)

Die Größenverhältnisse

Damit wir *Window* und *Viewport* auch als Rechtecke verstehen können, benötigen wir bildlich gesehen neben den im letzten Abschnitt besprochenen Ursprüngen noch die Seitenlängen. Diese lassen sich nur in den Abbildungsmodi *mmIsotropic* und *mmAnisotropic* sowie mit den Funktionen *SetViewportExt* und *SetWindowExt* verändern.

Wenn Grafik vergrößert oder verkleinert wird, so wird ein bestimmter Abschnitt aus dem Weltkoordinatensystem auf die Ausgabefläche projiziert (Abbildung 6.7). Vergrößern Sie diesen Ausschnitt, so verkleinert sich automatisch die Grafik, falls die Größe der Ausgabefläche gleich geblieben ist. Umgekehrt erreichen Sie eine Vergrößerung, indem Sie einen kleineren Ausschnitt aus dem Weltkoordinatensystem wählen.

Die folgenden Anweisungen dehnen einen Bereich von 100 mal 100 virtuellen Einheiten auf den gesamten Arbeitsbereich eines Fensters aus. Wenn Sie ein Fenster auf diese Weise mit einer Grafik füllen, können Sie diese zur Laufzeit vergrößern und verkleinern, indem Sie einfach die Größe des Fensters anpassen:

```
with VCanvas do begin
  MapMode := mmAnisotropic;
  SetWindowExt(100, 100);
  SetViewportExt(ClientHeight, ClientWidth);
  ... Grafikausgabe ...
end;
```

Hinter der anschaulichen Bezeichnung als »WindowExt/ViewportExt«, also den Ausmaßen von Window und Viewport, versteckt sich jedoch ein abstrakteres Konzept: Das Grafiksystem von Qt interessiert sich gar nicht für die absoluten Werte, die Sie diesen beiden Funktionen übergeben (wenn Sie das Ausmaß des Viewports auf 10*10 Einheiten festlegen, beschränkt dies die Ausgabe nicht auf 10*10 Geräteeinheiten, sondern füllt weiterhin die gesamte Ausgabefläche – zur Beschränkung der Ausgabe müssten Sie Clipping verwenden).

Vielmehr interessiert sich das System nur für das *Größenverhältnis* zwischen logischer Einheit und Geräteeinheit. Im obigen Beispiel bedeutet das z.B., dass in X-Richtung 100 logische Einheiten auf *PaintBox.Height* Geräteeinheiten abgebildet werden. Die beiden folgenden Anweisungen bewirken, dass eine logische Einheit sowohl in X- als auch in Y-Richtung in fünf Geräteeinheiten umgewandelt wird:

```
SetWindowExt(1, 1);
SetViewportExt(5, 5);
```

Gegenüber dem Gerätekoordinatensystem ergibt sich so eine Vergrößerung um den Faktor 5.

Umdrehung der Achsenrichtungen

Auch in den Modi *mmIsotropic* und *mmAnisotropic* sind die Achsenrichtungen zunächst so definiert, dass die Y-Koordinaten nach unten hin ansteigen. Um das zu ändern, genügt es, wenn Sie eine negative Höhe des Viewports angeben:

```
with VCanvas do begin
  SetWindowExt(100, 100);
  SetViewportExt(Width, -Height);
  ...
```

Größenverhältnisse im Modus mmIsotropic

Wenn Sie *SetWindowExt* und *SetViewportExt* im Modus *mmIsotropic* verwenden, passt *TVirtualCanvas* die von Ihnen gemachten Angaben so an, dass eine logische X-Einheit genauso viele Geräteeinheiten misst wie eine logische Y-Einheit. Dabei wird die Grafik unter Umständen etwas kleiner, so dass sie entweder in der Breite oder in der Höhe nicht mehr das gesamte Fenster abdeckt, aber in jedem Fall bleiben Quadrate quadratisch und Kreise werden nicht zu Ellipsen, sofern die Pixel des Ausgabegeräts quadratisch sind.

6.6.2 Das Koordinatensystem des TreeDesigners

Der TreeDesigner stellt sein Koordinatensystem ein, indem er die Methoden von *TVirtualCanvas* aufruft.

Wie am Anfang von Kapitel 6.5.4 genauer beschrieben, beinhaltet die vollständige reeDesigner-Version auf der CD-ROM die Option, das virtuelle Koordinatensystem ein- und auszuschalten, und auch der Programmcode enthält entsprechende Fallunterscheidungen. Der in diesem Kapitel abgedruckte Code enthält nur die Anweisungen, die bei *eingeschaltetem* virtuellen Koordinatensystem gültig sind.

Anpassung der Zeichenfläche *R101*

Der *TreeDesigner* arbeitet am Bildschirm im Abbildungsmodus *mmIsoTropic* und druckt im Modus *mmLoMetric*. Über den ScrollBar in der Werkzeugleiste können Sie einen Vergrößerungsfaktor zwischen 0.1 und 10 einstellen.

Spätestens an dieser Stelle ist es sinnvoll, die Größe der Zeichenfläche in virtuellen Koordinaten festzulegen. Sie wird in der Unit *GrDoc* dadurch festgelegt, dass die Properties *Width* und *Height* des Grafikdokuments wie folgt initialisiert werden:

```
constructor TGraphicDoc.Create;
begin
  FWidth := 3000;
  FHeight := 3000;
  ...
```

Der Schritt vom Geräte- zum virtuellen Koordinatensystem soll sich nahtlos an die gewählte Methode des Scrollens anschließen. Das heißt, die Klasse *TScrollBox* soll auch dann, wenn die Grafik verkleinert oder vergrößert ist, die beiden folgenden Aufgaben automatisch erledigen:

- Der scrollbare Bereich soll die gesamte Größe der Grafik umfassen
- und beim seitenweisen Blättern mit den Bildlaufleisten soll die Grafik jeweils um den sichtbaren Bildbereich verschoben werden.

Das funktioniert nur, wenn die Paintbox-Zeichenfläche weiterhin den gesamten virtuellen Grafikbereich umfasst. Bei einer Vergrößerung muss also auch die Größe der Paintbox weiterwachsen, so wie sie bei einer Verkleinerung schrumpfen soll. Genau genommen errechnet sich die Größe der Paintbox aus den virtuellen Größen *Document.Width* und *Document.Height*, jeweils multipliziert mit dem Vergrößerungsfaktor. Beim maximalem Vergrößerungsfaktor 10 wird die Paintbox also zu einem 30000 Einheiten großen Quadrat.

Als weitere Maßnahme ist es notwendig, die *TScrollBox*-Properties *VertScrollBar.Range* und *HorzScrollBar.Range* an die geschrumpfte bzw. gewachsene Zeichenfläche anzupassen, denn *TScrollBox* verändert den scrollbaren Bereich nicht automatisch, wenn sich eines der Kindfenster verändert.

Beide Maßnahmen sind in der Methode *ResizePaintbox* zusammengefasst, die auch noch die gesamte Paintbox mit *Invalidate* für ungültig erklärt, denn nach einer Änderung des Vergrößerungsfaktors muss die gesamte Grafik neu ausgegeben werden:

```
procedure TDocumentForm.ResizePaintbox;
begin
  PaintBox.Width := trunc(Document.Width * ZoomFactor);
  PaintBox.Height := trunc(Document.Height * ZoomFactor);
  PaintBox.Invalidate;
end;
```

ResizePaintBox wird im einfachsten Fall aufgerufen, wenn das im Folgenden besprochene Property *ZoomFactor* geändert wird.

Das ZoomFactor-Property

Der Vergrößerungsfaktor selbst befindet sich in der privaten Variablen *FZoomFactor*, die als Property *ZoomFactor* von der Methode *SetZoomFactor* geschrieben wird. *SetZoomFactor* ruft nur dann die Methode *ResizePaintbox* auf, wenn sich der Vergrößerungsfaktor auch geändert hat:

```
TDocumentForm = class(TForm)
  ...
private
  FZoomFactor: real;
```

```
    procedure SetZoomFactor(NewValue: real);
  public
    property ZoomFactor: real read FZoomFactor write SetZoomFactor;
    ...

procedure TDocumentForm.SetZoomFactor(NewValue: real);
begin // vereinfachte Version, siehe Hinweis unten
  // Der Faktor muss zwischen 0,1 und 1 liegen:
  if NewValue < 0.1 then NewValue := 0.1;
  if NewValue > 10 then NewValue := 10;
  if FZoomFactor <> NewValue then begin
    FZoomFactor := NewValue;
    ResizePaintbox; // siehe unten
  end;
  // Aktualisierung des Labels neben dem TrackBar:
  DecimalSeparator := '.'; // kein Komma verwenden
  ZoomLabel.Caption := FormatFloat('#0.####', ZoomFactor);
end;
```

Die wohl häufigste Ursache für die Änderung des Properties *ZoomFactor* ist die *OnChange*-Methode des Vergrößerungsfaktor-Scrollbars (weitere Ursachen sind ein paar Menüpunkte aus dem Ansichtsmenü und die Initialisierung des Fensters beim *OnCreate*-Event). Da ein Scrollbar nur ganze Zahlen als Position liefert, wurde statt eines Bereiches von 0,1 bis 10 der Bereich 1 bis 100 gewählt, folglich muss die Scrollbar-Position wieder in einen Vergrößerungsfaktor umgerechnet werden. Für diese Umrechnung können Sie über das Popup-Menü des TrackBars zwischen zwei Modi wählen: einem linearen Modus, in dem der Bereich zwischen Faktor 0,1 und 1 im ersten Zehntel des TrackBars liegt, und einem exponentiellen Modus, in dem dieser Bereich über ungefähr drei Viertel des TrackBars ausgedehnt ist. Die *OnChange*-Methode berücksichtigt diese beiden Alternativen:

```
procedure TDocumentForm.ZoomFactorBarChange(Sender: TObject);
begin { vereinfachte Version }
  // LinearZooming wird über das Popupmenü des TrackBars eingestellt:
  if LinearZooming then ZoomFactor := ZoomFactorBar.Position / 10
    else ZoomFactor : =0.1 + exp(ZoomFactorBar.Position/10) / 2000;
end;
```

Hinweis: Damit beim Verändern des Vergrößerungsfaktors Schritt für Schritt der Eindruck entsteht, Sie würden auf das Zentrum des Bildes zoomen (oder von diesem wegzoomen), merkt sich die vollständige Methode *SetZoomFactor* vor der Änderung der Paintbox-Größe die Koordinaten des Punktes in der Mitte des Fensters und rückt diesen Punkt nach der Ausführung der oben abgedruckten Zeilen erneut in die Bildmitte.

Einstellung des Abbildungsmodus

Da der Abbildungsmodus und die zugehörigen Daten an mehreren Stellen eingestellt werden müssen, fasst das Dokumentformular des TreeDesigners alle dazugehörigen Aufrufe in einer Methode zusammen:

```
function TDocumentForm.SetMapMode(DestCanvas: TCanvas;
                 DestControl: TControl): TVirtualCanvas;
begin
  Result := PaintboxVCanvas;
  Result.VCLCanvas := DestCanvas;
  if not NoMapMode.Checked then begin
    Result.SetViewPortOrg(PaintBoxHMargin div 2,
                          PaintBoxVMargin div 2);
    Result.SetScrollPos(DestControl.Left, DestControl.Top);
    Result.MapMode := mmIsoTropic;
    Result.SetWindowOrg(0, Document.Height);
    Result.SetWindowExt(Document.Width, -Document.Height);
    Result.SetViewPortExt(DestControl.Width - PaintBoxHMargin,
                          DestControl.Height - PaintBoxVMargin);
  end;
end;
```

Koordinatenumrechnung

Mit den bisher getroffenen Maßnahmen können Sie eine bestehende Grafik im *TreeDesigner* zwar ansehen und blättern; einige Dinge würden so jedoch ohne weitere Vorbereitung noch nicht funktionieren. Wenn Sie in einer von 1.0 verschiedenen Vergrößerungsstufe Objekte einzeichnen oder sonstige Mausaktionen durchführen wollen, geriete die Grafik aus den Fugen: Neu gezeichnete Elemente erscheinen nicht mehr unter dem Mauszeiger und sind vielleicht überhaupt nicht sichtbar, *TGraphicElement.Invalidate* lässt nicht mehr die richtigen Bereiche des Fensters neu zeichnen und auch das Abschneiden der Schrift durch manuelles Clipping ist nicht mehr treffsicher. Solche Probleme können entstehen, wenn Sie in einem Funktionsaufruf virtuelle Koordinaten verwenden, wo Gerätekoordinaten gefragt sind, oder umgekehrt.

DPtoLP R84

Die erste Richtung gilt bei den Methoden für die Mausereignisse: Sie erhalten Koordinaten, die in Geräteeinheiten angegeben sind, die aufgrund des virtuellen Koordinatensystems nicht mehr den Koordinaten der Grafik entsprechen. Eine einfache *TVirtualCanvas*-Funktion verspricht hier Abhilfe: *DPtoLP* ist eine Abkürzung für »Device Points to Logical Points« und transformiert dementsprechend Gerätekoordinaten in logische Koordinaten.

Die in Kapitel 6.4.1 abgedruckten Methoden für die Bearbeitung von *OnMouseDown*, *OnMouseMove* und *OnMouseDown* berücksichtigen die virtuellen Koordinatensysteme, indem sie die *x/y*-Werte der Maus durch die Methode *MapMousePos* umrechnen lassen. *MapMousePos* besteht im Wesentlichen nur aus dem Aufruf der *DPtoLP*-Funktion:

```
procedure TDocumentForm.MapMousePos(var x, y : integer);
var
  P : TPoint;
begin
  P := Point(x, y);
  PaintboxVCanvas.DPtoLP(P);
  x:=P.x; y:=P.y;
end;
```

Hinweis: Statt *MapMousePos* könnte unter Linux auch direkt die überladene Methode *DPtoLP(x,y)* verwendet werden. Dass trotzdem eine Methode *MapMousePos* existiert, hängt damit zusammen, dass unter Windows noch die Position des Scrollbars einberechnet werden muss, weshalb die tatsächliche *MapMousePos*-Methode auf der CD noch etwas größer ist und es sich eher lohnt, diesen Code in einer eignen Methode aufzubewahren.

LPtoDP

Entsprechend funktioniert die Umwandlung von logischen Koordinaten in Gerätekoordinaten mit der Funktion *LPtoDP*. Gerätekoordinaten werden im TreeDesigner benötigt, um den Clipping-Bereich einzustellen und rechteckige Bereiche des Fensters für ungültig zu erklären. Der Quelltext der entsprechenden Methoden ist bereits in den Kapiteln 6.5.3 und 6.5.5 gezeigt worden.

6.6.3 Drucken

Der unter Linux übliche Weg der Ausgabe von Grafik auf den Drucker besteht darin, ein PostScript-Dokument zu erzeugen und dieses von einem im Hintergrund wirkenden Tool in druckerabhängige Befehle zu übersetzen oder das Dokument direkt über einen PostScript-Drucker auszugeben. Die Qt-Bibliothek bietet eine hervorragende Unterstützung dieser Art des Druckens und »vererbt« diese Unterstützung an die *TCanvas*-Klasse der CLX. Im Endeffekt können Sie dadurch dieselben Grafikmethoden von *TCanvas*, mit denen Sie im Fenster zeichnen, auch für die Druckerausgabe verwenden. Allerdings gibt es beim Drucken einige Abweichungen zur Grafikausgabe im Fenster:

- Die Auflösung eines Druckers ist erheblich höher als die der heutigen Bildschirme.
- Die Seite, auf der der Drucker druckt, verhält sich insofern anders, als dass sie sich nur zeitweise im Drucker befindet. Beim Drucken auf mehreren Seiten müssen Sie

steuern, wann die Seite gewechselt wird. Auch dann, wenn Sie nur eine Druckseite verwenden, müssen Sie festlegen, wann diese beginnt und wann sie ausgeworfen werden kann.

- Drucker weisen untereinander und im Vergleich zum Bildschirm unterschiedliche Fähigkeiten und Beschränkungen auf. So können beispielsweise Plotter keine Bitmaps zeichnen. Die Tatsache, dass Drucker die gedruckte Grafik nicht mit weißer Farbe übermalen können, wird bereits in den Grafiktreibern (oder Druckern) berücksichtigt, die eine Seite im Speicher aufbauen und erst dann drucken, wenn sie vollständig fertig ist.
- Die Zeichenfläche des Druckers ist geringfügig komplizierter zu erreichen als die Zeichenfläche eines Formulars: Sie müssen zuerst die Unit *Printers* einbinden und dann auf deren *Printer*-Objekt zugreifen, um an die Drucker-Zeichenfläche (*Printer.Canvas*) zu gelangen.

Textausgabe schnell und einfach *R108*

Sie können in Kylix auf zwei verschiedene einfache Weisen Ausgaben an den Drucker senden: Zur reinen Textausgabe kommen Sie mit den schon fast antiken Prozeduren *write* und *writeln* sowie mit der Prozedur *AssignPrn* aus. Für Letztere müssen Sie die Unit *QPrinters* in die Unit einbinden, in der Sie den Drucker verwenden wollen.

AssignPrn verbindet den Drucker mit einer Textdateivariablen, die Sie so verwenden können wie jede andere derartige Variable, die mit einer normalen Datei verbunden ist (Siehe Kapitel 2.8.2). Das bedeutet, dass Sie die »Datei« (im folgenden Beispiel die Variable *DruckerAlsDatei*) mit *Rewrite* öffnen und am Schluss mit *Close* wieder schließen müssen. Zwischen diesen beiden Anweisungen geben Sie die zu druckenden Zeilen mit der Standardfunktion *writeln* aus, der Sie eine beliebig lange Liste von Parametern verschiedenen Typs (ohne die eckigen Klammern, die bei anderen Funktionen mit variabler Parameterzahl notwendig sind) übergeben können. Das folgende Beispiel ist dem Projekt *TextPrinter* von der CD-ROM entnommen:

```
var
  DruckerAlsDatei: System.Text;
begin
  AssignPrn(DruckerAlsDatei);
  Rewrite(DruckerAlsDatei);
  Writeln(DruckerAlsDatei, 'Dateiausgabe vom ', DateToStr(Now));
  for i := 0 to Memo1.Lines.Count - 1 do
    Writeln(DruckerAlsDatei, Memo1.Lines[i]);
  ...
  System.Close(DruckerAlsDatei);
  ...
```

Da in Formularen die Bezeichner *Close* und *Text* bereits für eine Methode bzw. für ein Property reserviert sind, müssen Sie in Formularmethoden den Bezeichner *System* vor diese Bezeichner stellen, damit der Compiler erkennt, dass Sie nicht die Methode bzw. das Property des Formulars, sondern die Funktion bzw. den Typ der Standardbibliothek *System* meinen.

AssignPrn verbindet die Textdatei nicht direkt mit dem Drucker, sondern mit dem Druckerobjekt *Printer* der Unit *Printers* (im Stichwortverzeichnis zu finden unter der Klasse *TPrinter*). Dieses leitet die Textdaten als Grafik an den Druckertreiber weiter und verwendet dabei die in seinem Property *Canvas.Font* angegebene Schriftart. Um also die Schriftart für den gedruckten Text zu setzen, schreiben Sie sie in das *TFont*-Objekt *Printer.Canvas.Font*. Das zuständige *Printer*-Objekt erhalten Sie als Rückgabewert von der Funktion *AssignPrn*, so dass in obigem Listing die erste Anweisung etwa wie folgt erweitert werden könnte, um die Schrift auf die aktuelle Wahl in einem Schriftdialog zu setzen:

```
AssignPrn(DruckerAlsDatei).Canvas.Font := FontDialog1.Font;
```

Grafikausgabe schnell und einfach *R109*

Auch bei der Grafikausgabe kommt *Printer.Canvas* ins Spiel, und zwar verwenden Sie dieses Objekt wie die Zeichenfläche des Formulars. So können Sie mit *Draw* ganze Grafiken wie Bitmaps oder Metadateien ausgeben, sie mit *StretchDraw* vergrößern, so dass sie auch auf dem Drucker nicht zu klein erscheinen, und einfache Grafikbefehle wie *LineTo* und *TextOut* verwenden. Für die Seitensteuerung müssen Sie vor der ersten Ausgabe auf die Seite die Methode *BeginDoc*, für jede neue Seite die Methode *NewPage* und am Ende des Druckauftrags die Methode *EndDoc* aufrufen. Das Grundgerüst der Grafikausgabe sieht demnach so aus:

```
Printer.BeginDoc;
with Printer.Canvas do begin
  ... Grafikausgabe ...
end;
Printer.EndDoc;
```

Falls Sie einen mit *BeginDoc* gestarteten Druckauftrag abbrechen wollen, rufen Sie statt *EndDoc* die Methode *Abort* auf.

Druckerausgabe und Abbildungsmodi

Bei der Portierung der Ausgabe vom Fenster auf den Drucker ist es entscheidend, dass Sie auf dem Drucker nicht dieselben *Gerätekoordinaten* verwenden wie am Bildschirm, denn durch die erheblich höhere Auflösung eines Druckers würde ein unverändert auf den Drucker übertragener Fensterinhalt beträchtlich schrumpfen. Unter Linux stellt sich jedoch das Problem der Drucker-Gerätekoordinaten kaum, wenn Sie dort die Post-

Script-Schnittstelle verwenden. Denn bei dieser werden die Grafik-Koordinaten nicht als Gerätekoordinaten, sondern als PostScript-Einheiten interpretiert. Eine solche Einheit misst 1/72 Zoll (ca. 0,35 mm), mit anderen Worten: Das virtuelle Druckerkoordinatensystem hat eine Auflösung von 72 dpi. Verglichen mit der wahren Druckerauflösung ist dies natürlich sehr niedrig und ein klarer Nachteil liegt darin, dass die Positionierung von Grafikausgaben nur in einem relativ groben Raster möglich ist. Die Verwendung dieser Standard-PostScript-Auflösung von 72 dpi besagt jedoch nicht, dass auch der Drucker in dieser Auflösung druckt. Dieser kann seine Ausgabe in beliebig hoher Auflösung vornehmen, so dass also schräge Linien und Schrift keineswegs grob gerastert auf das Papier gelangen. Nur die den Grafikausgabemethoden übergebenen (Linienend-)Koordinaten fallen in das 72-dpi-Raster.

Der Vorteil bei dieser Verwendung von PostScript liegt darin, dass die Druckerausgabe eine der Bildschirmansicht vergleichbare Größe erhält (wenn der Bildschirm eine höhere Auflösung als 72 dpi hat, erscheint die Grafik sogar auf dem Bildschirm etwas kleiner), ohne dass Sie Koordinatenumrechnungen vornehmen oder Koordinatensysteme einstellen müssen.

In einem seiner beiden Druckmodi steuert auch der TreeDesigner den Ausdruck in genauen Maßeinheiten, im anderen Modus nimmt er es nicht so genau, lässt dem Benutzer dafür aber im Drucken-Dialog mit Vorschau eine erheblich größere Flexibilität.

TreeDesigner-WYSIWYG 1/2 *R110*

Im ersten seiner beiden Druckmodi verwendet der TreeDesigner den Abbildungsmodus *mmLoMetric* der Klasse *TVirtualCanvas* und legt den Ursprung des Drucker-Viewports in Übereinstimmung mit der Darstellung am Bildschirm auf die linke untere Ecke der Seite.

Eine Einheit auf dem Drucker ist im Modus *mmLoMetric* 0,1 mm groß. Das heißt also, dass 100 virtuelle Einheiten auf dem Bildschirm zu einem Zentimeter Grafik auf dem Drucker werden. Damit der Druck genau mit der Darstellung auf dem Bildschirm übereinstimmt, muss der Vergrößerungsfaktor auf dem Bildschirm so angepasst werden, dass 100 an den Linealen angezeigte Einheiten ein Zentimeter groß dargestellt werden (dies ist ungefähr bei Faktor 0,2 der Fall).

Beim Drucken werden außerdem dieselben Grafikausgabemethoden verwendet wie für die Anzeige des Dokuments am Bildschirm. Bis auf die wahrscheinlich leicht unterschiedliche Größe ist die Bildschirmdarstellung damit eine WYSIWYG-Darstellung (what you see is what you get), die der Druckerausgabe entspricht, soweit es die Unterschiede zwischen Drucker und Bildschirm zulassen. Eine 29,7 cm lange DIN-A4-Seite passt mit ihren 2970 logischen Einheiten gerade in die 3000 Einheiten hohe Zeichenfläche hinein.

Der erste Druckmodus des TreeDesigners lässt Ihnen zwar beim Zeichnen einer Grafik die Kontrolle über die genaue Größe der Grafikelemente auf dem Ausdruck, dies ist jedoch nicht immer die Art der Kontrolle, die man bei einem Grafikprogramm dieser Art benötigt: So wie am Bildschirm sollte sich die Grafik auch für den Ausdruck skalieren lassen, damit sie z. B. genau auf eine Seite eingepasst werden kann. Um die Grafik automatisch skalieren zu lassen, bietet sich hier wieder die Verwendung von Window- und Viewport-Transformation an, bzw. bei der Verwendung von *TVirtualCanvas* im TreeDesigner der Abbildungsmodus *mmIsoTropic*, den der TreeDesigner schon bei der Bildschirmausgabe verwendet.

Hinweis: Der TreeDesigner könnte auch ganz ohne Koordinatensystemeinstellungen einfach im Standard-PostScript-Modus drucken. Aus historischen Gründen entspricht jedoch im TreeDesigner eine virtuelle Einheit des Grafikdokuments dem Maß von 0,1mm im Ausdruck, daher wird stattdessen *TVirtualCanvas* und *mmLoMetric* verwendet. Um den TreeDesigner auf den Standard-PostScript-Modus umzustellen, würde es genügen, die Koordinatensystemeinstellungen aus der Druckmethode zu entfernen und bei der Bearbeitung des Dokuments am Bildschirm sichtbar zu machen, dass eine Einheit nun einer PostScript-Einheit entspricht. Dies würde jedoch die »Auflösung« der TreeDesigner-Dokumente um den Faktor 3,5 reduzieren, wäre also auch noch nicht der Weisheit letzter Schluss.

eeDesigner-WYSIWYG 2/2: Druckvorschaudialog *R111*

Die erwähnte Einschränkung der Flexibilität beim Ausdruck führt uns zur zweiten Form der WYSIWYG-Darstellung im TreeDesigner, der Vorschaugrafik im Druckdialog (Abbildung 6.9). Anhand der Vorschaugrafik können Sie in diesem Dialog im Modus *Druckgröße anpassen* genau einstellen, wie groß und an welcher Position der Seite die Grafik gedruckt werden soll.

- Im Bereich *Zu druckender Ausschnitt* werden die Koordinaten des Ausschnittes angegeben, der gedruckt werden soll. Da dies normalerweise immer die gesamte Grafik ist, genügt hier ein Druck auf *Optimal*, um den Druckdialog selbst herausfinden zu lassen, wie groß die Grafik ist (der Dialog bedient sich dazu der in Kapitel 6.3.2 erwähnten Methode *TGraphicDoc.GetPaintRect*). Per Voreinstellung wird der gewählte Ausschnitt so groß gedruckt, dass er die gesamte Seite bedeckt (allerdings ohne das Seitenverhältnis zu verzerren, wie es im Modus *mmAnisoTropic* möglich wäre).

Der Bereich *Zielfläche in Prozent* dient dazu, Ränder einzustellen. Mit *X* und *Y* verschieben Sie die Grafik von der linken oberen Ecke nach rechts und nach unten, mit *Breite* und *Höhe* können Sie die Voreinstellung von 100% des Seitenmaßes verkleinern.

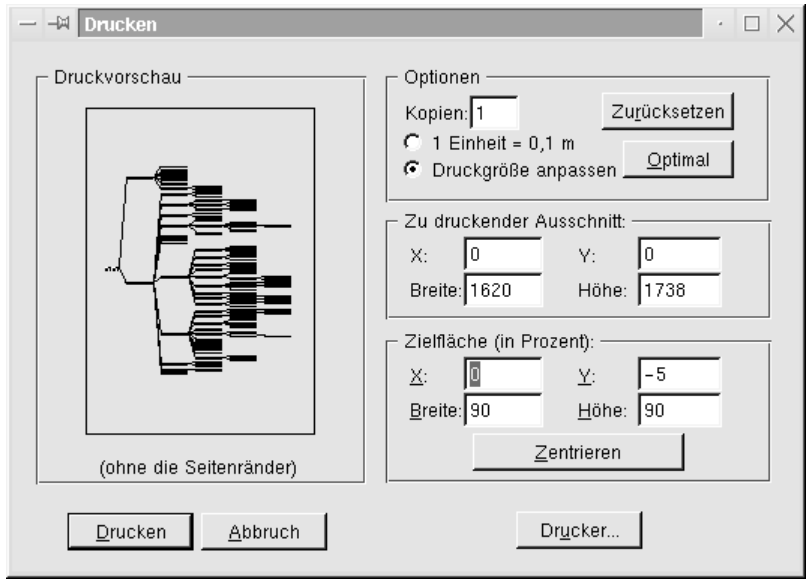


Abbildung 6.9: Nach Drücken des Schalters »Optimal«, Verkleinern der Zielfläche auf 90% und Verschieben der Zielfläche um 5% in Y-Richtung wird diese Grafik zentriert gedruckt und füllt die Seite fast ganz aus.

Eine Beispieleinstellung dieser Werte finden Sie in Abbildung 6.9. Zur Implementierung dieses zweiten Druckmodus sind zwei neue Einstellungen von Koordinatensystemen erforderlich:

- Für die Vorschau muss die gesamte Grafik bzw. der gewählte Ausschnitt in einen vergleichsweise winzigen Bereich des Dialogs gezwängt werden, der in Pixeln angegeben wird (ca. 200*230 Pixel, der Dialog enthält dafür eine *TPaintBox*-Komponente).
- Für den Ausdruck muss der gewählte Ausschnitt der Grafik auf den im Dialog angegebenen Zielbereich abgebildet werden, der in Druckereinheiten angegeben wird. Beim Druck in PostScript-Einheiten à 72 dpi sind das z.B. 841*595 Einheiten für die gesamte Seite.

Im Folgenden ist der Code abgedruckt, der im Falle des frei skalierbaren Modus die Druckvorschau zeichnet. Dafür wird das Koordinatensystem so eingestellt, dass der in den Eingabefeldern des Dialogs eingestellte Dokumentbereich auf die (ebenfalls im Dialog eingestellte) Zielfläche des Papiers (hier das Papier in der Vorschau-Grafik) projiziert wird:

```
// Auszug aus der OnPaint-Methode für den Vorschaubereich des Dialogs:
// VCanvas bezieht sich in diesem Fall auf die Paintbox des Dialogs.
if (not LoMetric.Checked) then begin
  VCanvas.SetWindowOrg(WinX.Value, WinY.Value);
```

```
    VCanvas.SetWindowExt(WinW.Value, WinH.Value);
    VCanvas.SetViewPortExt(PrevW, -PrevH);
    VCanvas.SetViewPortOrg(PrevX, PrevH+PrevY);
    Doc.PaintAll(VCanvas);
  end;
```

PrevX, *PrevY*, *PrevW* und *PrevH* werden vorher in derselben Methode berechnet, sie geben die Position, Breite und Höhe des Teils im Vorschaubereich an, der der Einstellung der *Zielfläche* im Dialog entspricht (also einen Teilbereich innerhalb der weiß gezeichneten Seitenfläche; zur Berechnung dieser Werte siehe die CD-ROM). Auf genau diesen Bereich soll nämlich der gewählte Ausschnitt der Grafik abgebildet werden. Für die Einstellung dieses Ausschnitts stellt der Dialog, wie in Abbildung 6.9 dargestellt, vier Eingabefelder zur Verfügung, deren Eingabe-Elemente *WinX*, *WinY*, *WinW* und *WinH* in den obigen Zeilen abgefragt werden.

Dies ist jedoch nur ein kleiner Auszug aus dem vollständigen Code, der auch noch den weißen Seitenhintergrund zeichnen, Ränder vom Seitenhintergrund zu den Grenzen des Vorschaubereichs berechnen und die verschiedenen alternativen Dialogeinstellungen berücksichtigen muss.

Hinweis: Die Verkleinerung der Ausgabefläche im Dialog ist nur dann wirksam, wenn Sie die gesamte Grafik ausdrucken, denn der TreeDesigner verwendet beim Drucken kein zusätzliches Clipping. Es wird also auch außerhalb der Ränder gedruckt, wenn die Grafik noch über den Bereich hinausgeht, den Sie in *Zu druckender Ausschnitt* eingestellt haben.

Geräteunabhängigkeit

Das elementare Prinzip, das beim Drucken und beim Vorschaudialog verwendet wird, ist dasselbe wie beim Zoomen der Grafik auf dem Bildschirm: die völlige Geräteunabängigkeit eines virtuellen Koordinatensystems. Überall wird die TreeDesigner-Grafik mit einem einzigen Aufruf von *TGraphicDoc.PaintAll* ausgegeben:

- am Bildschirm, wenn dort das Gerätekoordinatensystem verwendet wird,
- am Bildschirm, wenn dort ein virtuelles Koordinatensystem mit einer beliebigen Vergrößerungseinstellung verwendet wird,
- in der Druckvorschau, bei der die Grafik noch stärker verkleinert wird als beim kleinsten Zoomfaktor im Grafikfenster,
- beim Drucken im Modus *mmLoMetric*,
- beim Drucken im frei skalierbaren Modus.

Druckereinrichtung

Wir kommen nun zur Implementierung der Menüpunkte für das Drucken. Der TreeDesigner stellt dafür in seinem Dateimenü die Punkte DRUCKEN... und DRUCKEREINRICHTUNG... zur Verfügung.

Zum Einrichten des Druckers bietet Qt einen speziellen Dialog an (siehe Abbildung 6.10), in dem Sie einen der installierten Drucker oder eine Datei als Ziel für die folgende Druckerausgabe wählen können. In der CLX rufen Sie diesen Dialog über das *Printer*-Objekt auf, für dessen Nutzung Sie die Unit *QPrinters* einbinden müssen.

```
procedure TDocumentForm.CmPrinterSetupClick(Sender: TObject);
begin
  Printer.ExecuteSetup;
end;
```

Die vom Benutzer vorgenommenen Einstellungen des Dialogs werden wirksam, wenn Sie anschließend über dasselbe *Printer*-Objekt, über das Sie *ExecuteSetup* aufgerufen haben, auch den Ausdruck laufen lassen. Der TreeDesigner verwendet für beides das globale *Printer*-Objekt der Unit *QPrinters*.

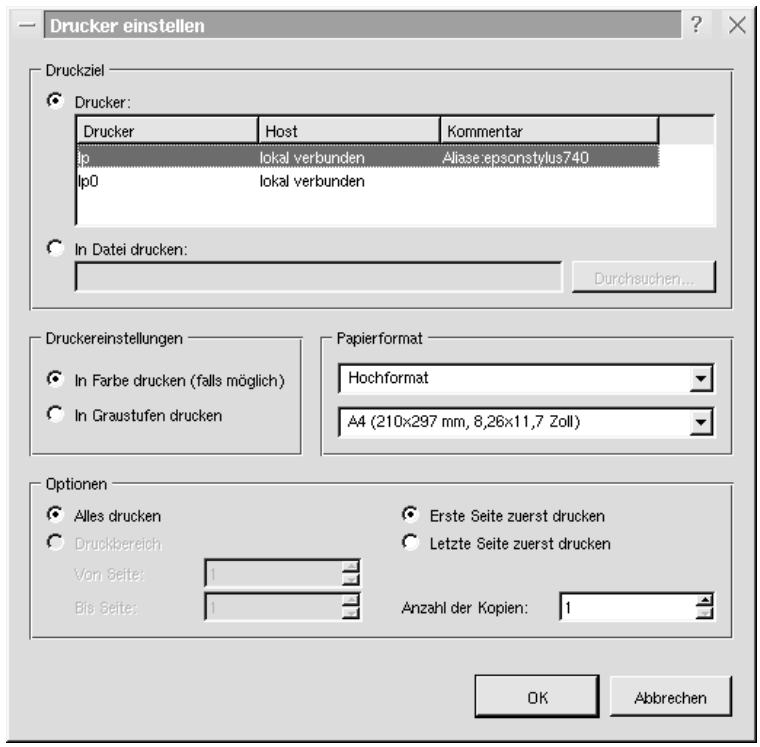


Abbildung 6.10: Der Druckereinrichtungsdialog von Qt

Die Druckmethode

Nun zur Methode für das *OnClick*-Ereignis des Menüpunkts DATEI | DRUCKEN, durch den der besprochene und in Abbildung 6.9 gezeigte Druckdialog aufgerufen werden soll. Wenn dieser mit *OK* beendet wird, werden die folgenden Properties des Dialogs ausgewertet:

- *Copies.Value* ist der Wert des *TSpinEdit*-Feldes für die Kopienanzahl. Dieser Wert muss nur an *Printer.Copies* weitergegeben werden, damit automatisch die entsprechende Anzahl an Kopien gedruckt wird.
- *LoMetric.Checked* ist der Zustand des Radioschalters *Druckgröße anpassen*.
- *WinX/WinY/WinW/WinH.Value* sind die schon erwähnten *TUpDown*-Werte im Dialogbereich *Zu druckender Ausschnitt*.
- *VPX/VPY/VPW/VPH.Value* sind die entsprechenden Werte im Dialogbereich *Zielfläche*.

Das Drucken selbst läuft zwischen den schon erwähnten Aufrufen von *BeginDoc* und *EndDoc* des *Printer*-Objekts ab. Der folgende Ausschnitt zeigt auch den Code für den erstgenannten Druckmodus, in dem das *mmLoMetric*-Koordinatensystem verwendet wird:

```
procedure TDocumentForm.PrintClick(Sender: TObject);
var
  VCanvas: TVirtualCanvas;
begin
  Document.DeselectAll; // Markierungen würden sonst mitgedruckt werden
  PrintDialog.Doc := Document; // Dem Dialog das Dokument übergeben
  if PrintDialog.ShowModal = mrOK then begin
    Screen.Cursor := crHourGlass; // Sanduhrsymbol anzeigen
    Printer.Title := 'TreeDesigner Druckauftrag';
    Printer.Copies := PrintDialog.Copies.Value;
    Printer.BeginDoc;
    VCanvas := TVirtualCanvas.Create;
    VCanvas.VCLCanvas := Printer.Canvas;
    with PrintDialog do begin
      if LoMetric.Checked then begin
        VCanvas.MapMode := mmLoMetric; { 0.1 mm pro Einheit }
        VCanvas.SetWindowOrg(0, Document.Height);
        // Die im Dialog eingestellte Verschiebung berücksichtigen:
        VCanvas.SetViewportOrg(Printer.PageWidth * VPX.Value div 100,
                               Printer.PageHeight * VPY.Value div 100);
      end else begin
        VCanvas.MapMode := mmIsoTropic;
        VCanvas.SetWindowOrg(WinX.Value, WinY.Value);
        VCanvas.SetWindowExt(WinW.Value, -WinH.Value);

        VCanvas.SetViewportExt(Printer.PageWidth * VPW.Value div 100,
```

```
                                Printer.PageHeight * VPH.Value div 100);
          VCanvas.SetViewportOrg(Printer.PageWidth * VPX.Value div 100,
                  Printer.PageHeight * (VPW.Value+VPY.Value) div 100);
        end;
      end;
      Document.PaintAll(VCanvas);
      Printer.EndDoc;
      Screen.Cursor := crDefault; // normale Mauszeigerform wiederherstellen
    end;
  end;
```

Zu einer Zusammenfassung der Properties von *Printer*, die in dieser Methode verwendet werden, und weiterer Properties siehe die nächste Tabelle.

TPrinter-Properties und die Druckerkonfiguration

Die wichtigen Properties der Klasse *TPrinter*, darunter die oben verwendeten Properties *Title*, *Copies*, *PageWidth* und *PageHeight*, sind in der folgenden Tabelle zusammengefasst:

TPrinter-Property	**Beschreibung**
Aborted: Boolean	*True*, wenn der aktuelle Druckauftrag durch den Benutzer abgebrochen wurde; könnte oben in *CmPrintClick* abgefragt werden, um den Druck vorzeitig zu beenden.
Canvas	Zeichenfläche des Druckers, in *Font* können Sie eine Schrift für die Ausgabe von Textdaten (siehe obiger Abschnitt *Textausgabe schnell und einfach*) wählen, während eines mit *BeginDoc* gestarteten Druckauftrags können Sie die gewohnten *TCanvas*-Grafikmethoden aufrufen.
Orientation: TPrinterOrientation	*poPortrait* oder *poLandscape* geben an, ob das Papier längs oder quer bedruckt wird.
PageHeight: Integer	Höhe der Seite in logischen Einheiten (Punkten).
PageNumber: Integer	zählt beginnend bei 1 die Seitenzahlen, wird durch *NewPage* erhöht.
PageWidth: Integer	Breite der Seite in logischen Einheiten (Punkten).
Printers: TString	Liste aller im System installierten Druckertreiber (die in der Datei `/etc/printcap` aufgelistet sind).
Printing: Boolean	*True*, wenn gerade ein Druckauftrag erstellt wird (wenn weder *EndDoc*, noch *Abort* aufgerufen wurde).
Title: String	Titel, den der Druckertreiber für dieses Dokument anzeigen soll.
XDPI: Integer	dpi-Auflösung des Druckers in X-Richtung (normalerweise 72, da PostScript verwendet wird)
YDPI: Integer	dpi-Auflösung in Y-Richtung, entsprechend *XDPI*

erwendung der TPrinter-Informationen

Ein Beispiel für die nützliche Anwendung der Properties von *TPrinter*, die Information ber den Drucker liefern, ist der TreeDesigner-Menüpunkt ANSICHT | SEITENANZEIGE, durch den die standardmäßig 3000 mal 3000 Einheiten große Zeichenfläche an die Ausmaße der Druckseite angepasst werden soll (10 virtuelle Einheiten pro Millimeter). Der TreeDesigner braucht dazu also die Seitenausmaße in Zentimetern, aber *TPrinter* gibt diese in Punkten (1/72 Zoll) an. Die folgende Funktion rechnet die Punkt-Werte in Millimeter um:

```
function PointsToMM(Value: Integer): Integer;
begin
  Result := trunc(Value * (25.4/72));
end;
```

Auf diese Weise kann nun die Größe der Zeichenfläche eingestellt werden, wobei im reeDesigner maximale Ausmaße von 3000 Einheiten in beiden Richtungen gelten[3]:

```
NeededWidth := 10 * PointsToMM(Printer.PageWidth);
NeededHeight := 10 * PointsToMM(Printer.PageHeight);
if NeededHeight<3000 then
  Document.Height:=NeededHeight;
if NeededWidth<3000 then
  Document.Width:=NeededWidth;
```

Drucken in Millimetern *R112*

Abschließend soll noch die Methode von *TVirtualCanvas* gezeigt werden, die den von Windows bekannten Abbildungsmodus *MM_LOMETRIC* simuliert, in dem eine Einheit dem Zehntel eines Millimeters entspricht. Sie besteht aus zwei Teilen:

- Zunächst wird die Auflösung des Ausgabeobjekts bestimmt. Da *TVirtualCanvas* unabhängig von einem Drucker und erst recht unabhängig vom *Printer*-Objekt sein soll, muss auf dessen Properties *XDPI* und *YDPI* verzichtet werden. Qt bietet jedoch eine gute alternative Informationsquelle: Während der Initialisierung des *TCanvas*-Objekts mit der *Start*-Methode wird das verwendete *QPainter*-Objekt mit einem *QPaintDevice*-Objekt verbunden, das ein *QPaintDeviceMetrics*-Objekt zu liefern imstande ist, in welchem sich die gewünschten Angaben über die Auflösung befinden. Diese verschiedenen Objekte und schließlich die gesuchte Auflösungen werden im unten abgedruckten Code über eine selbst erklärende Kette von Qt-Funktionsaufrufen ermittelt.

Diese Beschränkung ist unter Windows 9x notwendig und könnte unter Linux theoretisch nach oben korrigiert werden.

- Im zweiten Teil werden Window und Viewport eingestellt, wobei es wieder auf das *Verhältnis* der beiden ankommt. Wenn wir beispielsweise *XDPI* Einheiten auf dem Ausgabegerät nehmen, dann sind diese genau ein Zoll lang. Dieses Zoll entspricht 2,54 Zentimetern, also 254 Einheiten im *mmLoMetric*-Modus. Damit muss das Verhältnis von Viewport zu Window auf *XDPI:254* eingestellt werden.

```
procedure TVirtualCanvas.SetMapMode(const Value: TMapModes);
var
  QPaintDevice: QPaintDeviceH;
  QPaintDeviceMetrics: QPaintDeviceMetricsH;
  xDpi, yDpi: Integer;
begin
  if not Assigned(FVCLCanvas) then
    raise ENoVCLCanvas.Create(NoVCLCanvasError);
  FMapMode := Value;
  if FMapMode = mmLoMetric then begin
    QPaintDevice := QPainter_device(FVCLCanvas.Handle);
    QPaintDeviceMetrics := QPaintDeviceMetrics_create(QPaintDevice);
    try
      xDpi := QPaintDeviceMetrics_logicalDpiX(QPaintDeviceMetrics);
      yDpi := QPaintDeviceMetrics_logicalDpiY(QPaintDeviceMetrics);
    finally
      QPaintDeviceMetrics_destroy(QPaintDeviceMetrics);
    end;
    FViewPortExt.cx := xDPI;
    FWindowExt.cx := 254;
    FViewPortExt.cy := yDPI;
    FWindowExt.cy := -254;   // Y-Achse umgedreht (Windows-kompatibel!)
  end else
    ...(Einstellung der anderen Modi siehe CD-ROM)...
end;
```

Und da wir uns hier in der Klasse *TVirtualCanvas* finden, werden Window und Viewport nicht über *SetWindowExt* und *SetViewportExt* eingestellt, sondern in den *TVirtualCanvas*-eigenen privaten Variablen *FViewportExt* und *FWindowExt* gespeichert.

Wenn Sie also ohne *TVirtualCanvas* ein ebensolches Koordinatensystem verwenden wollen, können Sie auf die gleiche Weise die Werte *xDPI* und *yDPI* bestimmen und dann abweichend vom obigen Listing die Verhältnisse mit *QPainter_setWindow* und *QPainter_setViewport* setzen.

Hinweise: Aufgrund des Verzichts auf das *Printer*-Objekt funktioniert die obige Methode auch bei der Ausgabe auf dem Bildschirm, allerdings nimmt das System für diesen die Standardwerte von 72 dpi an, was für moderne Bildschirme eine Unterschätzung darstellt.

Außerdem sei noch einmal erwähnt, dass die »Druckerauflösung« unter Linux normalerweise mit 72 dpi angegeben wird, unabhängig von der Auflösung, in der der tatsächliche Drucker nach Umwandlung der PostScript-Ausgabe in sein natives Format druckt. Im Modus *mmLoMetric* muss *TVirtualCanvas* ca. 3,5 virtuelle 0,1-Millimeter-Einheiten in eine PostScript-Einheit umrechnen – bei dieser Umrechnung geht natürlich die Genauigkeit der Eingabekoordinaten verloren.

6.7 MDI-Anwendungen

Anwendungen, die dem Benutzer mehrere Dokumente gleichzeitig zur Bearbeitung anbieten wollen, finden von Qt und CLX Unterstützung durch die MDI-Schnittstelle. Der MDI-Standard sorgt für eine Vereinheitlichung in der Handhabung mehrerer Dokumente und erspart es dem Benutzer auf diese Weise, diese Handhabung in jeder Anwendung aufs neue erlernen zu müssen.

6.7.1 MDI im Überblick

Das wichtigste und eigentlich schon selbstverständliche Merkmal einer MDI-Anwendung ist, dass jedes Dokument in einem eigenen Fenster, dem *Dokumentfenster* oder *MDI-Kindfenster*, dargestellt wird. Diese Dokumentfenster befinden sich innerhalb des Hauptfensters, können sich beliebig überlappen und sind auch ansonsten in ähnlicher Weise wie die Hauptfenster auf dem Desktop zu bedienen.

Aufbau einer MDI-Anwendung *R63*

Um eine Kylix-Anwendung zu einer funktionsfähigen MDI-Anwendung zu machen, sind die folgenden drei Schritte erforderlich:

- Entwerfen Sie ein MDI-Hauptformular, dessen *FormStyle*-Property Sie auf *fsMDIForm* einstellen. Normalerweise wird dieses Formular automatisch als Hauptfenster der Anwendung erzeugt.
- Entwerfen Sie ein Formular für MDI-Kindfenster mit dem *FormStyle* von *fsMDIChild*. Sie können auch mehrere Typen von Kindfenstern in mehreren Formularen definieren. Normalerweise wird die automatische Erzeugung der Kindformulare in den Projektoptionen abgeschaltet.
- Statt dieser automatischen Erzeugung sollten Sie einen Menüpunkt wie DATEI | NEU zur Verfügung stellen, über den der Anwender zur Laufzeit nicht nur ein Kindfenster, sondern beliebig viele erstellen kann. In der zu diesem Menüpunkt gehörenden Methode erzeugen Sie dynamisch ein neues dieser MDI-Kindfenster (Kapitel 6.7.4).

Die folgenden Schritte sind ebenfalls empfehlenswert, aber optional – teilweise sind sie besonders einfach durchzuführen:

- Geben Sie sowohl dem Hauptformular als auch dem Kindformular ein eigenes Menü (im Property *Menu*). Die CLX verbindet diese Menüs zur Laufzeit automatisch zu einem einzigen Menü (Kapitel 6.7.3).
- Um die gewohnten Funktionen zum Anordnen der Fenster zur Verfügung zu stellen, legen Sie ein *Fenster*-Menü an, dessen *OnClick*-Methoden aus ein oder zwei Zeilen bestehen, oder Sie verwenden eine Aktionsliste mit Standardaktionen und müssen keine einzige Zeile schreiben.
- Damit der Benutzer die MDI-Kindfenster auch wieder schließen kann, bearbeiten Sie das *OnClose*-Ereignis (per Voreinstellung wird das Fenster bei einem solchen Ereignis nur zum Symbol verkleinert).
- Nachdem der Benutzer die Anwendung gestartet hat, sollte er die Möglichkeit haben, sofort mit der Arbeit zu beginnen. Die Anwendung sollte also automatisch ein neues Dokumentfenster öffnen. Die letztgenannten drei Punkte werden in Kapitel 6.7.4 behandelt.
- Die Werkzeugleiste oder Mauspalette können Sie lokal für die Dokumentfenster oder global für die gesamte Anwendung anlegen (Kapitel 6.7.5 und 6.7.6).
- Der Rahmen der MDI-Kindfenster wird – im Gegensatz zum Rahmen der Hauptfenster – nicht vom verwendeten Fenstermanager, sondern von Qt gezeichnet, und zwar immer mit einem Icon in der linken oberen Fensterecke – bzw. eine wenig interessante leere Fläche, wenn Sie kein Icon im *Icon*-Property des MDI-Kindformulars angegeben haben (Sie können über den Objektinspektor eine Bitmapdatei hineinladen).

Zwei gegensätzliche Ziele

Bei der Entwicklung einer MDI-Anwendung gibt es zwei gegensätzliche Ziele: Einerseits soll das Dokumentformular weitgehend unabhängig vom Hauptformular sein, weil die Programmstruktur sonst sehr unübersichtlich werden könnte. Andererseits würde der Bildschirmaufbau unübersichtlich werden, wenn jedes MDI-Kindfenster über ein eigenes Menü und eigene Schalter- und Statusleisten verfügen würde. Wenn Sie viele derartige MDI-Kindfenster nebeneinander anordnen würden, würde die kleine, jedem Fenster noch verbleibende Fläche zum größten Teil durch Menü, Mauspalette und Statuszeile gefüllt werden.

Die CLX bietet für die Menüs eine Lösung an, die beiden Zielen entspricht: Sie entwerfen das Dokumentformular als abgeschlossene, unabhängige Einheit mitsamt einem eigenen Menü, und zur Laufzeit blendet die CLX dieses Menü in das Menü des Hauptfensters ein.

hnliches geschieht mit Mauspaletten und Statuszeilen nicht automatisch, jedoch finden Sie in Kapitel 6.7.5 eine Beschreibung, wie Sie auch einen solchen Fensterbestandteil des Dokumentfensters im Hauptfenster einblenden können.

6.7.2 MDI- und SDI-Versionen des TreeDesigners

Die SDI-Version des TreeDesigners (Projektname: *TreeDesignerSDI*) unterscheidet sich von der MDI-Version dadurch, dass alle Leisten, die in der MDI-Version im Hauptfenster erscheinen, im Dokumentfenster enthalten sind. In der SDI-Version *ist* das Dokumentformular das Hauptfenster.

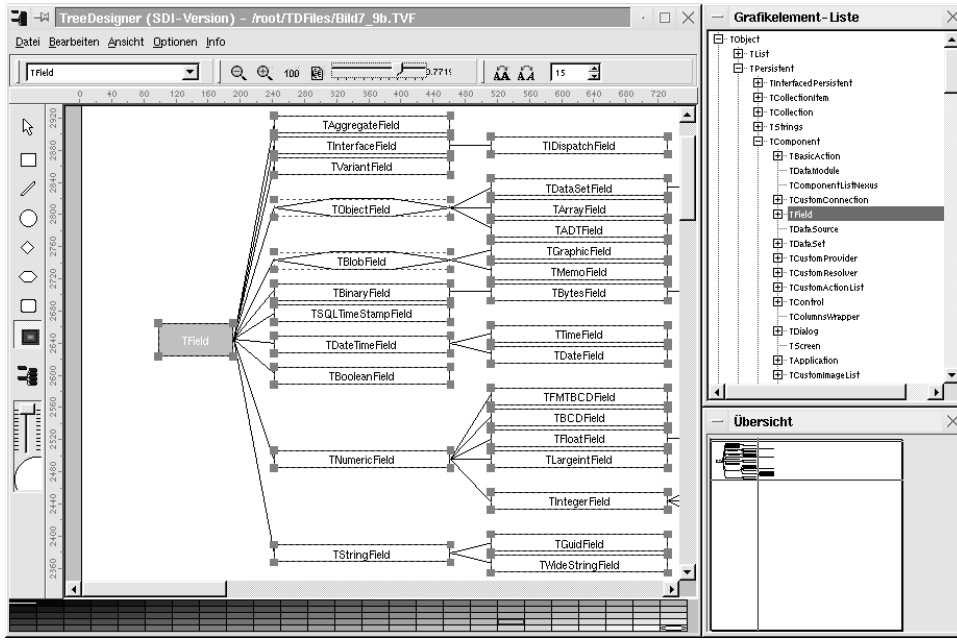


Abbildung 6.11: In der SDI-Version behält das Dokumentfenster seine Toolbars (oben) und erhält eine eigene erkzeugleiste (links) und eine Farbpalette (unten). Die vom MDI-Hauptformular bekannte Andockmöglichkeit für die Hilfsfenster (rechts) wurde in der SDI-Version nicht übernommen.

In Absehung davon, dass Sie die MDI-Version des TreeDesigners schon in den bisherigen Kapiteln verwenden konnten, nehmen wir hier an, dass wir bisher nur die SDI- ersion vorliegen haben (Abbildung 6.11), die wir hier zur MDI-Anwendung ausbauen. Wir wandeln also das Dokumentformular in ein MDI-Kindfenster um, indem wir das Property *TDocumentForm.FormStyle* auf *fsMDIChild* ändern. Dadurch entsteht auch gleich der Bedarf für ein »neues« Hauptformular.

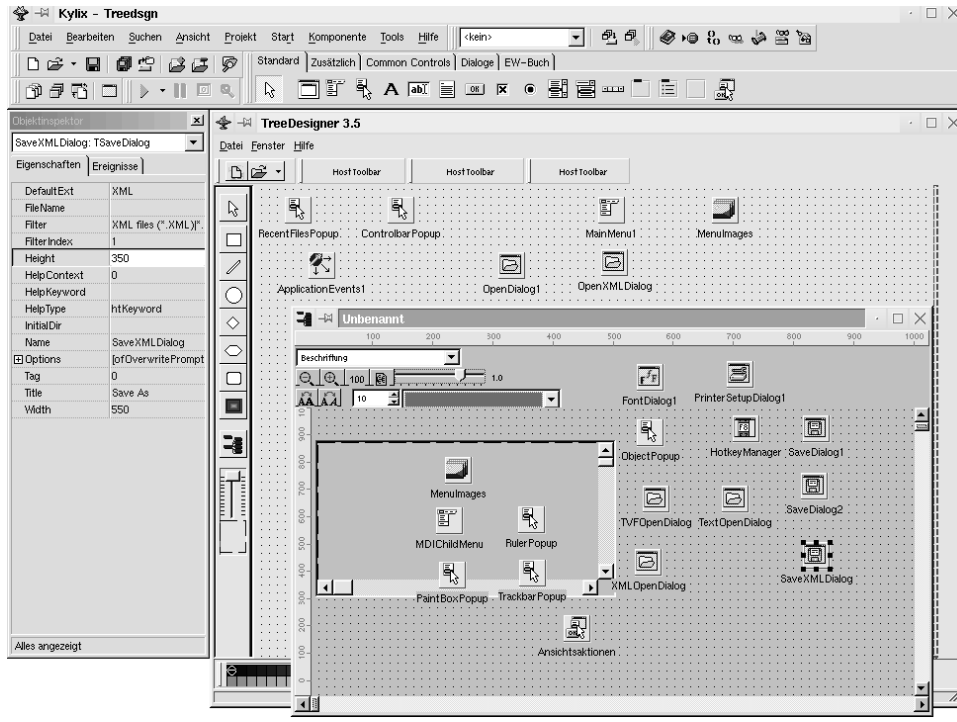


Abbildung 6.12: Die Formulare für Haupt- und Dokumentfenster der MDI-Version zur Entwurfszeit

Abbildung 6.12 zeigt Haupt- und Dokumentformulare im Entwurfsmodus. Das Hauptformular des TreeDesigners heißt *MainForm* und ist in der Unit *MdiForm* definiert. Die wichtigste Property-Einstellung für das neue Hauptfenster findet wieder in *FormStyle* statt. Hier bewirken Sie durch Auswahl von *fsMdiForm*, dass beispielsweise neu erzeugte MDI-Kindfenster automatisch in dieses Formular eingefügt werden und dass die Menüs verschmolzen werden, wie im nächsten Kapitel beschrieben. Im TreeDesigner wird außerdem das Property *Caption* geändert, damit das Hauptfenster einen angemessenen Titel erhält.

Alle weiteren Änderungen am Hauptfenster finden nicht in den Properties statt, sondern über neu eingefügte Komponenten:

- eine *TOpenDialog*-Komponente, die den *Datei-öffnen*-Dialog zum Öffnen eines Dokuments bereitstellt,
- eine *TMainMenu*-Komponente, in der die globalen Menüpunkte DATEI, FENSTER und HILFE untergebracht werden. Zur Laufzeit wird dieses Hauptmenü nur angezeigt, solange keine MDI-Kindfenster geöffnet sind. Wichtig ist vor allem, dass in dieser Situation ein Menüpunkt zum Öffnen einer neuen Datei zugänglich ist. Dies

ist durch das globale DATEI-Menü gewährleistet, welches einzig aus den Punkten ÖFFNEN... und NEU besteht und nach Öffnen eines Dokumentfensters durch ein ausführlicheres Menü ersetzt wird.

- eine *TImageList* für die im Hauptmenü anzuzeigenden Symbole. Das Hauptformular kann in diesem Fall *nicht* die Bilderliste des Dokumentformulars mitbenutzen, weil das Dokumentformular dynamisch erzeugt wird, also gar nicht immer ein Exemplar davon existiert. Daher wurde die Bilderliste des Dokumentformulars einfach über die Zwischenablage in das Hauptformular kopiert. Sollte die Bilderliste des Dokumentformulars einmal geändert werden, ist es unter Umständen erforderlich, sie erneut in das Hauptformular zu kopieren. (Möglicherweise werden Sie in ähnlichen Situationen die Benutzung einer gemeinsamen Bilderliste vorziehen. Dies wäre natürlich auch im TreeDesigner möglich, wenn die Liste z.B. in einem Datenmodul untergebracht wäre oder das Dokumentformular die Bilderliste des Hauptformulars mitverwenden würde.)
- eine neue *TControlBar*-Komponente, die mit *alTop* am oberen Rand ausgerichtet wird.
- Aus dem SDI-Formular werden übernommen: die Werkzeugleiste (das mit *alLeft* ausgerichtete Panel) und die *TControlBar*-Komponente vom unteren Fensterrand mitsamt der darin enthaltenen Farbpalette.
- Aus der oberen *TControlBar*-Komponente des SDI-Formulars werden die kleine Toolbar mit den Schaltern zum Öffnen von Dateien sowie das zugehörige *RecentFilesPopup* für das Menü der zurückliegenden Dateien übernommen. *Die vier anderen Toolbars bleiben im Dokumentformular*, werden aber mit *Visible:=False* unsichtbar gemacht. Die ControlBar, in der sich diese vier Toolbars befinden, kann entweder auch unsichtbar gemacht oder ganz entfernt werden.
- Als Vorbereitung für die Einblendung der Kind-Toolbars werden in die obere ControlBar des Hauptformulars vier Platzhalter-Panels eingebaut, sie zeigen sich zur Entwurfszeit wie in Abbildung 6.12 durch vier schmale ControlBar-Bänder.

Weitere Umstellungsmaßnahmen

Das folgende Kapitel wird sich zunächst um das Verschmelzen des globalen Hauptmenüs mit den Hauptmenüs der Dokumentfenster kümmern. In Kapitel 6.7.4 wird es um die Verwaltung der Kindfenster gehen, zum Schluss bleibt noch das Symbolleisten-Management. Die obige Aufzählung hat hier schon einen wichtigen Unterschied deutlich gemacht:

- Vier Toolbars des SDI-Formulars werden im Hauptfenster eingeblendet. Für dieses Einblenden entwickelt Kapitel 6.7.5 eine Schnittstelle, über die Sie ähnliche Einblendungsvorgänge leicht in eigene Anwendungen übernehmen können.

- Die Farbpalette, die Toolbar zum Öffnen von Dateien und die Werkzeugleiste zur Auswahl der Grafikform werden schon zur Entwurfszeit in das Hauptformular verlagert. Dies hat zur Folge, dass die darin enthaltenen Komponenten nicht mehr so leicht in der Unit des Dokumentformulars angesprochen werden können. Um dieses Problem wird sich Kapitel 6.7.6 kümmern.

6.7.3 Verschmelzen von Menüs

Die CLX bietet eine einfache Möglichkeit an, sowohl für das MDI-Hauptformular als auch für das Dokumentformular jeweils ein eigenes Hauptmenü im Property *Menu* anzugeben. Wenn ein Dokumentfenster aktiviert wird, blendet die CLX dessen Menü in das Menü des Hauptfensters ein. Jedes Mal, wenn ein anderes MDI-Kindfenster aktiviert wird, konstruiert die CLX das Hauptmenü neu, so dass darin immer die Menüpunkte des gerade aktiven Kindes enthalten sind. Es können also auch Dokumentfenster von verschiedenen Typen gleichzeitig im Hauptfenster enthalten sein. Ist kein MDI-Kindfenster vorhanden, enthält die Menüzeile des Hauptfensters nur das Original-Menü des Hauptformulars, das Sie in dessen *Menu*-Property angegeben haben.

Beim Entwurf der Dokumentformulare erweist es sich als sehr günstig, dass Sie das Menü lokal innerhalb derselben entwerfen können, denn dadurch werden die einzelnen Menübefehle zu Teilen des Dokumentfensters und Sie können die zugehörigen Ereignismethoden im Objektinspektor direkt mit Methoden dieses Fensters in Verbindung bringen. Dies wäre nicht möglich, wenn die Menüpunkte dem Hauptformular gehören würden, denn dann könnten Sie sie auch nur mit Methoden des Hauptformulars verknüpfen.

Sollten Sie das Dokumentfenster einmal außerhalb der MDI-Anwendung als eigenständiges Fenster verwenden, wird das Menü automatisch im Dokumentfenster angezeigt. Sie brauchen also oft kaum mehr als das Property *FormStyle* zu ändern, um ein MDI-Dokumentfenster zu einem eigenständigen Fenster, beispielsweise zum Hauptfenster einer SDI-Anwendung, zu machen.

Der GroupIndex bestimmt den Menüinhalt

Eine wichtige Regel für das (kombinierte) Menü, das zur Laufzeit im Hauptfenster angezeigt wird, ist, dass sich in diesem Menü immer alle Menüpunkte des MDI-Kindes befinden. Wenn Punkte ersetzt werden, dann sind das immer die Punkte des Hauptformular-Menüs. Die kleinste Einheit bei der Verschmelzung von Hauptfenster- und Dokumentmenüs sind die Punkte der obersten Menüebene (also der Menüzeile). Innerhalb der Popup-Menüs findet kein Austausch statt, ein Popup-Menü gehört also entweder ganz zum Haupt- oder ganz zum Dokumentfenster.

Maßgeblich bei der Kombination der Menüs ist hauptsächlich das Property *GroupIndex* der Menüpunkte der obersten Ebene. Per Voreinstellung ist dieses bei jedem Menüpunkt mit dem Wert 0 belegt, was zur Folge hat, dass das Menü des Hauptfensters komplett durch das Menü des Kindfensters ersetzt wird.

Allgemein wird ein Menü des Hauptmenüs immer dann ersetzt, wenn es im Kindmenü einen oder mehrere Punkte gibt, die denselben Gruppenindex besitzen. Hat ein Kindmenüpunkt einen Index, der im Menü des Hauptfensters nicht enthalten ist, so wird dieser Punkt zusätzlich in das Hauptmenü eingefügt, und zwar so, dass die *GroupIndex*-Werte im kombinierten Menü von links nach rechts in aufsteigender Reihenfolge angeordnet sind. Dabei setzt die CLX voraus, dass sich diese Werte in den einzelnen, noch nicht kombinierten Menüs bereits in aufsteigender Reihenfolge befinden.

erschmelzungs-Beispiel

Im TreeDesigner enthält das Menü des Hauptformulars beispielsweise die Punkte *Datei* (*GroupIndex=1*), *Fenster* (*GroupIndex=9*) und *Hilfe* (*GroupIndex=10*). Die Menüs des Dokumentformulars haben alle einen Gruppenindex von 1. Zur Laufzeit bedeutet das, dass das Dateimenü des Hauptfensters ersetzt wird, da das Dokumentformular Menüpunkte mit demselben Gruppenindex besitzt. Die Menüs *Fenster* und *Hilfe* werden nicht ersetzt, sondern nach den Menüpunkten mit dem Gruppenindex 1 eingefügt. Letzter Menüpunkt bleibt das Hilfemenü mit dem größten Gruppenindex.

Da die Punkte aus dem Dateimenü des Hauptfensters durch das neue Dateimenü ersetzt werden, muss dieses die Punkte des Hauptfenstermenüs wiederholen, wenn diese weiter anwählbar sein sollen. (In diesem Beispiel handelt es sich um die Punkte FFNEN... und NEU.) Das hat jedoch zur Folge, dass das Dokumentformular auch die zugehörigen *OnClick*-Events bearbeiten muss, die schon vom Hauptformular bearbeitet wurden. Normalerweise ruft es dazu einfach die entsprechende Event-Bearbeitungsmethoden des Hauptfensters auf oder verknüpft die *OnClick*-Ereignisse direkt mit diesen (siehe Kapitel 6.7.4).

AutoMerge

Mit dem *TMainMenu*-Property *AutoMerge* können Sie auch bei Nicht-MDI-Anwendungen erreichen, dass das Menü eines untergeordneten Fensters in das Menü des Hauptfensters eingeblendet wird. Standardmäßig ist dieses Property *False*, was bedeutet, dass die Verschmelzung unterbleibt. Die Verschmelzung der Menüs in MDI-Anwendungen findet jedoch auch dann statt, wenn Sie das *AutoMerge*-Property der Dokumentmenüs bei seinem voreingestellten Wert *False* belassen.

Hinweis: Hat *AutoMerge* den Wert *True*, dann wird das Menü nicht in dem Fenster eingeblendet, dem es eigentlich gehört, selbst dann nicht, wenn es gar kein übergeordnetes Fenster gibt, in dem es eine Ersatzunterkunft finden könnte. Das bedeutet, dass Sie zur Laufzeit gar kein Menü erhalten, wenn Sie *AutoMerge* auch beim Hauptformular auf *True* setzen.

6.7.4 Verwaltung der MDI-Kindfenster

Der Entwurf eines Haupt- und eines Kindformulars führt alleine noch nicht zu einer funktionierenden MDI-Anwendung. Das Mindeste, was noch hinzukommen muss, ist eine Möglichkeit, zur Laufzeit neue Kindfenster zu öffnen. Aber auch dann bleiben noch einige kleinere Aufgaben zu erledigen, bis der gewohnte MDI-Komfort erreicht wird. Bevor wir uns diesen Einzelheiten zuwenden, sehen wir uns die Art an, in der die Kindfenster angesprochen werden. (Die MDI-Dokumentfenster werden hier so lange allgemein als Kindfenster bezeichnet, wie es sich nicht um spezielle Fenster handelt, die ein Dokument darstellen.)

Die Kindfenster ansprechen

Drei Properties des Hauptformulars erleichtern Ihnen das direkte Ansprechen der MDI-Kindfenster zur Programmlaufzeit. Zunächst speichert *ActiveMDIChild* das gerade aktive Kindfenster. Wenn Sie auch auf die anderen Kindfenster zugreifen wollen, so finden Sie in *MDIChildCount* die Zahl dieser Fenster und können über das Property *MDIChildren* auf jedes einzelne davon zugreifen (siehe auch *TForm*-Übersicht in Kapitel 3.4.3).

Die globale Formularvariable, die Kylix für jede Formularklasse anlegt, wird für die Kindfenster nicht benötigt. Normalerweise dient sie dazu, eine einzelne Fensterinstanz zu speichern. Wenn Kylix ein Formular automatisch erzeugt (durch den *CreateForm*-Aufruf, siehe Kapitel 1.5.3), so verwendet es dazu diese globale Variable. Da der Benutzer jedoch zur Programmlaufzeit beliebig viele MDI-Kindfenster öffnen kann, ist diese Variable völlig unzureichend. Selbst wenn sie von Kylix dazu verwendet wird, beim Programmstart automatisch ein Kindfenster zu öffnen, sollten Sie sie nach dem Programmstart nicht mehr verwenden, denn der Benutzer kann üblicherweise jedes Dokumentfenster jederzeit schließen und dadurch die globale Variable ungültig machen.

Hinweis zur Portierung von der VCL: Trotz großer Ähnlichkeit in der Programmierschnittstelle wird an manchen Details doch sichtbar, dass die MDI-Funktionalität intern völlig anders realisiert ist als in der VCL. Das Property *ActiveMDIChild* etwa, welches in der CLX nur eine dünne Hülle um einen Funktionsaufruf von Qt darstellt, ist in der CLX nicht zu den gleichen Zeiten gültig wie in der VCL. So wurde bei der TreeDesigner-Portierung im Ereignis *TScreen.OnActiveFormChange* festgestellt, dass *ActiveMDIChild* noch nicht auf *nil* gesetzt war, obwohl das letzte MDI-Kindfenster gerade geschlossen worden war. In diesem Fall erwies sich die Variable *Screen.ActiveCustomForm* als zuverlässiger, die dann auch statt *ActiveMDIChild* abgefragt wird (siehe *MdiForm.pas*, *TMainForm.ScreenActiveFormChange*).

Kindfenster erzeugen *R36*

Die Erzeugung eines Kindfensters läuft nach dem gleichen Muster ab wie die dynamische Erzeugung beliebiger anderer Objekte, und zwar über einen Konstruktoraufruf. Die Erzeugung eines Fensters sollte in einer eigenen, von Ereignissen unabhängigen Methode gekapselt werden. Im TreeDesigner heißt sie *NewWindow* und erwartet als Parameter ein bereits erzeugtes Dokument-Objekt (zur Unabhängigkeit zwischen Dokument und Fenster siehe Kapitel 6.3.1):

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  { vereinfachter Create-Aufruf (Erweiterung in Kapitel 6.7.5): }
  Window := TDocumentForm.Create(self, Document);
  Window.Visible := True;
  Window.Show;
  Result := Window;
end;
```

Diese Methode kann dann zu mehreren Anlässen aufgerufen werden:

- als Antwort auf den Menübefehl DATEI | NEU,
- wenn die Anwendung gestartet wird und
- beim Menübefehl DATEI | ÖFFNEN: Der TreeDesigner erzeugt bei Anwahl dieses Menüpunkts mit *NewWindow* zuerst ein neues leeres Kindfenster und lädt dann eine Datei in dieses hinein.

Die Methoden für die beiden Punkte des Datei-Menüs sind schnell programmiert, da sie nur aus Aufrufen der oben gezeigten Methode *NewWindow* bzw. aus Methoden, die n anderen Kapiteln besprochen werden, bestehen:

```
{ OnClick-Ereignis von Datei|Öffnen }
procedure TMainForm.MDIOpenClick(Sender: TObject);
var
```

```
  NewChild: TDocumentForm;
  Document: TGraphicDoc;
begin
  if OpenDialog1.Execute then begin
    Document := AddDoc(OpenDialog1.FileName); { sucht das Dokument unter
       den bereits geöffneten bzw. öffnet ein neues, siehe Kapitel 6.3.1 }
    NewChild := NewWindow(Document);
  end;
end;

{ OnClick-Ereignis von Datei|Neu }
procedure TMainForm.MDINewClick(Sender: TObject);
begin
  NewWindow(AddDoc(''));
end;
```

Vorsicht: Wichtig ist, dass Sie in einer Methode wie *NewWindow* nicht das *Parent*-Property des neuen Kindfensters setzen. Dies ist nur dann erforderlich, wenn Sie ein Nicht-MDI-Kind als Unterfenster in ein Hauptfenster einfügen wollen. Für MDI-Kindfenster bleibt das *Parent*-Property *nil*.

Kindfenster bei Programmstart öffnen *R41*

Etwas schwieriger ist es, schon beim Programmstart ein neues Dokumentfenster anzulegen. Im Zusammenhang mit dem *OnCreate*-Ereignis des Hauptfensters ist dies beispielsweise nicht möglich, weil das Hauptfenster zu diesem Zeitpunkt noch nicht vollständig initialisiert ist. Zumindest im TreeDesigner darf dieses erste Kindfenster auch nicht automatisch von der CLX erzeugt werden, da diese nicht den neuen *Create*-Konstruktor von *TDocumentForm*, der in Kapitel 6.7.5 eingeführt wird, aufrufen würde.

Die wohl einfachste noch bleibende Möglichkeit ist, die Erzeugung eines Fensters durch *NewWindow* per Hand in die Projektdatei einzufügen, und zwar zwischen den letzten von Kylix automatisch erzeugten *CreateForm*-Aufruf und den ebenfalls automatisch erzeugten Aufruf von *Application.Run* (diese Projektdatei erhalten Sie am schnellsten über den Menüpunkt PROJEKT | QUELLTEXT ANZEIGEN:

```
{ Hauptprogramm der Projektdatei }
begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  ... Erzeugung der anderen Formulare ...
  Application.CreateForm(TOverviewForm, OverviewForm);
  MainForm.BeforeRun; { <<< Manuell eingefügt }
  Application.Run;
end.
```

Da der TreeDesigner neben dem Öffnen eines ersten Fensters auch noch andere Dinge zu diesem Zeitpunkt zu erledigen hat, wurde all dies in der Methode *BeforeRun* verpackt. Bei weiteren Änderungen braucht so statt des Projektdatei-Quelltextes nur noch diese Methode angepasst zu werden:

```
procedure TMainForm.BeforeRun;
begin
  { automatisch eine Datei laden, falls das Programm
    mit Befehlszeilenparametern aufgerufen wurde: }
  if (ParamStr(1)<>'') and (ParamStr(1)[1]<>'-')
  {$ifdef Windows} and (ParamStr(1)[1]<>'/')  {$endif}
  then
    NewWindow(AddDoc(ParamStr(1)))
  else
    NewWindow(AddDoc('')); // neues leeres Dokument
  // hier folgt noch die Wiederherstellung des Docking-Layouts,
  // siehe Kapitel 6.8.2
end;
```

Die *if*-Anweisung untersucht, ob der Anwendung ein Startparameter übergeben wurde (die Startparameter geben Sie in der Kylix-IDE unter START | PARAMETER ... an) und ob dieser nicht durch ein vorangehendes »-« als Option gekennzeichnet ist. Alles, was keine Option ist, wird als Dateiname interpretiert. Die obige Methode versucht dann, diese Datei mit *LoadFromFile* in das neue Fenster zu laden.

Hinweis: Während Windows auch das Zeichen »/« als Kennzeichnung für Parameter verwendet wird, verbietet sich dies unter Linux, da sonst Pfadangaben als Option verwechselt werden können. Der TreeDesigner erwartet übrigens in der aktuellen Linux-Version gar keine Kommandozeilenoptionen, während in der Windows-Version spezielle COM-Server-Parameter denkbar sind, die zwar von der VCL automatisch ausgewertet werden, aber in obiger Methode nicht einem Dateinamen verwechselt werden würden, wenn die Abfrage des Options-Kennzeichens fehlen würde.

Kindfenster schließen

Das Schließen der Kindfenster läuft so ab wie bei normalen Formularen. So fragt die CLX vor dem Schließen des Fensters mit dem *OnCloseQuery*-Event um Erlaubnis. Die hierfür in Kapitel 6.2.4 geschriebene Methode bleibt daher weiterhin gültig. Wenn die gesamte MDI-Anwendung geschlossen werden soll, ruft die CLX für jedes Kindfenster die *OnQueryClose*-Methode auf und schließt die Anwendung nur dann, wenn jede dieser Methoden die Schließung erlaubt.

Damit der Benutzer die Dokumentfenster auch schließen kann, muss darüber hinaus das *OnClose*-Event wie folgt bearbeitet werden:

```
procedure TDocumentForm.
   FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Ohne Zuweisung von *caFree* würde das Formular lediglich minimiert werden (zu *caFree* siehe auch in Kapitel 3.4.4 den Abschnitt *Eingriff in das Schließverhalten*).

Das Fenstermenü

Eine besonders einfache Aufgabe stellt das Implementieren des Fenstermenüs dar, wenn auch die Unterstützung der CLX (noch) nicht so weit geht wie die der VCL. Zur Implementierung der typischen Fenster-Menüpunkte NEBENEINANDER und ÜBEREINANDER können Sie entweder einfache Methodenaufrufe schreiben oder eine Aktionsliste mit Standardaktionen verwenden. Als Beispiel für die Methodenaufrufe folgt die Implementierung des Menüpunktes FENSTER | NEBENEINANDER im TreeDesigner:

```
procedure TMainForm.HorizontalTile1Click(Sender: TObject);
begin
  Tile;
end;
```

Hinweis: Der TreeDesigner unterscheidet bei »Nebeneinander« zwischen horizontaler und vertikaler Aufteilung (in der VCL über das Property *TileMode* einstellbar). Diese Unterscheidung sowie auch der unter Windows übliche Fenstermenüpunkt ICONS ANORDNEN werden jedoch von der Qt-MDI-Implementation nicht unterstützt.

Für die Implementierung des Punktes ÜBEREINANDER müssten Sie nun entsprechend die Prozedur *Cascade* aufrufen. Wenn Sie jedoch Aktionslisten (*TActionList*-Komponenten, siehe Kapitel 4.6.4) verwenden, brauchen Sie für beide Menüpunkte keinen Code zu schreiben:

- Öffnen Sie den Aktionslisteneditor zu Ihrer *TActionList*-Komponente, rufen Sie aus dem lokalen Menü NEUE STANDARD-AKTION... auf und fügen Sie die beiden Standardaktionen für MDI-Fenster ein. Danach genügt es bereits, die *Action*-Properties der beiden Menüpunkte mit diesen Aktionen zu verknüpfen.

Die Fensterliste R46

Was für ein komplettes Fenstermenü noch fehlt, ist die Liste der offenen Dokumente (bzw. Dokumentfenster), die üblicherweise hinter einer Trennungslinie an das bestehende Fenstermenü angehängt wird. Hierzu bietet die CLX nicht wie die VCL ein ein-

faches Property an, indem Sie einen Menüpunkt angeben können, der automatisch erweitert wird. Aber auch das manuelle Anhängen einer Liste aller offenen Fenster geht mit Hilfe der in Kapitel 4.6 besprochen Menü-Funktionen sehr schnell.

Im Folgenden bezeichnet *TLM_Windows* die *TMenuItem*-Komponente des Fenstermeüs im TreeDesigner-Hauptfenster (»TLM« steht dabei für »Top Level Menu«). *ItemBeforeWindowList* bezeichnet die *TMenuItem*-Komponente für die Trennlinie, die zur Entwurfszeit an das Ende dieses Menüs angefügt wurde. Ihr sollen die Menüpunkte ür die offenen Fenster folgen. Vor dem Hinzufügen der Menüpunkte werden jedoch eventuell früher erzeugte Menüpunkte wieder mit *Delete* entfernt:

```
procedure TMainForm.TLM_WindowsClick(Sender: TObject);
var
  i: Integer;
begin
  while TLM_Windows.Count >
        TLM_Windows.IndexOf(ItemBeforeWindowList)+1 do
    TLM_Windows.Delete(TLM_Windows.Count-1);
  for i := 0 to MdiChildCount - 1 do
    TLM_Windows.Add(
      NewItem(MdiChildren[i].Caption, 0, False, True,
              DoWindowItemClick, 0, ''));
end;
```

Beim Aufruf von *NewItem* werden alle neu erzeugten Menüpunkte mit der folgenden Methode *DoWindowItemClick* verknüpft. Sie ermittelt den Titel des gewählten Fensters aus dem *Caption*-Property des Menüpunkts und sucht dann die Liste der MDI-Kindfenster nach diesem Titel ab. Einer Besonderheit der MDI-Implementation von Qt zufolge wird dieses Fenster abschließend nicht mit der üblichen Methode *BringToFront* nach vorne gebracht, sondern mit *SetFocus*:

```
procedure TMainForm.DoWindowItemClick(Sender: TObject);
var
  i: Integer;
  SelectedWindowTitle: String;
  AmpPos: Integer;
begin
  if Sender is TMenuItem then begin
    SelectedWindowTitle := (Sender as TMenuItem).Caption;
    // automatisch von der CLX eingefügtes '&' - Zeichen
    // aus dem Menü-Text entfernen:
    AmpPos := pos('&', SelectedWindowTitle);
    if AmpPos > 0 then Delete(SelectedWindowTitle, AmpPos, 1);
    for i := 0 to MdiChildCount-1 do
      if MdiChildren[i].Caption = SelectedWindowTitle then
        MdiChildren[i].SetFocus;
  end;
end;
```

Hauptmenüpunkte im Kindfenster R37

Wenn ein Menüpunkt des Dokumentfensters einem globalen Befehl aus dem Menü des Hauptfensters entspricht, so sollte dieser auch vom Hauptfenster bearbeitet werden. So wurden in diesem Kapitel bereits die Methoden für die Hauptformular-Menüpunkte DATEI | NEU und DATEI | ÖFFNEN des TreeDesigners gezeigt, die im Menü des Dokumentfensters wiederholt werden müssen – aus den erwähnten Gründen des Menüverschmelzens.

Eine nahe liegende Möglichkeit besteht darin, im Dokumentformular für den doppelten Menüpunkt einfach eine neue Bearbeitungsmethode zu erzeugen, in der Sie dann die entsprechende Methode des Hauptformulars aufrufen:

```
procedure TDocumentForm.CmGlobalOpenClick(Sender: TObject);
begin
  MainForm.MDIOpenClick(Sender);
end;
```

Man kann sich eine solche Methode jedoch auch sparen, indem man das *OnClick*-Ereignis des Dokumentfenster-Menüpunktes zur Entwurfszeit unverknüpft lässt und es stattdessen zur Laufzeit direkt mit der entsprechenden Methode des Hauptformulars verknüpft (z. B. beim *OnCreate*-Event). Im TreeDesigner wird der zweite der doppelten Menüpunkte (mit dem Namen *CmGlobalNew*) zur Abwechslung auf diese Weise bearbeitet, wobei die Methode *TMainForm.MDINewClick* bereits am Anfang des Kapitels gezeigt wurde:

```
procedure TDocumentForm.FormCreate(Sender: TObject);
begin
  CmGlobalNew.OnClick := MainForm.MDINewClick;
  ... weitere Initialisierung ...
```

Für beide Lösungsmöglichkeiten müssen Sie die Unit des Hauptformulars in die Unit des MDI-Kindfensters einbinden, um auf die globale Variable des MDI-Hauptfensters zugreifen zu können (im Beispiel die Variable *MainForm*). Da die Unit des Hauptformulars bereits die Unit des Dokumentformulars einbindet, können Sie die umgekehrte Einbindung nur innerhalb des Implementationsteils der Dokumentformular-Unit vornehmen (siehe Kapitel 2.1.7):

```
implementation
uses
  MdiForm;
```

Eine dritte Möglichkeit kommt ohne diese Unit-Einbindung aus: Setzen Sie den *OnClick*-Methodenzeiger des Dokumentformulars vom MDI-Hauptfenster aus, beispielsweise direkt nach der Erzeugung des Dokumentfensters:

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  Window := TDocumentForm.Create(self, Document);
  Window.CmGlobalNew.OnClick := MDINewClick;
  ...
```

Hinweis: Die visuelle Formularverknüpfung (benötigt bei Datenmodulen, siehe Kapitel 7.2.1) ist leider nicht dazu geeignet, Ereignisse mit Methoden eines anderen Formulars zu verknüpfen, so dass Sie hier eine der beschriebenen nicht-visuellen Vorgehensweisen wählen müssen.

6.7.5 Dynamisches Werkzeugleisten-Management

Wie schon in Kapitel 6.7.1 unter *Zwei gegensätzliche Ziele* festgestellt, sollten Symbol- und Werkzeugleisten sich in einer MDI-Anwendung aus Gründen der Übersichtlichkeit auch dann im Hauptfenster befinden, wenn sie zur Manipulation des Dokuments dienen. Da die CLX die Symbolleisten des Dokumentfensters nicht automatisch im Hauptfenster einblendet wie die Menüs, werden wir diese Aufgabe hier selbst übernehmen.

orteile lokaler Werkzeugleisten

Ziel ist es also, die Leiste im Dokumentfenster entwerfen zu können und erst zur Laufzeit in das Hauptfenster zu verschieben. Für diese Vorgehensweise sprechen wichtige Vorteile:

- Sie erhält dem Dokumentformular die weitgehende Unabhängigkeit vom Hauptfenster, die es durch sein eigenes Hauptmenü gewonnen hat.
- Die Programmierung der Werkzeugleiste vereinfacht sich erheblich, denn die aufwändigeren, in Kapitel 6.7.6 beschriebenen Vorgänge entfallen, wenn die Leiste lokal ist.
- Da schließlich jedes Dokumentfenster seine eigene Kopie der Werkzeugleiste besitzt, bleiben alle darin vorgenommenen Einstellungen erhalten. Haben Sie beispielsweise ein Fenster mit dem Vergrößerungsfaktor 3 und eines mit dem Faktor 0,5, so können Sie beliebig zwischen beiden wechseln, und die Werkzeugleiste zeigt für jedes den richtigen Vergrößerungsfaktor an. Es scheint zwar so, als passe die Leiste die angezeigten Werte dem jeweils aktuellen Fenster an, tatsächlich handelt es sich jedoch um zwei verschiedene Leisten, die sich jeweils nicht verändern.

Man könnte es höchstens als kleinen Nachteil bezeichnen, dass die Werkzeugleiste bei jedem Wechsel des Dokumentfensters neu aufgebaut werden muss. Solange die Gra-

fikkarte dies jedoch schnell erledigt und das Programm ein Flackern größerer Teile des Bildschirms unterbindet, bekommt der Anwender eher den angenehmen Eindruck, dass das Programm besonders fleißig und kontextsensitiv ist.

Elternwechsel für die lokale Werkzeugleiste

Durch die Unterscheidung zwischen dem Besitzer einer Komponente und deren Elternfenster erzielt die CLX eine bemerkenswerte Flexibilität. Ausnahmen von der Regel »Besitzer=Elternfenster« kennen Sie bereits aus Kapitel 3 (ein Formular besitzt auch seine »Enkel-Komponenten«, die einem seiner Gruppenfenster oder Panels als Kindfenster untergeordnet sind).

Auch der folgende Fall ist denkbar: Komponenten, die sich im Besitz eines MDI-Kindfensters befinden, können zur Laufzeit das MDI-Hauptfenster als Elternfenster zugeordnet bekommen, ohne dadurch den Besitzer zu wechseln.

Die grundsätzliche Vorgehensweise sieht also wie folgt aus:

- Wenn ein Dokumentfenster aktiviert wird, benachrichtigt es das Hauptfenster darüber, dass es gerne eine oder mehrere Werkzeugleisten abgeben würde, und gibt dem Hauptfenster damit die Möglichkeit, diese Werkzeugleisten zu übernehmen. Dazu muss das Hauptfenster lediglich das *Parent*-Property der Werkzeugleisten auf sich selbst setzen und die Leisten eventuell noch sichtbar machen (damit dieser Vorgang vor dem Benutzer verborgen bleibt, sollten die Leisten nämlich im Dokumentfenster unsichtbar sein).
- Wenn das Fenster wieder deaktiviert wird, gibt es dem Hauptfenster erneut Bescheid, so dass dieses die Werkzeugleisten wieder an das Dokumentfenster zurückgeben kann.

Hinweis: Das Dokumentfenster könnte die Werkzeugleisten natürlich auch selbst in das Hauptfenster einblenden, die hier beschriebene Vorgehensweise entspricht aber bereits der noch zu zeigenden Vorgehensweise im TreeDesigner, bei der das Ziel verfolgt wurde, das Dokumentfenster möglichst unabhängig vom Hauptfenster zu halten.

Maßnahme für ein ruhiges Bild

Das alleinige Einblenden der Werkzeugleiste ist noch nicht als endgültige Lösung geeignet. Für den Anwender läuft das Einblenden nämlich nicht ganz so unsichtbar ab wie für den Programmierer. Wenn der Anwender zwischen zwei Dokumentfenstern wechselt, wird zuerst die Werkzeugleiste des bisher aktiven Fensters entfernt. Angenommen, die Werkzeugleiste war zu diesem Zeitpunkt gerade unter der Menüzeile per *Align* = *alTop* befestigt. Durch das Entfernen der Leiste könnte sich dann der

gesamte Arbeitsbereich des MDI-Hauptfensters mitsamt allen Dokumentfenstern nach oben in die von der Werkzeugleiste gelassene Lücke verschieben. Daraufhin wird die neue Werkzeugleiste eingefügt, was eine sofortige Zurückversetzung des verschobenen Bereichs an die bisherige Position zur Folge hätte. Je nach Systemgeschwindigkeit entstünde so ein störendes Flackern oder ein umständlich erscheinender Bildumbau.

Um das zu vermeiden, können Sie ein leeres Panel als Platzhalter in das Hauptfenster einfügen und die lokale Werkzeugleiste als Kindfenster in dieses Panel einblenden. Das Platzhalter-Panel braucht niemals entfernt zu werden und dient quasi als Stütze gegen ein kurzfristiges Einbrechen des Arbeitsbereichs in die von der Werkzeugleiste gelassene Lücke.

Hinweis: Im TreeDesigner werden die Symbolleisten zwar nicht in ein Panel, sondern in eine *TControlBar*-Komponente des Hauptfensters eingefügt, trotzdem verwendet auch das TreeDesigner-Hauptfenster vier »Stütz-Panel«, die nicht nur einen möglichst ruhigen Bildaufbau garantieren, sondern in der Windows-Version des TreeDesigners mit VCL-Docking auch noch sicherstellen, dass die Symbolleisten der Dokumentfenster beliebig abgetrennt und angedockt werden können, ohne dass die Dokumentfenster etwas davon merken und ohne dass das Hauptfenster sich um die Symbolleisten des Dokumentfensters kümmern muss.

Abstimmung zwischen Dokument- und Hauptformular *R43*

Die schnelle Lösung wäre nun, das Platzhalter-Panel des Hauptformulars direkt aus dem Dokumentformular heraus anzusprechen, um die Werkzeugleiste einzufügen. Auf diese Weise würde jedoch wieder eine Abhängigkeit des Dokumentformulars vom Hauptformular entstehen. Diese kann aber leicht vermieden werden, zum Beispiel,

- indem das Hauptformular dem Dokumentformular sagt, welches Panel zur Leisten-Einblendung vorgesehen ist (es könnte z. B. eine Variable des Dokumentfensters auf dieses Panel setzen);
- oder indem das Hauptfenster dem Dokumentfenster eine Schnittstelle zur Verfügung stellt, über die das Dokumentfenster dem Hauptfenster mitteilen kann, welche Werkzeugleisten ein- oder auszublenden sind. Das Einblenden und Ausblenden könnte dann im Hauptfenster stattfinden.

Der TreeDesigner definiert die genannte Schnittstelle über ein Interface:

```
type
  IToolbarHost = interface
    function TBHInsertPart(W: TWidgetControl; Width: Integer;
                           Visible: Boolean): Boolean;
    procedure TBHRemovePart(W: TWidgetControl);
    procedure TBHBeginUpdate;
    procedure TBHEndUpdate;
```

```
    procedure SetVisible(Part: TWidgetControl; Value: Boolean);
    function GetVisible(Part: TWidgetControl): Boolean;
    property PartsVisible[W: TWidgetControl]: Boolean
               read GetVisible write SetVisible;
  end;
```

Ein Hauptfenster, das zur Aufnahme von Toolbars des TreeDesigner-Dokumentfensters bereit ist, muss eine solche Schnittstelle bereitstellen, dem Dokumentformular also eine Möglichkeit geben, die in diesem Interface deklarierten Methoden aufzurufen (zu Details über die Interfaces siehe Kapitel 2.7). Die Aufgabe der einzelnen Methoden wird später am praktischen Aufruf erläutert.

Übergabe der Schnittstelle

Die Klasse *TDocumentForm* des TreeDesigners verlangt, dass eine solche Schnittstelle als Parameter für den *Create*-Konstruktor angegeben wird. Dieser speichert die Schnittstelle in der *TDocumentForm*-Variablen *ToolbarHost*, über die *TDocumentForm* später die oben genannten vier Methoden aufrufen wird:

```
constructor TDocumentForm.Create(AOwner: TComponent;
                                 TBH: IToolbarHost);
begin
  ToolbarHost := TBH;
  inherited Create(AOwner);
end;
```

Für die Übergabe dieses zweiten *Create*-Parameters muss die Formularerzeugungs-Methode *TMainForm.NewWindow* aus Kapitel 6.7.4 angepasst werden:

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  Window := TDocumentForm.Create(self, MainFormUtil, Document);
  ...
```

Dabei ist *MainFormUtil* die geforderte Schnittstelle, deren Herstellung wir uns später genauer ansehen werden; zunächst geht es nur darum, wie die Methoden dieser Schnittstelle vom Dokumentformular aufgerufen werden. Wichtig ist dabei, dass der Aufbau des Hauptfensters für das Dokumentformular überhaupt keine Rolle spielt. Das Dokumentformular weiß noch nicht einmal, ob das Hauptfenster seine Werkzeugleisten in eine ControlBar oder in ein normales Panel einordnen wird.

Praktische Umsetzung

Unser Ziel ist es also, bei jeder Aktivierung oder Deaktivierung eines Dokumentfensters die oben gezeigte Schnittstelle zu verwenden, um die Symbolleisten ein- bzw. auszublenden. Dazu brauchen wir als Erstes ein passendes Ereignis. Die Ereignisse

OnActivate und *OnDeactivate* scheinen vom ihrem Namen her bereits gut geeignet zu sein, bei näherer Betrachtung stellt sich jedoch heraus, dass sie nicht nur bei einem Wechsel zwischen den MDI-Fenstern auftreten, sondern auch, wenn der Eingabefokus an ein Eingabeelement einer Leiste wechselt, angenommen, dass sich diese Leiste bereits im Hauptfenster, also *außerhalb* des MDI-Kindfensters befindet. Würden wir die Leiste beim *OnDeactivate*-Ereignis entfernen, so würde sie paradoxerweise auch dann sofort verschwinden, wenn der Anwender mit der Maus auf eines ihrer Eingabeelemente klickt.

MdiActivate und MdiDeactivate *R40*

Wir benötigen also ein Ereignis, das nur auftritt, wenn der Benutzer von einem MDI-Kindfenster zu einem anderen MDI-Kindfenster wechselt. Während frühere TreeDesigner-Versionen noch ein spezielles Windows-Ereignis mit dem Namen *WM_MdiActivate* abgefangen haben, geht die aktuelle Version auf eine plattformunbhängige Weise vor: Das Hauptfenster erhält dabei die Aufgabe, zu überwachen, wann ein anderes MDI-Kind/Dokumentfenster aktiviert wird. Dafür definieren die Dokumentfenster ein neues Ereignis namens *OnActivateDocWin*. Dieses soll eintreten, wenn das Dokumentfenster ein *OnActivate*-Event empfängt. Im Prinzip ist also *OnActivateDocWin* lediglich die Weitergabe des *OnActivate*-Ereignisses vom Dokumentfenster an ein anderes Fenster (in diesem Fall das Hauptfenster):

```
procedure TDocumentForm.FormActivate(Sender: TObject);
begin
  if Assigned(FOnActivateDocWin) then
    FOnActivateDocWin(self);
end;
```

Das Hauptfenster bearbeitet das Ereignis mit der folgenden Methode:

```
procedure TMainForm.ChildWindowActivate(Sender: TObject);
var
  NewActiveDoc: TDocumentForm;
begin
  if (Sender is TDocumentForm) then begin
    NewActiveDoc := Sender as TDocumentForm;
    if NewActiveDoc <> LastActiveDoc then begin
      if LastActiveDoc <> nil then
        LastActiveDoc.MdiDeactivate;
      NewActiveDoc.MdiActivate;
      LastActiveDoc := NewActiveDoc;
    end;
  end;
end;
```

Wenn das neu aktivierte Fenster ein anderes ist als das bisher aktive Fenster, wird die *MdiDeactivate*-Methode des bisher aktiven Fensters und die *MdiActivate*-Methode des

neu aktivierten Fensters aufgerufen. Außerdem merkt sich die Methode in der Formularvariablen *LastActiveDoc*, welches Fenster zuletzt in *ChildWindowActivate* bearbeitet wurde.

Als Ergebnis der bisherigen Vorgänge können wir festhalten: Wenn der Benutzer zu einem anderen MDI-Dokumentfenster wechselt, können wir in der Methode *MdiDeactivate* die Leisten des alten Dokumentfensters aus dem Hauptfenster entfernen und in *MdiActivate* die des neuen in das Hauptfenster einsetzen. Die Verantwortlichkeit für den Aufruf von *MdiActivate/MdiDeactivate* liegt beim Hauptfenster, das allerdings umgekehrt von seinen Dokumentfenstern erwarten darf, über deren Aktivierung informiert zu werden.

Die Verknüpfung des Ereignisses *TDocumentForm.FOnActivateDocWin* mit *TMainForm.ChildWindowActivate* geschieht übrigens ausnahmsweise nicht über ein Property. Da die Bearbeitung diese Ereignisses so wichtig ist, wurde es als weiterer Parameter in den Konstruktor *TDocumentForm.Create* hinzugefügt. Dadurch wird dem oben bereits erweiterten Aufruf ein weiteres Element hinzugefügt:

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  Window := TDocumentForm.Create(self, MainFormUtil,
                                 Document, ChildWindowActivate);
```

Die Symbolleisten des TreeDesigners

Der Aufruf der Methoden *MdiActivate* und *MdiDeactivate* ist also vorbereitet, was noch fehlt, ist deren Implementierung. Zunächst zu *MdiActivate*, die dafür zuständig ist, die Symbolleisten des Fensters in das Hauptformular einzublenden. Im Falle des TreeDesigners haben wir es gleich mit vier *TToolBar*-Symbolleisten zu tun, die in das Hauptfenster eingeblendet werden sollen. Mit Hilfe der bereits genannten *ToolbarHost*-Schnittstellenmethoden fügt das Dokumentformular diese wie folgt in das Hauptfenster ein:

```
procedure TDocumentForm.MdiActivate(var Param: TWMMdiActivate);
begin
  if Assigned(ToolbarHost) then
    if Param.ActiveWnd = Handle then begin
      ToolbarHost.TBHBeginUpdate;
      ToolbarHost.TBHInsertPart(ShapeTextToolbar, 180, True);
      ToolbarHost.TBHInsertPart(ZoomToolbar, 240, True);
      ToolbarHost.TBHInsertPart(FontToolbar, 283, True);
      ToolbarHost.TBHInsertPart(NTToolbar, 60, Assigned(Document)
                                and Document.DrawEnhanced);
      ToolbarHost.TBHEndUpdate;
    end
  end;
end;
```

Jede Symbolleiste wird über einen Aufruf von *TBHInsertPart* in das Hauptfenster eingefügt.

- Erster Parameter der Methode ist die einzufügende Symbolleiste, in diesem Fall immer eine *TToolbar*-Komponente.
- Im zweiten Parameter wird die notwendige Mindestbreite der Toolbar angegeben. Werden die Toolbars nämlich zur Entwurfszeit im Dokumentformular mit *alTop* ausgerichtet, so stellt das ihre Breite auf die Breite des Dokumentformulars ein. Um nicht alle Begrenzungen der ControlBar des Hauptfensters zu sprengen, muss die Breite daher auf das notwendige Minimum verringert werden, bevor die Leisten im Hauptfenster eingeblendet werden.
- Der dritte Parameter besagt, ob die Symbolleiste sichtbar sein soll. Dies ist bei den Leisten des TreeDesigners per Voreinstellung immer der Fall.

Alle Aufrufe von *TBHInsertPart* werden schließlich durch Aufrufe von *TBHBeginUpdate* und *TBHEndUpdate* eingeklammert, so dass sich das Hauptfenster (bzw. das Objekt, das sich hinter der Schnittstelle *ToolbarHost* verbirgt) nicht nach jedem einzelnen Einfügen aktualisieren muss, sondern erst nach dem *TBHEndUpdate*-»Signal«. Dies ist allerdings nur eine Idee für zukünftige Erweiterungen, denn in der derzeitigen Version tun *TBHBegin/EndUpdate* nichts, die Bildschirmdarstellung wird also bei jedem *TBHInsertPart*-Aufruf sofort aktualisiert.

Das Entfernen der Werkzeugleiste findet auf ähnliche Weise über die Methode *TBHRemovePart* statt:

```
procedure TDocumentForm.MdiDeactivate;
begin
  if Assigned(ToolbarHost) then begin
    ToolbarHost.TBHBeginUpdate;
    ToolbarHost.TBHRemovePart(ShapeTextToolbar);
    ToolbarHost.TBHRemovePart(ZoomToolbar);
    ToolbarHost.TBHRemovePart(FontToolbar);
    ToolbarHost.TBHEndUpdate;
  end;
end;
```

Implementierung der IToolbarHost-Schnittstelle

Offen geblieben ist bisher noch die *IToolbarHost*-Implementierung, die im TreeDesigner in einer eigenen Klasse zu finden ist:

```
TMainFormUtil = class(TInterfacedObject, IToolbarHost)
public
  constructor Create;
  ... hier folgen die vier Methoden...
```

Da es auch nur ein globales Hauptfenster gibt, schadet es nicht, ein globales Objekt dieser Hilfsklasse im Initialisierungsteil der Hauptformular-Unit zu erzeugen:

```
initialization
  MainFormUtil := TMainFormUtil.Create;
```

MainFormUtil ist das Objekt, das oben bereits an den *Create*-Konstruktor der Dokumentformulare übergeben wurde (innerhalb dieser Dokumentformulare ist *MainFormUtil* allerdings nicht als Objekt, sondern nur als Schnittstellenvariable ansprechbar).

Aus Platzgründen können hier nur zwei der vier implementierten Methoden abgedruckt werden. Beim Einfügen einer Symbolleiste verwendet der TreeDesigner die vier zur Entwurfszeit erzeugten Platzhalter-Panels innerhalb der ControlBar. *TBHInsertPart* sucht den ersten freien Platzhalter und fügt die als Parameter erhaltene Komponente dort ein:

```
function TMainFormUtil.TBHInsertPart(W: TWidgetControl;
               PartWidth: Integer; Visible: Boolean): Boolean;
var
  Dest: TPanel;
begin
  Result := False;
  Dest := nil;
  with MainForm do // Freien Platzhalter suchen und in Dest speichern:
    if Panel1.ControlCount = 0 then Dest := Panel1
    else if Panel2.ControlCount = 0 then Dest := Panel2
    else if Panel3.ControlCount = 0 then Dest := Panel3
    else if Panel4.ControlCount = 0 then Dest := Panel4;
  if Assigned(Dest) then begin // Wenn ein Platzhalter frei ist...
    // Wenn W unsichtbar sein soll, werden die Elternkomponenten
    // gleich zu Beginn unsichtbar gemacht, damit keine unnötigen
    // Aktualisierungen am Bildschirm erscheinen:
    if not Visible then begin
        Dest.Visible := False;
    end;
    Dest.Width := PartWidth;
    W.Parent := Dest;
    W.Visible := True; // W ist nun sichtbar, wenn auch die
                       // Elternkomponenten sichtbar sind.
    if W is TToolbar then
      Dest.Caption := (W as TToolBar).Caption;
    // Symbolleiste erst am Schluss sichtbar machen, falls erforderlich:
    if Visible then PartsVisible[W] := True;
    Result := True;
  end;
end;
```

Für das Abtrennen der Symbolleisten ist es auch wichtig, dass das *Caption*-Property der Platzhalter-Panel auf den *Caption*-Text der Symbolleisten gesetzt wird, damit die CLX die frei verschiebbaren Symbolleisten mit einem aussagekräftigen Fenstertitel versieht.

Das Entfernen der Symbolleiste verläuft etwas einfacher:

```
procedure TMainFormUtil.TBHRemovePart(W: TWidgetControl);
var
  Dest: TPanel;
begin
  Dest := W.Parent as TPanel;
  with MainForm do
    if (Dest = Panel1) or (Dest = Panel2) or
       (Dest = Panel3) or (Dest = Panel4) then
    begin
      Dest.Caption := '';
      W.Visible := False;
      W.Parent := W.Owner as TWidgetControl;
    end;
end;
```

Wichtig ist dabei, dass *W* unsichtbar gemacht wird, *bevor* es wieder ein Kindelement seines Besitzers (also des Dokumentfensters) wird. Und da das Platzhalter-Panel dann ür einen Moment leer am Bildschirm erscheinen wird, sollte als Allererstes sein *Caption*-Property gelöscht werden, das dem Benutzer sonst als eine Art »Bildschirmmüll« in der Mitte der leeren *Panel*-Komponente erscheinen könnte.

Zusammenfassung *R42*

Zum schnellen Nachbau der dynamischen Toolbars können Sie die folgenden Schritte ausführen, wobei der einzufügende Code teilweise bereits an einem Beispiel gezeigt wurde:

- Erstellen Sie ein beliebiges MDI-Hauptformular und ein MDI-Dokumentformular, und übernehmen Sie die Unit *TBHost* mit der Schnittstellendefinition *IToollbarHost* aus dem TreeDesigner, oder passen Sie diese Schnittstelle für Ihre eigenen Bedürfnisse an.
- Fügen Sie im Hauptformular eine *TControlBar*-Komponente als Fundament für die Symbolleisten ein. Bei Bedarf fügen Sie globale Toolbars direkt in diese ControlBar ein, für jede einzublendende ToolBar eines MDI-Kindfensters fügen Sie nur ein Platzhalter-Panel ein.
- Entwerfen Sie in Ihrem Dokumentformular wie im Dokumentformular des TreeDesigners eine oder mehrere Toolbars, und schreiben Sie *MdiActivate*- und *MdiDeactivate*-Methoden, in denen Sie die Werkzeugleisten wie oben gezeigt über die *IToolbarHost*- oder Ihre eigene Schnittstelle einfügen und ausblenden (je nach dem Parameter *Param.ActiveWnd*).
- Um den Aufruf von *MdiActivate* und *MdiDeactivate* sicherzustellen, können Sie im MDI-Kindfenster ein Ereignis *OnActivateDocWindow* bereitstellen, das Sie im Hauptfenster bearbeiten, um *MdiActivate* und *MdiDeactivate* aufzurufen (nach dem Muster der gezeigten Methode *ChildWindowActivate*).

- Implementieren Sie die *IToolbarHost*-Schnittstelle in Ihrer Hauptformular-Unit und übergeben Sie die Schnittstellenvariable an jedes Ihrer Dokumentfenster (z. B. wie in der oben gezeigten Methode *TMainForm.NewWindow*). Zur Implementation der Schnittstelle müssen Sie den oben gezeigten Code wahrscheinlich kaum abändern.

6.7.6 Globale Werkzeugleisten und Mauspaletten

Dieses Kapitel beschäftigt sich mit der Programmierung von zwei der globalen Leisten des TreeDesigner-Hauptfensters: der Farbpalette am unteren und der Werkzeugleiste am linken Rand. Anders als die globale Toolbar mit den Schaltern für DATEI | NEU und DATEI | ÖFFNEN enthalten diese beiden Leisten nämlich zahlreiche Elemente, die vom gerade geöffneten Dokumentfenster abhängig sind. Sie müssen in zwei Richtungen mit den Dokumentfenstern verknüpft werden:

- Das Dokumentformular (bzw. das aktive Dokumentfenster) muss die Komponenten der beiden Leisten ansprechen können, z. B. muss es beim Einzeichnen eines neuen Grafikelements wissen, welche Grafikform und welche Farben gerade eingestellt sind. Also muss es auf Elemente zugreifen, die dem Hauptformular gehören.
- Die Ereignisse der Komponenten in den Leisten müssen vom Hauptformular an das Dokumentformular weitergeleitet werden. Hier findet also der umgekehrte Zugriff vom Haupt- auf das Dokumentformular statt.

Auslagern von Werkzeugleiste und Farbpalette *R44*

Wir nehmen dazu wie bereits in Kapitel 6.7.2 an, dass wir Werkzeugleiste und Farbpalette bereits im Dokumentformular definiert haben und diese nun in das Hauptformular auslagern wollen. Das eigentliche Auslagern einer Werkzeugleiste in das Hauptformular ist einfach über die Zwischenablage möglich: Markieren Sie die Leiste, wählen Sie BEARBEITEN | AUSSCHNEIDEN und fügen Sie die Leiste dann im Hauptformular entsprechend wieder ein.

Hinweis: Bei diesem Kopiervorgang werden grundsätzlich alle Property-Einstellungen kopiert, jedoch mit den folgenden Ausnahmen: Zur Vermeidung doppelter Namen erhalten einige Komponenten unter Umständen Vorgabenamen wie *Panel1*; leere *Caption*-Properties von *TPanel*-Komponenten werden mit dem Namen des Panels gefüllt; und wenn Sie die Komponenten in ein anderes Formular kopieren, werden die Verknüpfungen der Ereignisse grundsätzlich nicht übertragen (wenn Sie auch die verknüpften Methoden kopieren wollen, können Sie dies dadurch erreichen, dass Sie die Komponenten nicht über die Zwischenablage kopieren, sondern mit DER OBJEKTABLAGE HINZUFÜGEN... eine Schablone davon anfertigen und diese im Zielformular einfügen.)

Nun muss wie erwähnt der Programmcode des Dokumentformulars auf einige Elemente der Leisten zugreifen. Ein Beispiel dafür ist die Methode *TDocumentForm.SetAttributes*, in der die Einstellungen der Leisten-Steuerelemente auf die Properties eines Grafikelements übertragen werden:

```
procedure TDocumentForm.SetAttributes(Obj: TGraphicElement);
{ Alle Einstellungen in den Symbolleisten auf ein Grafikobjekt anwenden. }
begin
  Obj.Pen.Color := ColorControl.PenColor;
  Obj.Brush.Color := ColorControl.BrushColor;
  Obj.Font.Color := ColorControl.FontColor;
  Obj.Font.Height := FontSize.Value;
  Obj.Font.Name := FontList.Text;
  // Der folgende Aufruf wird unten noch einmal geändert, wenn
  // der Trackbar PenWidthControl in ein Frame ausgelagert wird:
  Obj.Pen.Width := PenWidthControl.Position;
  ...
```

Hier werden in den ersten drei Zeilen die Properties der Farbpalette, in den folgenden beiden Zeilen zwei Einstellungen aus der Schriftattribut-Toolbar und in der letzten Zeile ein Property des TrackBars zur Einstellung der Linienbreite ausgelesen. Um die Einstellungen der Schriftattribute-Toolbar brauchen wir uns hier nicht zu kümmern, denn wie in Kapitel 6.7.5 beschrieben, wird diese Toolbar im TreeDesigner zwar in das Hauptformular eingeblendet, bleibt aber trotzdem im Besitz des Dokumentformulars. Der Zugriff auf die Komponenten dieser Toolbar – hier *FontSize* und *FontList* – ist also kein Problem.

Die Farbpalette und die Werkzeugleiste mit dem TrackBar zur Einstellung der Linienbreite jedoch sollen ganz in das Hauptformular verlagert werden. Dies würde normalerweise bedeuten, dass *ColorControl* und *PenWidthControl* nicht mehr wie oben aus dem Dokumentfenster angesprochen werden können, sie müssten in den obigen Zeilen mit *MainForm.ColorControl* und *MainForm.PenWidthControl* angesprochen werden. Es ist jedoch möglich, dass die Zugriffe auf *ColorControl* und *PenWidthControl nicht* geändert zu werden brauchen. Dazu werden *ColorControl* und *PenWidthControl* nach der Verlagerung erneut als Variablen des Dokumentformulars deklariert, diesmal aber manuell im *public*-Bereich und nicht in dem von Kylix verwalteten *published*-Bereich am Anfang der Deklaration:

```
TDocumentForm = class(TForm)
begin
   // Von Kylix verwalteter published-Bereich.
   ...
public
   // Eigene Deklarationen
  FDocument: TGraphicDoc;
  ColorControl: TColorPalette;
  PenWidthControl: TTrackBar;
  ...
```

Nun müssen diese Variablen lediglich noch auf die entsprechenden Komponenten des Hauptformulars gesetzt werden. Im TreeDesigner geschieht das sofort bei der Konstruktion eines neuen Formulars, also im Formularkonstruktor. Der schon in Kapitel 6.7.5 um zwei Parameter erweiterte Konstruktor erhält nun abermals zwei neue Parameter:

```
constructor TDocumentForm.Create(AOwner: TComponent; TBH: IToolbarHost;
   APalette: TColorPalette; ATrackBar: TTrackBar; ADocument: TGraphicDoc;
   AOnActivateDoc: TNotifyEvent);
begin
  // hier finden zunächst die Dokument-View-bezogenen
  ...         // Initialisierungen statt (siehe Kapitel 6.3.1)
  ToolbarHost := TBH;
  ColorControl := APalette;
  PenWidthControl := ATrackBar;
  FOnActivateDocWin := AOnActivateDoc;
  inherited Create(AOwner);
end;
```

Entsprechend muss auch der Aufruf des Konstruktors im Hauptformular angepasst werden. Die Initialisierung der Variablen *ColorControl* und *PenWidthControl* gleich zu Anfang und die Tatsache, dass das Hauptfenster immer länger existiert als seine Kindfenster, garantieren, dass das Dokumentformular jederzeit auf die beiden Variablen zugreifen kann, als handle es sich dabei um seine eigenen Steuerelemente.

Hinweis: In der kompletten TreeDesigner-Version ist *TDocumentForm.Create* um noch zwei weitere Parameter reicher. Der erste gibt dem Dokumentformular Zugriff auf eine *TMenuToolBar*-Komponente des Hauptformulars (die Verwendung dieser Komponente ist optional, standardmäßig verwendet der TreeDesigner unter Linux eine normale Menüleiste). Der letzte Zusatz-Parameter ist für die Liste der zurückliegenden Dateien gedacht, die vom Dokumentformular erweitert wird, wenn der Benutzer eine Datei mit DATEI | SPEICHERN UNTER speichert.

Weiterleiten der Ereignisse zum Dokumentformular *R38*

Da der Objektinspektor die Ereignisse der Komponenten nur mit Methoden desselben Formulars verknüpfen kann, müssen im TreeDesigner einige Ereignisse der globalen Werkzeugleisten manuell an das Dokumentformular weitergegeben werden. Die einzige kleine Schwierigkeit dabei ist die Tatsache, dass es zur Laufzeit sehr viele Dokumentfenster geben kann und normalerweise nur eines davon das Ereignis bearbeiten soll.

Doch lässt sich das aktive Dokumentfenster leicht über das Property *TForm.ActiveChild* herausfinden. Da wir im Gegensatz zum Compiler ja wissen, dass alle MDI-Kinder des TreeDesigners immer den Typ *TDocumentForm* haben, können wir *ActiveChild* außerdem noch ohne vorherige Sicherheitsabfrage mit dem *as*-Operator in ein *TDocumentForm*

umwandeln, so dass sich anschließend alle Methoden des aktiven Dokumentformulars direkt aufrufen lassen. Das folgende Beispiel zeigt die Weitergabe des *OnBrushColorChange*-Ereignisses der Farbpalette:

```
procedure TMainForm.PaletteBrushColorChange(Sender: TObject;
  NewColor: TColor);
begin
  StylePreview.Invalidate; // Vorschau (Werkzeugleiste unten) neu zeichnen
  if ActiveMdiChild <> nil then
    with ActiveMdiChild as TDocumentForm do
      BrushColorChange(Sender, NewColor);
end;
```

Hinweis: Wenn Ihre Anwendung über mehrere Typen von MDI-Kindfenstern verfügt, können Sie die Klasse des aktiven Fensters über den *is*-Operator überprüfen.

Wenn mehrere ähnliche Ereignisse an das Dokumentformular weitergeleitet werden müssen, brauchen Sie nicht für jedes eine eigene Methode zu schreiben. Mehrere Schalter können beispielsweise durch den Wert des *Tag*-Properties unterschieden und von einer einzigen Methode bearbeitet werden:

```
procedure TMainForm.ChildCommandClick(Sender: TObject);
begin
  if ActiveMdiChild <> nil then
    with ActiveMdiChild as TDocumentForm do
      case (Sender as TSpeedButton).Tag of
        1: cmArrangeTreeClick(Sender);
      { 2: ...  (in der TreeDesigner-Werkzeugleiste befindet
                sich nur ein Aktionsschalter) }
      end;
end;
```

erwendung der Frame-Technik zur Mehrfachnutzung einer Leiste *R3 I*

Nehmen wir nun einmal an, dass auf diese Weise sehr viele Komponenten von einem Dokumentformular in ein Hauptformular verschoben werden sollen. Bei der oben beschriebenen Vorgehensweise müsste jede verschobene Komponente einzeln wieder mit ihrem Ursprungsformular verknüpft werden.

Oder nehmen wir an, dass zwei Versionen der Anwendung parallel gepflegt werden sollen – z. B. beim TreeDesigner sowohl die MDI- als auch die SDI-Version. Wenn jetzt etwa die Werkzeugleiste im Hauptformular der MDI-Version geändert wird, muss uch das Dokumentformular der SDI-Version, in dem ja eine Kopie der Leiste enthalten ist, entsprechend mitverändert werden und umgekehrt.

Um diese sehr umständlichen Wartungsarbeiten zu vermeiden, bietet sich die in Kapitel 3.7.2 detailliert erläuterte Frame-Technik an. Im Falle des TreeDesigners wird sie

wie folgt verwendet: Die Werkzeugleiste wird weder im Dokument- noch im Hauptformular definiert, sondern in einem separaten Frame namens *GlobalToolbar*. Zu diesem Frame gehören auch einige Methoden, die in der zugehörigen Unit *FrameGlobalToolbar* definiert sind, z. B. die Methode, die im unteren Bereich der Toolbar einen kleinen Kreis als Vorschau für die gerade eingestellten Farben und die Stiftbreite zeichnet (siehe z. B. Abbildung 6.1).

Das *GlobalToolbar*-Frame wurde so gestaltet, dass es sich wie eine Komponente verwenden lässt. Dazu gehört auch, dass es Properties und Events zur Verfügung stellt:

```
type
  TGlobalToolbar = class(TFrame)
    ...
  public
    property CurShapeType: TShapeType read FCurShapeType
                                      write SetCurShapeType;
    property OnShapeTypeChange: TNotifyEvent read ...;
    property OnPenWidthChange: TNotifyEvent read ...;
    property OnArrangeTreeClick: TNotifyEvent read ...
    property ColorPalette: TColorPalette read FColorPalette
  end;
```

In *CurShapeType* speichert das Frame, welche Grafikform gerade gewählt ist. Das Formular, das diesen Frame verwendet, braucht so nicht die einzelnen Schalter des Frames abzufragen. Mit *OnShapeTypeChange* und *OnPenWidthChange* sowie *OnArrangeTreeClick* kann es seine Methoden verknüpfen, die auf das Ändern der Grafikform oder der Stiftbreite und auf den Schalter zum Anordnen des Baumes reagieren. Schließlich erwartet *GlobalToolbar*, dass das *ColorPalette*-Property auf die Farbpalette gesetzt wird, deren Farben im Vorschaubereich berücksichtigt werden sollen.

All diese Properties richtig zu setzen, sei dem vollständigen Code auf der CD überlassen. An dieser Stelle soll nur kurz skizziert werden, wie der Frame im TreeDesigner »eingebaut« wird.

Zunächst zur MDI-Version: In dieser wird ein Exemplar des Frames in das Hauptformular eingezeichnet und jedes Dokumentformular erhält eine Referenz darauf. Wenn Sie das Kapitel 6 bisher vollständig gelesen haben, wird Ihnen der Konstruktor von *TDocumentForm* ja schon als ständige Baustelle bekannt sein. Auch das Werkzeugleisten-Frame wird ihm als Parameter übergeben. Da die Werkzeugleiste jedoch bereits den TrackBar für die Stiftbreite enthält, kann der in R44 Seite 725 dafür vorgesehene Parameter *ATrackBar* wieder weggelassen werden. Es ergibt sich dann der folgende Konstruktor:

```
constructor TDocumentForm.Create(AOwner: TComponent;
   GTB: TGlobalToolbar; TBH: IToolbarHost; APalette: TColorPalette;
   ADocument: TGraphicDoc; AOnActivateDoc: TNotifyEvent);
begin
```

```
ToolbarHost := TBH;
GlobalToolbar := GTB;
...
```

Im restlichen Code des Dokumentformulars kann nun über die *GlobalToolbar*-Variable auf die einzelnen Komponenten und Properties des Frames zugegriffen werden. Z.B. ndert sich die Einstellung der Stiftbreite in der oben schon einmal gezeigten Methode *SetAttributes* wie folgt:

```
procedure TDocumentForm.SetAttributes(Obj: TGraphicElement);
begin
  if Assigned(GlobalToolbar) then
    Obj.Pen.Width := GlobalToolbar.PenWidth.Position
```

Die linksseitige Werkzeugleiste des TreeDesigners war in früheren Versionen einmal fester Bestandteil des Hauptformulars und wurde erst in der Version 3.5 als Frame ausgelagert. Eine solche Umstellung bringt folgende Vorteile:

- Für alle im Dokumentformular genutzten Elemente der Toolbar muss am Anfang nur eine Referenz auf den Frame an das Formular übergeben werden – im Falle des TreeDesigner an dessen Konstruktor.
- Da der Frame selbst einige Methoden enthält, verringert sich dadurch die Größe der Hauptformular-Unit (in diesem Fall immerhin um 100 von ca. 1500 Zeilen). Die vorher über die Hauptformular-Unit verstreute Funktionalität des Frames ist nun erheblich besser gekapselt.
- Die SDI-Version der Anwendung kann mit minimalem Aufwand aus der MDI-Version erzeugt werden.

erwenden des MDI-Kindfensters als SDI-Hauptformular

Das Dokumentformular des TreeDesigners erreicht eine hohe Unabhängigkeit vom Hauptfenster, indem es keine Variablen des Hauptformulars direkt anspricht, sondern nur indirekt über eigene Variablen, die im Konstruktor gesetzt werden. Die an zwei Stellen zu findende Abhängigkeit vom Dokumentformular wurde nur zur Demonstration der verschiedenen Möglichkeiten im Abschnitt *Hauptmenüpunkte im Kindfenster* auf Seite 715 eingegangen. Beide Abhängigkeiten ließen sich (wie ebenfalls im genannten Abschnitt beschrieben) leicht auflösen.

Die Unabhängigkeit geht sogar so weit, dass das Dokumentformular sich selbstständig machen und als Hauptfenster in einer eigenen SDI-Anwendung arbeiten kann. Auf der CD finden Sie im Projekt *TreeDesignerSDI* eine SDI-Version des TreeDesigners, die sich von der MDI-Version hauptsächlich dadurch unterscheidet, dass kein Exemplar des MDI-Hauptformulars konstruiert wird. Als Hauptformular dient stattdessen ein Exemplar der Klasse *TDocumentForm* aus der unveränderten Dokumentformular-Unit

der MDI-Version. Wie jedes Hauptformular wird auch das Dokumentformular nun in der Projektdatei erzeugt, und zwar durch folgenden Aufruf:

```
Application.CreateForm(TDocumentForm, DocumentForm);
```

Dabei wird der normale *Create*-Konstruktor, den *TDocumentForm* von *TForm* erbt, aufgerufen, nicht jedoch der oben beschriebene, ständig um neue Parameter erweiterte Konstruktor (was ja auch kein Wunder ist, denn wie sollte die CLX diese ihr unbekannten Parameter setzen?).

TDocumentForm muss also auch funktionieren, wenn es mit dem normalen *Create*-Konstruktor aufgerufen wird. Dazu müssen alle Variablen, die in der MDI-Version von außen an den *Create*-Konstruktor übergeben werden, nun vom Dokumentformular selbst initialisiert werden, indem die fehlenden Komponenten etwa dynamisch selbst konstruiert werden. *TDocumentForm* erledigt dies bei seinem *OnCreate*-Ereignis.

Da ja dieselbe Formularversion in MDI- und SDI-Version verwendet werden soll, müssen nun in *FormCreate* beide Fälle berücksichtigt werden. Der Fall »MDI-Version« kann daran erkannt werden, dass in der Projektdatei die Anweisung

```
Application.CreateForm(TMainForm, MainForm);
```

ausgeführt wurde, also die Variable *MainForm* in der Unit *MdiForm* gesetzt ist. In der SDI-Version wird kein Exemplar von *TMainForm* erzeugt, daher bleibt *MainForm* dort *nil*.

Aus Platzgründen kann nur ein Ausschnitt aus der *OnCreate*-Methode gedruckt werden, was jedoch zum Verständnis des Prinzips ausreichen sollte. Außer dem unbedingt notwendigen *GlobalToolbar*-Frame und der ebenfalls wichtigen Farbpalette wird auch eine *TControlBar* für die flexible Aufbewahrung der Toolbars erzeugt (diese können ja nun nicht mehr in die ControlBar eines MDI-Hauptfensters verlagert werden):

```
procedure TDocumentForm.FormCreate(Sender: TObject);
var
  FontIndex: Integer;
  CB: TControlBar;
begin
  if Assigned(MdiForm.MainForm) then begin // MainForm nur in MDI-Version!
    CmGlobalNew.OnClick := MainForm.MDINewClick; // (bekannt aus K.6.7.4)
  end else begin
    // Einrichten des Formulars für die SDI-Version:
    Menu := MDIChildMenu; // Menü als Hauptmenü anzeigen
    GlobalToolbar := TGlobalToolbar.Create(self);
    GlobalToolbar.Parent := self;
    CB := TControlBar.Create(self);
    with CB do begin
      Parent := self;
      Align := alTop;
      AutoSize := True;
```

```
      Height := ShapeTextToolbar.Height * 2;
    end;
    ShapeTextToolbar.Parent := CB;
    ... (Einfügen der Toolbars in die Controlbar)
    ColorControl := TColorPalette.Create(self);
    ... (Erzeugen einer Farbpalette)
    // Verknüpfen der Frame-Events mit eigenen Methoden:
    with GlobalToolbar do begin
      OnShapeTypeChange := self.SpeedBtnRectangleDblClick;
      OnPenWidthChange := self.PenWidthChange; // Hier ist "self." wichtig
      OnArrangeTreeClick := self.cmArrangeTreeClick;
    end;
  end;
  ... (weitere Initialisierungen für beide Fälle)
end;
```

Hinweise: Auch das *FormStyle*-Property muss natürlich in SDI- und MDI-Version verschieden gesetzt werden. Im TreeDesigner-Dokumentformular wurde dieses Property zur Entwurfszeit bereits mit *fsNormal* initialisiert. Der bereits mehrfach erwähnte spezielle *Create*-Konstruktor, der nur in der MDI-Version aufgerufen wird, setzt *FormStyle* dann auf *fsMdiChild*.

Der Spezial-Code zur Initialisierung des SDI-Formulars ist noch erweiterungsfähig. Die aktuelle SDI-Version des TreeDesigners enthält beispielsweise noch keine Docking-Fähigkeiten für Baumansichts- und Übersichtsfenster und keine Statuszeile. Beides ist im MDI-Hauptfenster definiert und müsste noch gesondert in das Dokumentformular übertragen werden.

6.8 Erweiterung der Benutzerschnittstelle

In diesem Kapitel geht es um einige weitere Themen, die mit der Benutzerschnittstelle einer Anwendung zusammenhängen:

- Tastatureingaben in eine Komponente, die von sich aus noch keine Tastatureingabe bearbeitet, um die automatische Weiterleitung des Tastaturfokus und speziell um die Handhabung des Tastaturfokus im TreeDesigner (Kapitel 6.8.1);
- Andocken der beiden Hilfsfenster des TreeDesigners (Kapitel 6.8.2);
- Positionsabhängige Popup-Menüs für die TreeDesigner-Zeichenfläche (Kapitel 6.8.3).
- Drag&Drop (Kapitel 6.8.3).

6.8.1 Tastatursteuerung

Da die meisten Tastatureingaben bereits von Menüs und Steuerelementen verarbeitet werden, ist es in den meisten Formularen nicht notwendig, die Tastaturereignisse selbst zu bearbeiten. Auch der TreeDesigner bearbeitet nur zwei verschiedene Tasten zur Veränderung der Grafikobjekte: Entf und Bild↓. Dieses Kapitel zeigt Besonderheiten, die bei dieser Bearbeitung auftreten können, und gibt ein Beispiel für die gezielte Kontrolle des Tastaturfokus. Grundlegende Informationen zu den drei Tastatureingabe-Ereignissen, zu den beiden Fokusereignissen *OnEnter* und *OnExit* und zur Tastatursteuerung im Allgemeinen finden Sie in Kapitel 3.3.2.

Tastatureingaben

Ein Grafikprogramm kommt normalerweise nicht mit den Mausziehoperationen aus Kapitel 6.4 aus, sondern macht auch einige Funktionen über die Tastatur zugänglich, ohne dass dabei Menüs und Schalter eingesetzt werden. Im TreeDesigner sind das die folgenden Funktionen:

- Mit Entf löschen Sie alle markierten Grafikobjekte.
- Mit Bild↓ verschieben Sie das oberste Grafikelement hinter alle anderen Elemente und können so Elemente sichtbar machen, die vorher verdeckt waren.

Da beide Tasten Sondertasten sind und keinem Zeichen des ANSI-Zeichensatzes entsprechen, können sie nicht im Ereignis *OnKeyPress* bearbeitet werden, sondern sind nur über das Ereignis *OnKeyDown* erreichbar. Die abzufragenden Tastencodes sind *Key_Delete* für Entf und *Key_Next* für Bild↓.

Herkunft der Tastaturereignisse im TreeDesigner

Es stellt sich nun die Frage, wo sich der Tastaturfokus befindet, wenn der Benutzer im TreeDesigner beispielsweise Entf drückt, um ein Grafikelement zu löschen. Wir nehmen dazu den schwierigsten Fall für die Tastaturfokus-Verwaltung an: wenn die Werkzeugleiste im Dokumentfenster integriert ist (wie in der SDI-Version des TreeDesigners) und deren Eingabeelemente den Fokus an sich ziehen. Wenn der Benutzer nun gerade Text im Beschriftungsfeld der Leiste eingegeben hat, so hat dieses Eingabefeld noch den Fokus.

Will der Benutzer nun das aktuelle Grafikelement mit Entf löschen, wird er versuchen – wenn er mit der Bedienung grafischer Benutzeroberflächen vertraut ist –, die Zeichenfläche zu fokussieren, indem er beispielsweise mit der Maus auf sie klickt. Da die Paintbox als *TGraphicControl*-Komponente den Fokus jedoch nicht erhalten kann, kann der Fokus nur an die übergeordnete Scrollbox gehen (siehe auch *Der Fokus für die Scrollbox* auf Seite 733).

ür die beiden oben beschriebenen Tastatureingaben müssten also die Tastaturereignisse von *TScrollBox* behandelt werden. Bedauerlicherweise sind diese jedoch noch als *protected* deklariert und demnach weder im Objektinspektor noch zur Laufzeit ansprechbar.

Die Bergung verlorener Ereignisse

Das heißt also, dass die Tastaturereignisse gewissermaßen in den Tiefen von *TScrollBox* verloren gehen. Der TreeDesigner ist nur eines von vielen vorstellbaren Beispielen, in denen Tastatureingaben auch in einer Scrollbox auftreten können.

Um an solche verloren gegangene Ereignisse zu gelangen, können Sie zwischen den folgenden beiden Möglichkeiten wählen:

- Sie können das *KeyPreview*-Property des Formulars auf *True* einstellen, wodurch die Tastaturereignisse *aller* im Formular befindlichen Komponenten an dieses gesendet werden (in Form der üblichen *OnKey...*-Ereignisse). Sie müssten dann auch abfragen, welche Komponente gerade den Tastaturfokus hat (Property *ActiveControl*).
- Sie packen das Übel bei der Wurzel und bringen die Tastaturereignisse von *TScrollBox* an die Öffentlichkeit, und zwar durch Ableitung einer neuen Komponentenklasse von *TScrollBox*, bzw. Sie verwenden die Klasse *TScrollBoxEx* aus Kapitel 8.5.2.

Der TreeDesigner verwendet daher statt *TScrollBox* die erweiterte Klasse *TScrollBoxEx* und bearbeitet deren *OnKey...*-Ereignisse. Für die beiden erwähnten Tasten genügt bereits die Bearbeitung von *OnKeyDown*. Die mit den Tasten aufzurufenden Funktionen werden von eigenen Methoden bereitgestellt: *DeleteCommand* (Entf) und *ShiftOrderCommand* (Bild ↓), so dass beide Befehle leicht auch über *OnClick*-Methoden, beispielsweise des Hauptmenüs, aufgerufen werden können:

```
procedure TDocumentForm.ScrollboxKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if key = Key_Delete then
    DeleteCommand
  else if key = Key_Next then
    ShiftOrderCommand
end;

procedure TDocumentForm.DeleteCommand;
begin
  Document.DeleteSelected; { alle gewählten Objekte löschen }
  MouseObject := nil;  { kein Objekt ist mehr markiert }
end;

procedure TDocumentForm.ShiftOrderCommand;
```

```
begin
  if Document.Count > 1 then begin
    Document.DeselectAll; { die Markierung aller Objekte aufheben }
    Document.LastVisibleObjectToFront;
    MouseObject := Document.Items[Document.Count-1];
    MouseObject.Marked := True;
    // Markiertes Objekt in den sichtbaren Bereich rücken
    (* Im vollständigen Code wird außerdem sichergestellt, dass das
       markierte Objekt sichtbar ist (u.a. mit ScrollPointToCenter. *)
  end;
end;
```

Der Fokus für die Scrollbox *R78*

Das Bearbeiten der *OnKeyDown*-Ereignisse alleine genügt hier jedoch noch nicht, denn die Scrollbox gehört zu den Komponenten, die nicht automatisch den Tastaturfokus erhalten, wenn sie angeklickt werden. Der Benutzer könnte den Fokus also nur im Rahmen der üblichen Dialogsteuerung mit den Pfeiltasten oder mit ⇆ (falls das Property *TabStop* der Scrollbox *True* ist) zur Scrollbox bringen.

Normalerweise ist es angemessen, dass eine Scrollbox sich bei Mausklicks nicht den Fokus reserviert, denn oft kann sie gar nichts mit den Tastatureingaben anfangen. So ist es auch zu erklären, dass *TScrollBox* die Tastaturereignisse nicht veröffentlicht.

Um eine von *TWidgetControl* abgeleitete Komponente wie *TScrollBox* zu einem vollwertigen Eingabeelement zu machen, das den Tastaturfokus auch durch einen Mausklick erhalten kann, müssen Sie den Fokus beim Ereignis *OnMouseDown* selbst auf dieses Element setzen. Im TreeDesigner wird der Mausklick auf die Fläche der Scrollbox jedoch schon durch die darüber liegende Paintbox abgefangen. Daher besteht die erste Aktion der vollständigen Methode *PaintBoxMouseDown* im Setzen des Tastaturfokus auf die ScrollBox (die PaintBox selbst kann den Fokus ja nicht bekommen, da sie kein *TWidgetControl* ist):

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ScrollBox.SetFocus;
  ...
```

Tastatursteuerung für die Scrollbox *R83*

Wenn eine ScrollBox bereits den Tastaturfokus hat, bietet es sich an, die Ereignisse der Cursortasten zu bearbeiten, damit der Benutzer auch mit der Tastatur scrollen kann. Das Testprogramm für *TScrollBoxEx*, zu finden im CD-Verzeichnis `Complib` unter dem Namen *SbTest*, verwendet eine ScrollBoxEx, die Sie mit der Tabulatortaste ansteuern und mit den Cursortasten scrollen können:

```
procedure TForm1.ScrollBoxExtendedKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  with Sender as TScrollBox do begin
    case Key of
      Key_LEFT: HorzScrollBar.Position := HorzScrollBar.Position-10;
      Key_RIGHT: HorzScrollBar.Position := HorzScrollBar.Position+10;
      Key_UP: VertScrollBar.Position := VertScrollBar.Position-10;
      Key_DOWN: VertScrollBar.Position := VertScrollBar.Position+10;
    end;
  end;
end;
```

Automatisches Weiterleiten von Fokus und Tastatureingabe *R82*

Normalerweise steuert der Anwender den Tastaturfokus selbst durch die verschiedenen Steuerelemente, im einfachsten und möglicherweise langsamsten Fall einfach durch Anklicken der Elemente.

Manchmal weiß er es jedoch auch zu schätzen, wenn der Fokus automatisch zu der passenden Stelle springt. Im TreeDesigner ergibt sich eine gute Gelegenheit zu einer solchen Automatikfunktion: Der Benutzer kann nach dem Einzeichnen eines Grafikelements sofort den Beschriftungstext eingeben, obwohl der Tastaturfokus durch das Einzeichnen des Elements zur Scrollbox wandert.

Die Eingabe der Beschriftung ist sehr leicht von anderen Eingaben zu unterscheiden: Da in der Scrollbox nur die Tasten Entf und Bild↓ verwendet werden, kann sich der TreeDesigner denken, dass jede andere Taste einem anderen Zweck dienen soll. Bei jeder Eingabe eines einfachen Zeichens in die Scrollbox, also beim Ereignis *OnKeyPress*, leitet der TreeDesigner den Fokus und das schon eingegebene Zeichen daher an das Eingabefeld für die Beschriftung (*ShapeText*) weiter, indem er ein Tastendruck-Ereignis ür *ShapeText* simuliert.

Simulation von Ereignissen *R79*

Eine einfachere Lösung als die Simulation eines Ereignisses bestünde darin, die Eingabe des Benutzers einfach in das *Text*-Property von *ShapeText* zu schreiben und *ShapeText* zu fokussieren. Doch soll das bisherige *Text*-Property gelöscht oder das neu eingegebene Zeichen nur angehängt werden? An welcher Position soll danach die Eingabemarkierung stehen? Diese Fragen können wir vom Eingabefeld selbst beantworten lassen, indem wir *nicht* von außen sein *Text*-Property verändern, sondern ihm eine Nachricht über die gedrückte Taste senden.

Da unterhalb der CLX die Qt-Bibliothek für die Versendung von Nachrichten zu Widgets verantwortlich ist, verwendet der TreeDesigner die Methode *QApplication::sendEvent*. Diese verlangt als zweiten Parameter ein *QEvent*-Objekt; für den Fall einer Tastaturein-

gabe wird speziell ein *QKeyEvent* benötigt, das in Object Pascal mit der Funktion *QKeyEvent_create* erzeugt wird. Zu seinen Parametern finden Sie in der Qt-Dokumentation genauere Informationen, hier geht es nur darum, den *Key*, den das *OnKeyPress*-Ereignis als Parameter mitführt, weiterzugeben.

Als Ziel des Ereignisses im ersten *sendEvent*-Parameter ist das Handle eines Qt-Widgets anzugeben. Soll das Event beispielsweise an ein *TEdit*-Feld gesendet werden, wäre das etwa *Edit1.Handle*. Da *ShapeText* aber eine Combobox ist, die in Qt aus einer Liste und einem Editierfeld zusammengesetzt ist, muss der TreeDesigner zuerst noch das Editierfeld der Kombobox erfragen; dazu dient die Qt-Funktion *QComboBox_lineEdit*.

```
procedure TDocumentForm.ScrollBoxKeyPress(Sender: TObject; var Key: Char);
var
  DummyString: WideString;
  KeyEvent: QKeyEventH;
begin
  ShapeText.SetFocus;
  DummyString := chr(integer(key));
  KeyEvent := QKeyEvent_create(QEventType_KeyPress,
      integer(Key), integer(Key),
      Integer(ButtonState_NoButton),
      PWideString(@DummyString), False, 1);
  QApplication_sendEvent(
    QComboBox_lineEdit(ShapeText.Handle),
    KeyEvent);
  QKeyEvent_destroy(KeyEvent);
end;
```

Schließlich wird das selbst konstruierte Event mit *QKeyEvent_destroy* wieder freigegeben, da *sendEvent* das Ereignis sofort weiterleitet und erst zurückkehrt, wenn die Bearbeitung duchgeführt wurde. Anders würde es sich im Fall der alternativen Methode *postEvent* verhalten. Diese würde das Ereignis ans Ende der Warteschlange anfügen und sofort zum Aufrufer zurückkehren. Dieser dürfte das Ereignis-Objekt dann nicht mit *destroy* freigeben. Ein mit *postEvent* gesendetes Ereignis wird von Qt selbst freigegeben.

AutoSelect

Ein wichtiges Detail bei der Tastatursteuerung ist auch, dass der Inhalt des Editierfelds vollständig markiert wird, sobald das Feld den Fokus erhält. So ist es möglich, den gesamten Inhalt durch Eingabe eines neuen Zeichens zu überschreiben. In *TEdit*-Komponenten können Sie dieses Verhalten durch das Property *AutoSelect* abschalten, in *TCombobox* gibt es dieses Property nicht, der Feldinhalt wird dort immer automatisch selektiert.

eyPreview *R86*

Abschließend sei noch auf ein Beispiel zur Verwendung des am Anfang erwähnten *KeyPreview*-Events verwiesen. Das Beispielprogramm *PageControlNotebook* aus den Kapiteln 3.5.4 und 3.5.5 verwendet ein PageControl zur Darstellung mehrerer Dialogseiten. Der Benutzer kann bei dieser Komponente standardmäßig zwischen den Seiten umschalten, indem er [→] oder [←] drückt, solange das Seitenregister fokussiert ist; hat ein anderes Steuerelement den Fokus, kann er die Seiten noch mit [Strg]+[⇆] und [⇧]+[Strg]+[⇆] wechseln. Letztere beiden Tastenkombinationen können jedoch auch vom Fenstermanager für andere Zwecke reserviert sein (z.B. zum Umschalten zwischen Desktops), und falls der Fenstermanager keine Konfiguration zulässt oder der Benutzer diese Konfiguration behalten will, könnte das Programm selbst neue Tastenkürzel zum Umblättern definieren.

Das Beispielprogramm realisiert dies durch die Tastenkombinationen [Strg] + [Bild↓] und [Strg] + [Bild↑]. Das *KeyPreview*-Property des Formulars wurde auf *True* gesetzt, damit das Formular für alle Tastatureingaben in seine Komponenten Tastaturereignisse erhält. Die Methode für *OnKeyDown* braucht so nur die entsprechenden Tasten abzufangen und die Seiten umzublättern:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  NewPageIndex: integer;
begin
  NewPageIndex := OuterPageControl.ActivePageIndex;
  if ssCtrl in Shift then begin
    if Key = Key_Next then
      NewPageIndex := OuterPageControl.ActivePageIndex+1
    else if Key = Key_Prior then
      NewPageIndex := OuterPageControl.ActivePageIndex-1;
  end;
  if OuterPageControl.PageCount = NewPageIndex then NewPageIndex:=0
    else if NewPageIndex = -1 then
      NewPageIndex := OuterPageControl.PageCount-1;
  OuterPageControl.ActivePageIndex := NewPageIndex;
end;
```

6.8.2 Docking von Fenstern

Frei positionierbare Fenster wie das Übersichtsfenster und das Baumansichtsfenster im reeDesigner haben neben dem Vorteil der Flexibilität auch Nachteile: Wenn sie immer als oberstes Fenster angezeigt werden, können sie dem Benutzer schnell in den Weg geraten, weil sie beispielsweise darunter liegende Fensterbereiche verdecken. Wenn sie nicht als oberstes Fenster angezeigt werden, können sie leicht durch das größere

Hauptfenster verdeckt werden. In beiden Fällen muss der Benutzer die erhöhte Flexibilität mit mehr Arbeit (dem Verschieben von Fenstern bzw. dem Umherschalten zwischen diesen) bezahlen.

Die Vorteile von frei positionierbaren und fest in das Hauptfenster integrierten Fenstern lassen sich kombinieren, indem man das frei positionierbare Fenster andockbar macht. Die Kylix-IDE macht selbst ausgiebigen Gebrauch von dieser Technik, wobei Sie die andockbaren Fenster dort sogar miteinander verkoppeln und stapeln können.

Diese von der VCL unter Windows bekannte Docking-Unterstützung ist jedoch in der CLX (noch) nicht enthalten. Dieses Kapitel zeigt daher eine Lösung, wie Sie auch manuell die wichtigste Eigenschaft andockbarer Fenster nachbilden können: einen einfachen Weg für den Benutzer, das Fenster fest in das Hauptfenster zu integrieren und es wieder davon zu lösen.

Prinzipielle Funktionsweise des manuellen Dockings

Das Grundprinzip des manuellen Dockings ähnelt dem des Einblendens der Symbolleisten vom MDI-Kindfenster in das Hauptfenster (Kapitel 6.7.5) und besteht darin, dass eine Komponente eine neue Eltern-Komponente zugeordnet bekommt. Die angedockte Komponente wird zum Kind-Element des Hauptfensters bzw. einer Komponente des Hauptfensters (allgemein: einer Andockstelle).

Angedockt wird dabei nicht etwa das gesamte Hilfsfenster – denn Sie können in der CLX kein gesamtes Formular in ein anderes »einbauen« –, sondern nur der Inhalt des Hilfsfensters. Damit dieser über eine einzige Komponente in das Hauptfenster verschoben werden kann, können Sie beispielsweise ein Panel mit *alClient* ausrichten, so dass es das gesamte Hilfsfenster ausfüllt, und dann die Komponenten des Hilfsfensters darin einfügen. Beim Docken muss dann nur noch das Panel verschoben werden, alle darin enthaltenen Komponenten werden automatisch mitverschoben.

Als Nächstes benötigen Sie noch eine Andockstelle. Im einfachsten Fall können Sie die angedockte Komponente (im Folgenden sei angenommen, dass es sich um ein *Panel* handelt) einfach in das Zielformular einfügen und dort per *Align* an einem der Ränder andocken. Meistens dürfte es jedoch praktischer sein, das Panel in ein extra als Andockstelle vorgesehenes Platzhalter-Panel einzufügen, denn mit dieser Voraussetzung lässt es sich einfach realisieren, dass zwei Panels nebeneinander angedockt werden und die ihnen zur Verfügung stehenden Bereiche mit einer *TSplitter*-Komponente verschoben werden können.

Eine vererbbare Formularklasse für andockbare Fenster *R47*

Die Formulare für TreeExplorer und Übersichtsfenster des TreeDesigners bauen beide auf dem Basisformular *TDockableForm* auf (Unit *DockFormBase*), das die Grundfunktion

des manuellen Dockings mit folgenden Voraussetzungen implementiert: Als Andockstellen Hilfe von Platzhalter-Komponenten verwendet und der Fensterinhalt, der angedockt werden soll, muss durch eine einzige Komponente definiert sein, deren Kindelemente automatisch mit angedockt werden. In jeder der Platzhalter-Komponenten darf außerdem zur selben Zeit immer nur eine Komponente angedockt sein. Es folgt die öffentliche Schnittstelle von *TDockableForm*:

```
type
  TDockableForm = class(TForm)
    ... nicht-öffentliche Elemente weggelassen ...
  public
    property ManualDockTarget: TWinControl
       read FManualDockTarget write FManualDockTarget;
    property OnManualDock: TNotifyEvent
       read FOnManualDock write FOnManualDock;
    procedure ManualDockWithoutVCL(Control: TControl; Dock: Boolean);
    function IsDocked: Boolean;
    procedure RestoreDockStatus(IniPath: String);
    procedure SaveDockStatus(IniPath: String);
  end;
```

Bevor das Docking funktioniert, muss *ManualDockTarget* auf ein Komponente gesetzt werden, die als Andockstelle fungieren soll. Dann kann mit der Methode *ManualDockWithoutVCL* die Komponente *Control* je nach Wert des *Dock*-Parameters an- und abgedockt werden. Alle weiteren Properties und Methoden von *TDockableForm* sind optional und werden später bei Gelegenheit erläutert.

Die folgenden Quelltext-Auszüge zeigen die Verwendung von *TDockableForm* im TreeDesigner. Zunächst muss für jedes andockbare Fenster ein eigenes *ManualDockTarget* initialisiert werden. Der TreeDesigner stellt dafür auf der rechten Seite seines Hauptfensters zwei Panels namens *ManualDockPanel1* und *ManualDockPanel2* zur Verfügung (diese sind zur Entwurfszeit nicht zu sehen, da sie erst zur Laufzeit erzeugt werden; mehr dazu erfahren Sie am Schluss dieses Kapitels). Die Initialisierung der *ManualDockTarget*-Properties der beiden Hilfsfenster läuft dann wie folgt ab:

```
procedure TMainForm.InitToolWindows;
// aufgerufen vor Application.Run in TMainForm.BeforeRun
begin
  OverviewForm.ManualDockTarget := ManualDockPanel1;
  TreeExplorerForm.ManualDockTarget := ManualDockPanel2;
```

Bei dieser Gelegenheit können auch gleich die Ereignisse der *DockableForms* mit einer Methode des Hauptformulars verknüpft werden:

```
  TreeExplorerForm.OnManualDock := CheckDockPanelVisibility;
  OverviewForm.OnManualDock := CheckDockPanelVisibility;
  ... { auf der CD folgt hier u.a. noch das Setzen der Fensterposition }
end;
```

Bevor wir zum Zweck dieser Methoden kommen, sehen wir uns den wichtigsten Methodenaufruf an, den der Methode *ManualDockWithoutVCL*. Der TreeDesigner macht das Docking für den Benutzer nutzbar, indem er in den Popup-Menüs von Übersichts- und Baumansichtsfenster den Punkt MANUELLES DOCKING anbietet, der mit einer *Checked*-Markierung versehen wird, falls das jeweilige Fenster gerade angedockt ist. Die Bearbeitung dieser Menüpunkte läuft in beiden Hilfsfenstern gleich ab, als Beispiel sei hier die des Übersichtsfensters gezeigt:

```
procedure TOverviewForm.ManualDockingClick(Sender: TObject);
begin
  ManualDockWithoutVCL(Panel1, not IsDocked);
  ManualDocking.Checked := IsDocked;
end;
```

Dabei ist *Panel1* das Panel innerhalb des Übersichtsfensters, das alle Komponenten enthält, die beim Docken verschoben werden sollen. *IsDocked* ist die in *TDockableForm* deklarierte Methode, die besagt, ob die Komponente gerade angedockt ist oder nicht.

Die Methode *ManualDockWithoutVCL* erledigt nun, wie wir später noch sehen, die Verschiebung der *Panel1*-Komponente in das vorher eingestellte *ManualDockTarget* oder – beim Abdocken – zurück vom *ManualDockTarget* in den Rahmen des Hilfsfensters. Zusätzlich sind beim An- und Abdocken aber noch einige Aktionen notwendig, bei denen die Hilfe des Hauptfensters benötigt wird. Damit das Hauptfenster beim jedem An- und Abdockvorgang seine eigenen Aktionen durchführen kann, stellt *TDockableForm* das Ereignis *OnManualDock* zur Verfügung, das im TreeDesigner wie schon gezeigt mit der Methode *CheckDockPanelVisibility* des Hauptformulars verknüpft ist.

Diese Methode ist dafür verantwortlich, die Andockstelle und die daran angeschlossene *TSplitter*-Komponente am rechten Fensterrand ganz zu verstecken, falls keine Fenster angedockt sind. Außerdem sollen die Platzhalter-Komponenten *ManualDockPanel1* und *ManualDockPanel2* nur angezeigt werden, wenn darin ein Control angedockt ist. Schließlich befindet sich im TreeDesigner zwischen den beiden Platzhalter-Komponenten noch ein horizontaler Splitter. Dieser soll nur angezeigt werden, wenn *beide* Platzhalter mit angedockten Komponenten belegt sind:

```
procedure TMainForm.CheckDockPanelVisibility(DockedControl: TObject);
var
  Visible: Boolean;
begin
  Visible := (ManualDockPanel1.ControlCount = 1) or
             (ManualDockPanel2.ControlCount = 1);

  ToolwinDockSite.Width := 160;
  ToolwinDockSite.Visible := Visible;
  RightSplitter.Visible := Visible;
  ManualDockPanel1.Visible := ManualDockPanel1.ControlCount > 0;
  SplitterBetweenManualDockPanels.Visible :=
```

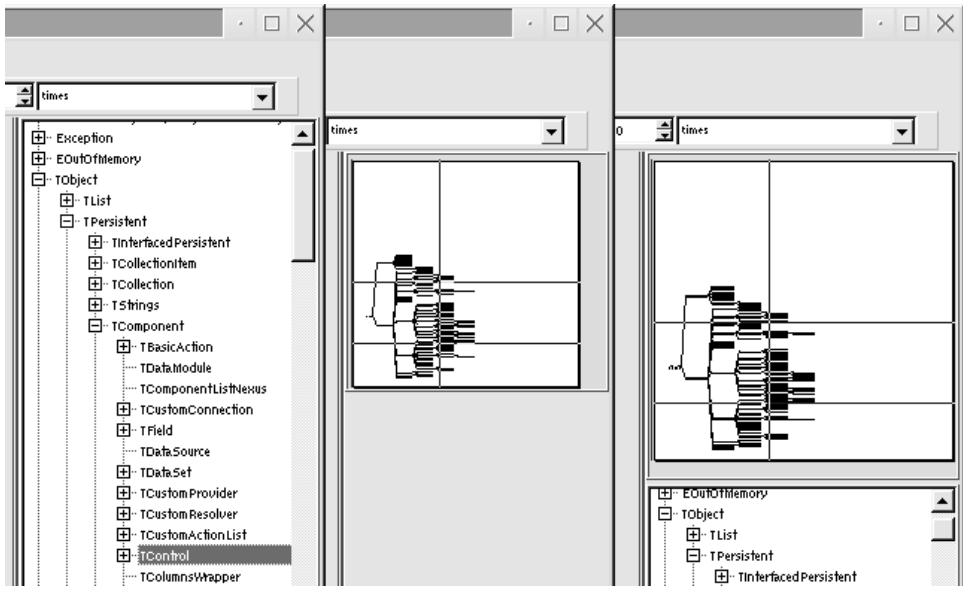


Abbildung 6.13: Die Komponente ToolwinDockSite im TreeDesigner-Hauptformular kann zur Laufzeit auf drei verschiedene Weisen mit angedockten Komponenten belegt werden; die Aufteilung zwischen zwei Komponenten wie die Breite des Andockbereichs kann mit jeweils einem Splitter verändert werden.

```
      (ManualDockPanel1.ControlCount > 0) and
      (ManualDockPanel2.ControlCount > 0);
    // Wichtig, damit der Splitter unter dem oberen Panel erscheint,
    // statt sinnloserweise darüber:
    SplitterBetweenManualDockPanels.
      SetBounds(0, ManualDockPanel1.Height + 10, 5, 5);
    ManualDockPanel2.Visible := ManualDockPanel2.ControlCount > 0;
  end;
```

Ohne diese Methode würde beispielsweise, wenn beide Hilfsfenster angedockt sind, beim Abdocken des oberen Fensters eine Lücke entstehen.

Speichern der Docking-Einstellungen *R123*

Die bisher noch nicht verwendeten Methoden von *TDockableForm* sorgen dafür, dass eine Anwendung den Status der andockbaren Fenster beim Beenden der Anwendung speichern und beim nächsten Start wiederherstellen kann. Der TreeDesigner etwa ruft, wenn das Hauptformular geschlossen wird (*OnClose*), die *SaveDockStatus*-Methoden seiner beiden Hilfsfenster auf und übergibt ihnen *IniPath*, die komplette Pfadangabe seiner Konfigurationsdatei:

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  OverviewForm.SaveDockStatus(IniPath);
  TreeExplorerForm.SaveDockStatus(IniPath);
  ...
```

Entsprechend wird beim Start der Anwendung *RestoreDockStatus* aufgerufen. Der TreeDesigner erledigt dies in der Methode *PlaceToolwindow*, die für beide Hilfsfenster je einmal angewendet wird. Für den Fall, dass noch kein Dock-Status gespeichert wurde, positioniert sie vorher die Hilfsfenster an den Rand des Hauptfensters unter der ControlBar (sonst würden die Hilfsfenster beim ersten Start des TreeDesigners eventuell ziemlich unordentlich über den Bildschirm verteilt erscheinen):

```
procedure PlaceToolwindow(Toolwin: TDockableForm;
  PlaceBelow: TForm = nil);
begin
  Toolwin.Left := Self.BoundsRect.Right-Toolwin.Width;
  if PlaceBelow <> nil then
    Toolwin.Top := PlaceBelow.BoundsRect.Bottom
  else Toolwin.Top :=
    ClientToScreen(ControlBar1.BoundsRect.BottomRight).y;
  Toolwin.RestoreDockStatus(IniPath);
end;
```

Implementierung des manuellen Dockings

Zum Abschluss sei noch kurz die Implementierung der Methode *ManualDockWithoutVCL* gezeigt. Sie stellt zunächst sicher, dass *ManualDockTarget* initialisiert wurde und der Parameter *Control* gesetzt ist und ein Element der *DockableForm* selbst darstellt. Ist dies nicht der Fall, bricht sie mit einer Exception ab, deren Meldungstext in den Stringkonstanten *rcstr_...* definiert ist.

Außer dem Ändern des *Parent*-Controls versteckt die Methode beim Docken auch noch das Hilfsfenster selbst, das ja nach dem Auslagern seines Inhalts nur noch als leere Hülle am Bildschirm erscheinen würde. Der aktuelle Docking-Status wird in den privaten Variablen *FDockedControlName* und *FDockedWithoutVCL* gespeichert (diese beiden Variablen sind es übrigens, durch die *SaveDockStatus* den Docking-Status speichern kann):

```
procedure TDockableForm.ManualDockWithoutVCL(Control: TControl;
                                             Dock: Boolean);
begin
{$B-}
  if (Control = nil) or (Control.Owner <> self) then
    raise EWrongControl.Create(rcstr_WrongControlError);
  if (ManualDockTarget = nil) then
    raise ENoTargetSet.Create(rcstr_NoTargetSetError);
  if Dock then begin
```

```
      // Wenn manuelles Docking verwendet werden soll,
      // muss "ManualDockTarget" von außen gesetzt werden.
      Control.Parent := ManualDockTarget;
      FDockedControlName := Control.Name;
      Visible := False;
    end else begin
      Control.Parent := self;
      FDockedControlName := '';
      Visible := True;
    end;
    FDockedWithoutVCL := Dock;
    if Assigned(FOnManualDock) then
      FOnManualDock(Control)
  end;
```

6.8.3 Interaktion mit Grafikobjekten

Die einzelnen Grafikobjekte im TreeDesigner sollen noch zwei weitere Fähigkeiten erhalten, die bisher nur ganzen Komponenten zuteil wurden: Jedes Grafikobjekt soll ein Popup-Menü erhalten und die Farben der Objekte sollen per Drag&Drop geändert werden können.

Popup-Menüs für ganze Komponenten sowie Drag&Drop zwischen ganzen Komponenten sind durch die automatische Verhaltensweise der CLX sehr einfach zu implementieren. Schwieriger ist es, innerhalb einer Komponente zwischen verschiedenen Objekten zu unterscheiden. Hier kommen die Parameter *X* und *Y* ins Spiel, die sowohl beim Ereignis *OnDragOver/OnDragDrop* als auch bei *OnMouseDown* Auskunft über die Mausposition geben.

In beiden Fällen läuft die Feststellung des unter der Maus befindlichen Objekts in denselben Schritten ab:

- Grundsätzlich genügt es, die Flächen aller Objekte daraufhin zu überprüfen, ob sie die Mausposition enthalten. Damit das oberste Objekt gewählt wird, wenn sich mehrere Objekte an der Mausposition befinden, müssen die Objekte von oben nach unten durchsucht werden.
- Wenn Sie mit einem virtuellen Koordinatensystem arbeiten, müssen Sie vorher noch die Geräteeinheiten, in denen *X* und *Y* angegeben sind, in virtuelle Einheiten umrechnen.

Im TreeDesigner sehen diese beiden Schritte wie folgt aus:

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  MapMousePos(x, y);
```

```
  MouseObject := Document.GetObjectByPoint(Point(X, Y),
                                           GetGripSize, ResizeHandle);
  ...
```

Die Umrechnung der Koordinaten mit *MapMousePos* wird in Kapitel 6.6.2 erläutert, die Methode *TGraphicDoc.GetObjectByPoint* in Kapitel 6.4.2.

Popup-Menüs für Grafikobjekte *R60*

Im Fall der Popup-Menüs kommt noch eine weitere Umwandlung von Koordinaten hinzu: Das Popup-Menü soll an der Mausposition erscheinen, tut dies aber nicht automatisch, sondern verlangt die Koordinaten als Parameter der Methode *Popup*. Die Mausposition, die relativ zur linken oberen Fensterecke angegeben ist, muss daher in Koordinaten relativ zur linken oberen Bildschirmecke umgewandelt werden, was Sie mit der Methode *ClientToScreen* erreichen.

Hinweis: In der erste CLX-Version liefert *ClientToScreen* falsche Ergebnisse, wenn ein Koordinate innerhalb einer ScrollBox umgerechnet werden soll und deren Scrollbars nicht auf der Nullposition stehen. Eine Untersuchung des Quelltexts zeigt, dass die Scrollposition doppelt einberechnet wird. Im unten gezeigten Code wird dies provisorisch korrigiert, allerdings dürfte dieser Ausgleich zu falschen Koordinaten führen, sobald Borland den Fehler innerhalb der CLX korrigiert hat. Wenn das der Fall ist, kompilieren Sie den TreeDesigner ohne das Compilersymbol *CLXClientToScreenWorkaround* neu (siehe Datei *TDDefines.pas*).

Das folgende Listing zeigt den Ausschnitt der Methode *TDocumentForm.PaintBoxMouseDown*, die für die Popup-Menüs zuständig ist (ein anderer Teil dieser TreeDesigner-Methode ist in Kapitel 6.4.1 gezeigt). *PaintBoxMouseDown* führt die beschriebenen Umwandlungen durch und testet danach, ob die rechte Maustaste gedrückt wurde. Ist das der Fall, gibt es zwei Möglichkeiten:

- Die Maustaste wurde in einem freien Bereich der Zeichenfläche geklickt, wodurch das Popup-Menü der Paintbox (*PaintBoxPopup*) aufgerufen wird.
- Oder die Maustaste hat ein Grafikobjekt getroffen, wodurch das Menü *ObjectPopup* aktiviert wird.

Beide Menüs werden wie jedes automatisch aktivierte Popup-Menü zur Entwurfszeit zum Formular hinzugefügt, werden aber nicht automatisch von der CLX aufgerufen (das *AutoPopup*-Property der Menüs ist *False*, das *PopupMenu*-Property der Paintbox bleibt leer). Beide Menüs werden auch in gleicher Weise aufgerufen, und zwar über die Methode *TPopupMenu.Popup*:

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
```

```
  PopupPos: TPoint;
begin
  PopupPos.x := x; PopupPos.y := y;
  { Objekt an Position feststellen }
      MapMousePos(x, y);
      MouseObject := Document.GetObjectByPoint(Point(X, Y),
                                               GetGripSize, ResizeHandle);
  { rechte Maustaste abfangen }
  if Button = mbRight then begin
{$ifdef CLXClientToScreenWorkaround}
    PopupPos.X := PopupPos.X - Paintbox.Left;
    PopupPos.Y := PopupPos.Y - Paintbox.Top;
{$endif}
    PopupPos := PaintBox.ClientToScreen(PopupPos);
    if Assigned(MouseObject) then
      ObjectPopup.Popup(PopupPos.X, PopupPos.Y)
    else PaintBoxPopup.Popup(PopupPos.X, PopupPos.Y);
  end;
  ...
```

Die Verknüpfung der einzelnen Menüpunkte mit Methoden läuft genauso ab wie bei jedem anderen Popup- und Hauptmenü. Der Inhalt beider Menüs besteht aus Funktionen, die im Laufe des sechsten Kapitels bereits auf andere Weise aufgerufen wurden *Markierte löschen* oder *Nach hinten schieben* für Tastaturbefehle), und aus spezielleren Methoden, die Sie im vollständigen Quelltext auf der CD finden. Das Popup-Menü der Grafikobjekte bietet beispielsweise Befehle zum Löschen von Verbindungen zu anderen Objekten und zum Anordnen der verbundenen Grafikelemente.

6.8.4 Drag&Drop

Ein Funktionsbereich, in dem die Laufzeit-Typinformationen von Object Pascal voll zur Entfaltung kommen können, ist das Drag&Drop (Ziehen und Ablegen). Bereits die Klasse *TControl* enthält alle für Drag&Drop notwendigen Properties und Methoden.

Funktionsweise

ber die Properties können Sie bereits im Objektinspektor nahezu beliebige Komponenten als *Quelle* einer Drag-Operationen bestimmen. Um Komponenten als Ablage*ziel* zu definieren, müssen Sie eine Zeile Code für das Ereignis *OnDragOver* schreiben. Um die Drop-Operation durchzuführen, bearbeiten Sie schließlich das Ereignis *OnDragDrop*.

Die folgende Tabelle fasst die beim Drag&Drop mitwirkenden Properties, Events und Methoden zusammen:

Elementart	*TControl*-Element	Aufgabe
Event	OnDragOver	feststellen, ob eine Komponente das Ablegen gestattet (Parameter: *Sender*, *Source*, *X*, *Y*, *State*, *Accept*).
Event	OnDragDrop	benachrichtigt eine Komponente davon, dass der Benutzer etwas auf ihr abgelegt hat (Parameter: *Sender*, *Source*, *X*, *Y*).
Event	OnEndDrag	benachrichtigt ein Kontrollelement davon, dass ein bei ihm begonnener Drag&Drop-Vorgang beendet ist, erfolgreich oder erfolglos (Parameter: *Sender*, *Target*, *X*, *Y*).
Property	DragMode	wählt zwischen automatischem und manuellem Modus (*dmManual* oder *dmAutomatic*).
Property	DragCursor	gibt eine Mauszeigerform für die Drag-Operationen an, die bei dieser Komponente beginnen. Diese Form wird nur angezeigt, wenn der Mauszeiger sich über einem möglichen Ablageziel befindet.
Methode	BeginDrag	starten das Drag&Drop im Modus *dmManual*.
Methode	EndDrag	bricht das Drag&Drop in beiden Modi ab.

Bevor wir zu einer genaueren Erklärung der Properties und Methoden sowie zu einem Beispiel kommen, sehen wir uns den allgemeinen Ablauf des Drag&Drop an.

Starten des Drag&Drop

Eine Drag&Drop-Operation beginnt damit, dass der Benutzer eine Maustaste auf einem Objekt drückt, das er verschieben will.

Für die Anwendung beginnt die Operation bei einer Komponente. Wenn Sie das Property *DragMode* dieser Komponente auf *dmAutomatic* gesetzt haben, startet die CLX den Vorgang automatisch, sobald der Benutzer mit der linken Maustaste auf diese Komponente klickt, und verhindert, dass das sonst übliche Maus-Event *OnMouseDown* an das Formular gesendet wird. Wollen Sie dieses Event nicht missen, müssen Sie den Vorgang manuell starten (der Modus *dmManual* ist die Voreinstellung). In diesem Fall steht es Ihnen frei, auch andere Events als *OnMouseDown* (theoretisch sogar Tastatureingaben oder Timer-Events) als Startsignal zu verwenden.

Der Inhalt des Drag&Drop

Wichtig ist dabei, dass normalerweise nicht die Quellkomponente selbst verschoben wird. Wenn Sie die Drag&Drop-Events genauer betrachten, erkennen Sie, dass dadurch eigentlich gar nichts verschoben wird, denn die Events *OnDragOver* und *OnDragDrop* kennen keinen Parameter »zu verschiebendes Objekt«, sondern nur den Parameter *Source*, der die Quellkomponente angibt, die irgendetwas an eine andere Komponente senden soll, und eine Zielkomponente, auf der dieses »Etwas« abgelegt werden soll

(diese Zielkomponente ist in den Ereignissen *OnDragOver* und *OnDragDrop* im Parameter *Sender* angegeben, denn von ihr gehen diese Ereignisse aus).

Die CLX übernimmt also nur die Vermittlung: Sie teilt der Zielkomponente mit, dass sie etwas von der Quellkomponente empfangen soll. Es liegt daraufhin in der Hand der Zielkomponente (bzw. der Methode, die deren *OnDragDrop*-Ereignis bearbeitet), sich die passenden Informationen selbst vom Parameter *Source* zu holen.

Diese Vermittlung der CLX ist so flexibel, dass Sie Drag&Drop-Operationen zu den verschiedensten Zwecken einsetzen können:

- Die einfachste Art des Drag&Drop ist das Bewegen von Dateien zwischen zwei Orten. Was mit den Dateien geschieht, hängt jedoch von weiteren Umständen ab, z.B. von der Art des Zielfensters oder von eventuell gedrückten Steuertasten der Tastatur: Verschieben, Kopieren, Löschen sind die einfachsten Drag&Drop-Dateioperationen, Übergabe eines Dokuments an eine Anwendung ist eine weitere Möglichkeit.
- Ähnlich wie bei Dateibewegungen können auch Einträge aus Listenfenstern und anderen Quellen leicht per Drag&Drop verschoben oder kopiert werden (zum Drag&Drop innerhalb einer ListBox siehe Kapitel 1.8.3, zu dem innerhalb eines IconViews siehe ***R54***).
- Manchmal werden jedoch keine Objekte kopiert, sondern bestimmte Eigenschaften verändert. Der TreeDesigner verwendet nur diese Möglichkeit des Drag&Drop: Sie können Farben aus der Farbpalette auf die Grafikobjekte ziehen.
- In der abstraktesten Form kann Drag&Drop zu jeglicher Nachrichtenübermittlung zwischen einer vom Anwender ausgewählten Quelle und einem ebensolchen Ziel verwendet werden. Damit ist beispielsweise die Synchronisation zwischen zwei Fenster(bereichen) möglich – beziehen sich beispielsweise zwei Fenster auf dieselbe Information (bzw. dasselbe Dokument), zeigen aber zwei verschiedene Bereiche daraus auf verschiedene Arten an, könnten Sie die aktuelle Position aus dem einen in das andere Fenster ziehen, so dass dann beide Fenster dieselben Bereiche zeigen, aber weiterhin auf zwei verschiedene Arten. (So geht beispielsweise der Execution Trace Analyzer aus einer mir vor scheinbar langer Zeit begegneten OS/2-Entwicklungsumgebung von IBM vor).

Drag

Nach dem »Greifen« des Objekts mit der Maus bewegt der Benutzer dieses an seinen Zielort. Solange der Drag&Drop-Vorgang noch nicht abgeschlossen ist, zeigt die Form des Mauszeigers dem Anwender an, ob das Objekt an der Stelle, an der sich die Maus gerade befindet, abgelegt werden kann.

Für die Kylix-Anwendung besteht der Ziehvorgang aus einer Reihe von *OnDragOver*-Events. Mit diesen fragt die CLX das Formular, ob die Komponente, die sich unter der Maus befindet, das Objekt von der Quellkomponente annehmen kann. Per Voreinstellung geben alle CLX-Komponenten eine negative Antwort. Dies führt dazu, dass die CLX den Mauszeiger in ein Verbotssymbol umwandelt.

Damit eine Komponente als Drop-Ziel dienen kann, müssen Sie eine Methode zum *OnDragOver*-Event dieser Komponente schreiben, in der Sie den Variablenparameter *Accept* auf *True* setzen. Falls es von der Mausposition abhängt, ob das Objekt akzeptiert wird oder nicht, können Sie zusätzlich die Parameter *X* und *Y* abfragen, die die Mausposition angeben. Die CLX startet nach jeder Mausbewegung eine neue Abfrage mit *OnDragOver*, auch wenn der Mauszeiger dadurch innerhalb derselben Komponente bleibt.

Wenn Sie mit *Accept=True* signalisieren, dass das Ablegen des Objekts erlaubt ist, verleiht die CLX dem Mauszeiger die Form, die Sie im *DragCursor* der Quellkomponente angegeben haben. Zum sehr selten benötigten *OnDragDrop*-Parameter *State* sei an dieser Stelle auf die Online-Hilfe verwiesen.

Drop

Wenn der Benutzer die Maustaste loslässt, erfahren sowohl Quell- als auch Zielkomponente davon. Für die Quellkomponente sendet die CLX dem Formular ein *OnDragEnd*-Ereignis, für die Zielkomponente das Ereignis *OnDragDrop*. Für beide Komponenten kann das Formular nun beliebige Aktionen durchführen. Wenn es sich bei beiden Komponenten um *TListBox*-Komponenten handelt, kann es beispielsweise einen Eintrag zwischen beiden Listen verschieben.

Ein Beispielprogramm für Drag&Drop-Varianten

Die Möglichkeiten des Drag&Drop sind so vielfältig, dass ein spezielles Demoprogramm sie am besten demonstrieren kann. Das Programm DragDropDemo, dessen Formular in Abbildung 6.14 gezeigt ist, enthält drei verschiedene Quell- und eine größere Sammlung von Zielkomponenten für Drag&Drop. Bei den meisten der Quellkomponenten wurde das *DragMode*-Property auf *dmAutomatic* gesetzt, lediglich ein Schalter bleibt für später beschriebene Zwecke im manuellen Modus.

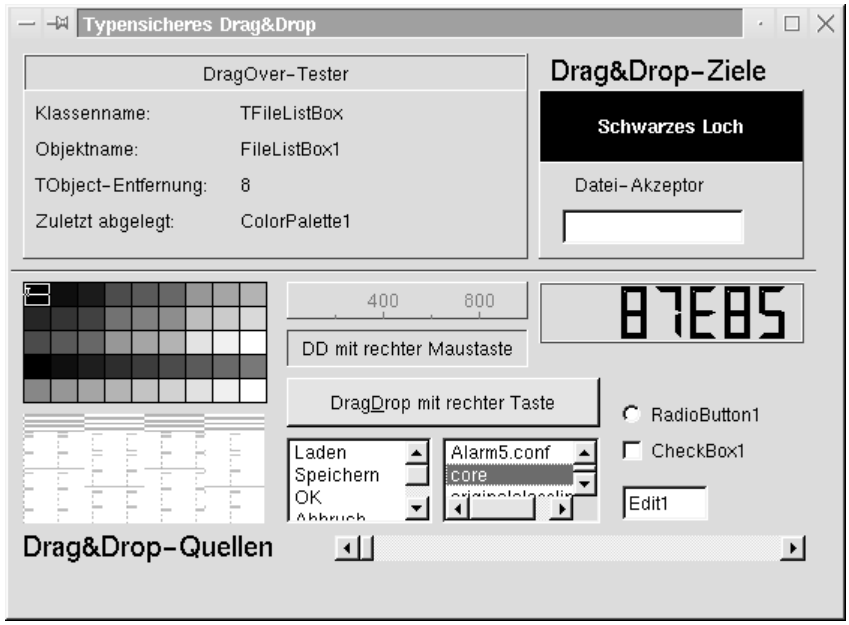


Abbildung 6.14: Ein Formular zur Demonstration verschiedener Arten des Drag&Drop

Zielkomponenten *R61*

Das mittlere Panel (*DragOverTester*) demonstriert, welche grundlegenden Informationen Sie von der Quellkomponente (*Source*) des Drag&Drops erhalten können. Zu jedem Objekt, das seinen Luftraum erreicht, zeigt es zunächst den Klassennamen an. Da es außerdem voraussetzen kann, dass nur Komponenten gezogen werden, kann es die gezogene Komponente *Source* auch in *TComponent* umwandeln und den Namen der Komponente anzeigen. Die Ausgabe der Informationen findet beim *OnDragOver*-Event statt:

```
procedure TForm1.DragOverTesterDragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  LabelKlassenName.Caption := Source.ClassType.ClassName;
  LabelObjektName.Caption := (Source as TComponent).Name;
  LabelEntfernung.Caption :=
    IntToStr(BaumEbene(Source.ClassType));{ siehe Kapitel 2.3.6 }
  Accept := True;
end;
```

Durch das Setzen von *Active* auf *True* akzeptiert die Komponente alle Objekte. Legt der Benutzer ein solches ab, zeigt *DragOverTester* den Namen dieser Komponente in der letzten Zeile an:

```
procedure TForm1.DragOverTesterDragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  LabelZuletzt.Caption := (Source as TComponent).Name;
end;
```

Das Editierfeld mit der Beschriftung *Datei-Akzeptor* ist wählerischer und akzeptiert das Drag&Drop nur, wenn es von einem Objekt der Klasse *TFileListBox* stammt. Wenn der Benutzer die Maustaste loslässt, zeigt es den Dateinamen, den die Dateilistbox im Property *FileName* aufbewahrt. Um aus den vielen möglichen Quellkomponenten nur die mit dem Typ *TFileListBox* auszuwählen, verwendet sie ebenfalls die Laufzeit-Typinformationen:

```
procedure TForm1.FilesAkzeptorDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TFileListBox;
end;

procedure TForm1.FilesAkzeptorDragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  Edit2.Text := (Source as TFileListBox).FileName;
end;
```

Das Panel »Schwarzes Loch« demonstriert, wie wenig die Komponenten untereinander vor schlechtem Benehmen geschützt sind: Es bereitet ihm keine Probleme, einfach jede Komponente, die um Ablageerlaubnis bittet, an sich zu ziehen und von der Bildfläche verschwinden zu lassen (indem es die Komponente versteckt – ein Löschen mit *Free* würde zu schweren Fehlern führen, da die Drag&Drop-Logik der CLX nicht damit rechnet, dass die Drag-Quelle plötzlich verschwindet).

Manuelles Drag&Drop

Der Unterschied des manuellen Drag&Drop-Modus zum automatischen besteht lediglich darin, dass eine Komponente, deren *DragMode*-Property *dmManual* ist, den Drag&Drop-Vorgang starten muss, indem sie die Methode *StartDrag* aufruft. Der Ablauf der Aktion verändert sich im manuellen Modus nicht, d.h. dass die CLX alle Drag&Drop-Events erzeugt, die auch beim automatischen Ablauf auftreten.

Grundsätzlich können Sie *StartDrag* jederzeit aufrufen, auch wenn keine Maustaste gedrückt ist. Die CLX verfolgt den Lauf der Maus wie bei einem normalen Ziehvorgang, beendet diesen Vorgang jedoch erst dann, wenn eine Maustaste losgelassen wurde (oder Sie auch das Ende manuell setzen, indem Sie die Methode *EndDrag* aufrufen).

Drag&Drop mit der rechten Maustaste

Eine Anwendung für manuelles Starten von Drag&Drop ist beispielsweise, den Ziehvorgang mit der rechten Maustaste zuzulassen:

```
procedure TForm1.Button1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbRight then
    (Sender as TControl).BeginDrag(True);
end;
```

Der Parameter von *StartDrag* besagt, ob der Vorgang sofort gestartet werden soll (*True*) oder ob die CLX so lange warten soll, bis die Maus ein winziges Stück bewegt wurde.

Entsprechend wird im Beispielprogramm der Drag&Drop-Vorgang vom Editierfeld bei per Tastatureingabe gestartet, indem *BeginDrag* im *OnKeyPress*-Ereignis aufgerufen wird.

Zielgenaues Drag&Drop im TreeDesigner

Im TreeDesigner wird Drag&Drop auf eine etwas feinere Weise eingesetzt als in DragDropDemo: Farben können von der Farbpalette auf einzelne Grafikelemente auf der Zeichenfläche gezogen werden und die Frage, ob das »Loslassen« der Farbe erlaubt ist, entscheidet sich nicht allein anhand der Zielkomponente, sondern anhand der Position der Maus über der Zielkomponente. Hierfür muss wie schon in Kapitel 6.8.3 für die Popup-Menüs festgestellt werden, ob sich ein Grafikelement an der Mausposition befindet.

Dabei kommen erneut die Methoden *MapMousePos* und *GetObjectByPoint* zum Einsatz. Zunächst akzeptiert die Methode für das *OnDragOver*-Event der Paintbox das Drag&Drop nur, wenn sich ein Grafikobjekt an der Mausposition befindet:

```
procedure TDocumentForm.PaintBoxDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
var
  Obj: TGraphicElement;
begin
  Accept := False;
  MapMousePos(x, y);
  Obj := Document.GetObjectByPoint(Point(X, Y));
  if Assigned(Obj) and (Source is TColorPalette) then
    Accept := True;
end;
```

Die Form des Mauszeigers wird daher sofort angepasst, wenn Sie die Maus von der eien Fläche auf ein Grafikobjekt und umgekehrt bewegen.

Die Methode für das *OnDragDrop*-Event nimmt die mit der linken Maustaste selektierte Farbe (*PenColor*) von der Farbpalette an und setzt die Füllungsfarbe (*Brush.Color*) des Zielelements auf diese. Logischer wäre es natürlich, stattdessen *Pen.Color* zu setzen, des grafischen Effekts wegen wurde jedoch die Änderung der Füllungsfarbe vorgezogen.

```
procedure TDocumentForm.PaintBoxDragDrop(Sender, Source: TObject; X,
  Y: Integer);
var
  Obj: TGraphicElement;
begin
  MapMousePos(x, y);
  Obj := Document.GetObjectByPoint(Point(X, Y));
  if Assigned(Obj) then
    if Source is TColorPalette then
      Obj.Brush.Color := (Source as TColorPalette).PenColor;
      { Aufgrund eines in Kapitel 6.3.3 beschriebenen Mechanismus
        "merkt" das Grafikelement diesen Property-Schreibzugriff
        und kann die Neuausgabe am Bildschirm veranlassen sowie den
        Zähler für ungespeicherte Änderungen erhöhen. }
end;
```

6.9 XML

Im Zuge der zunehmenden Vernetzung der Computer ist in den letzten Jahren der Bedarf an einem flexiblen Datenformat, mit dem Daten problemlos auch zwischen den unterschiedlichsten Rechnerarchitekturen transportiert werden können, immer größer geworden. Diesem Bedarf trug die Entwicklung von XML Rechnung. XML ist zwar an sich noch kein Datenformat, sondern der Bezeichnung »eXtensible Markup Language« nach eher eine Sprache, aber auf Grundlage dieser Sprache lassen sich hervorragend Datenformate definieren, durch die sich beispielsweise das Problem der unterschiedlichen Dateiformate unterschiedlicher Anwendungen oder verschiedener Versionen einer Anwendungen lösen lässt.

Kylix kann zwar lokale Datenbank-Tabellen in XML-Dateien speichern, enthält in der ersten Version aber noch keine eingebaute Programmierschnittstelle für XML. Für die Programmbeispiele dieses Kapitels ist daher eine kleine Zusatzbibliothek erforderlich, die Sie auf der CD dieses Buchs finden und die eine provisorische Schnittstelle zu den XML-Klassen von Qt herstellt. Diese Schnittstelle gibt lediglich Zugriff auf die in den Beispielprogrammen benötigten XML-Methoden von Qt. Sie kann theoretisch zwar leicht erweitert werden, wird aber möglicherweise in zukünftigen Kylix-Versionen durch eine eingebaute Object-Pascal-XML-Schnittstelle überflüssig. Da sich aber sowohl die Qt-Klassen als auch die provisorische Schnittstellenbibliothek auf der CD

an den DOM-Standard (Document Object Model) halten, sollte der Beispiel-Code sich mit diesen zukünftigen, vollständigen Schnittstellen sehr gut vertragen und sich nach minimalen Änderungen übersetzen lassen.

- In Kapitel 6.9.1 geht es zunächst um den Aufbau von XML-Dateien und die Repräsentation eines XML-Dokuments in Object Pascal durch Klassen bzw. Interfaces, die sich an den DOM-Standard halten.
- Kapitel 6.9.2 beschreibt als kleines Beispiel eine Erweiterung des TreeDesigners, die ein beliebiges XML-Dokument einliest und in einer grafischen Baumstruktur ausgibt.
- Kapitel 6.9.3 befasst sich daraufhin mit dem schon erwähnten Problem der Dateiformate, die durch XML wesentlich benutzerfreundlicher gehandhabt werden können (Stichwort »Kompatibilität« zwischen verschiedenen Formatversionen).

6.9.1 XML-Grundlagen

XML-Dateien sind menschenlesbare Textdateien, die sowohl die eigentlichen Daten enthalten als auch so genanntes *Markup* in Form von *Tag*s. Das Markup dient dazu, die *Bedeutung* der Daten zu kennzeichnen. In einem Dateiformat *ohne* Markup könnte beispielsweise ein Grafikelement des TreeDesigners wie folgt gespeichert werden:

```
TAutoObject/300/380/10/120/1/0/16777215/0/0/20/Arial
```

Der TreeDesigner selbst könnte dieses Dateiformat ohne weiteres lesen, da er genau wüsste, dass das erste Datenelement der Zeile die Beschriftung des Grafikelements angibt, das zweite die X-Koordinate der linken oberen Ecke, das dritte die Y-Koordinate usw. Nach diesem Prinzip verfährt der TreeDesigner ja in der Tat beim Speichern der `tvf`-Dateien durch die in Kapitel 6.3.3 gezeigten Methoden. Da die `tvf`-Dateien binär sind, lassen sie sich lediglich nicht so leicht vom Menschen lesen wie die oben abgedruckte Textzeile.

Nehmen wir nun an, der TreeDesigner 3.5 speichert das oben in der einzeiligen Datei ohne Markup gezeigte Grafikelement in einer XML-Datei (Kapitel 6.9.3 zeigt die dafür verwendeten Methoden). Diese XML-Datei sähe dann wie folgt aus:

```
<TreeViewerDocument Version="10">
  <GraphicElement shapeText="TAutoObject" left="300" right="380" top="10"
   bottom="120" FShapeType="1" penColor="0" brushColor="16777215"
   fontColor="0" penWidht="0" fontHeight="20" fontName="Arial">
  </GraphicElement>
</TreeViewerDocument>
```

Es dürfte leicht einsehbar sein, dass es nicht mehr unbedingt den TreeDesigner dafür erfordert, um die obige Datei interpretieren zu können. Jedes andere Programm, das extdateien des obigen Formats in eine interne baumartige Datenstruktur überführen

kann, könnte mit diesen TreeDesigner-Daten umgehen; unabhängig davon, unter welchem Betriebssystem oder auf welcher Rechnerplattform das Programm läuft (das Textformat dieser Datei ist auch leicht über das Internet übertragbar). Insbesondere könnten zukünftige TreeDesigner-Versionen die Liste der Attribute eines Grafikelements problemlos erweitern. Jedes neue Attribut würde durch einen eigenen Namen erkennbar sein und die früheren Programmversionen könnten das Attribut einfach ignorieren. Liest dagegen eine neue Programmversion eine alte Datei mit »fehlenden« Attributen, so ist auch dies leicht zu erkennen und die neue Programmversion kann diese Attribute mit Standardwerten belegen.

Die Frage ist nur, ob es nicht sehr aufwändig für ein Programm ist, Textdateien im XML-Format einzulesen (zu parsen). Normalerweise wäre dies zu bejahen, doch gehört zum XML-Standard auch die Definition von Programmierschnittstellen für Parser und für eine baumartige Datenstruktur, durch die ein XML-Dokument im Programm repräsentiert werden kann. Mit Hilfe fertiger Bibliotheken, die sich an diese Standards halten, ist es dann nur noch ein sehr kleiner Aufwand für den Entwickler, XML-Dateien zu bearbeiten. Der Aufwand ist trotz der großen Vorteile des flexiblen Datenformats nicht einmal höher als der zum Programmieren der Lese- und Schreibmethoden für binäre Dateien.

Ein sehr auffälliger äußerlicher Unterschied zwischen dem Beispiel-XML-Listing und der davor gezeigten einzeiligen Datenversion liegt wohl darin, dass sich die Größe der Daten durch das Markup vervielfacht, denn das Markup nimmt wesentlich mehr Platz ein als die eigentlichen Daten. Dies ist ein typisches Merkmal der meisten XML-Dateien und es wurde bei der Entwicklung von XML bewusst in Kauf genommen, da man aufgrund stetig wachsender Speicherkapazitäten und Datenübertragungsraten die Zeit für reif gehalten hat, mit dem knausrigen Zählen einzelner Bytes Schluss zu machen. Außerdem lässt sich eine XML-Datei hervorragend komprimieren, da sich der Text der Markups ständig wiederholt (in einem TreeDesigner-Dokument mit hundert Grafikelementen würde der größte Teil der Datei durch die hundert Vorkommnisse der Markup-Strings *ShapeText*, *left*, *right*, *top* etc. verbraucht werden).

Hinweis: Die von Kylix verwendeten Formulardateien enthalten schon seit Delphi 1 ebenfalls eine Art Markup: So werden etwa nicht einfach die Inhalte der Property-Einstellungen des Formulars und seiner Komponenten in die Datei geschrieben, sondern immer auch die zugehörigen Property-Namen. Diese Formulardateien werden zwar bisher nicht im XML-Format geschrieben, aber für den Datenaustausch zwischen den verschiedenen Delphi-Versionen und nun auch von Delphi nach Kylix bewährt sich das flexible Markup-Prinzip hervorragend (lediglich die in Kylix vorgenommenen Erweiterungen am Dateiformat bedeuten eine Einschränkung, da die Kylix-Formulardateien erst ab Delphi 6 wieder gelesen werden können).

ypdefinitionen und Style-Sheets

Die oben gezeigte TreeDesigner-XML-Datei verwendet zahlreiche selbst definierte Markup-Strings: Alle Zeichenketten, die nicht in Anführungszeichen eingeschlossen sind, gehören zum Markup. Die Sprache XML definiert lediglich die allgemeine Struktur der XML-Datei, die Markup-Strings werden von der Anwendung festgelegt. Wenn wir diese Markup-Strings mit den vordefinierten Wörtern z. B. der Sprache Object Pascal vergleichen, können wir jede XML-Datei als in einer speziellen »Sprache« geschrieben betrachten. Die Sprache XML ist damit ein Hilfsmittel zur Definition spezieller Sprachen, daher wird XML auch als *Metasprache* bezeichnet.

Zur »TreeDesigner-Grafikdokumentbeschreibungs-Sprache« gehören beispielsweise die Elementtypen *TreeViewerDocument* und *GraphicElement*, wobei ein *GraphicElement* ber ein vordefiniertes Set von Attributen verfügt, wie *shapeText*, *brushColor* und *fontHeight*. Die Struktur des TreeDesigner-XML-Datei wird implizit durch den reeDesigner definiert. Als Metasprache sieht XML jedoch auch konkrete Formen vor, mit denen die Struktur eines Dokuments sich schriftlich definieren lässt. Auch wenn wir in diesem Kapitel keinen Gebrauch der folgenden beiden Möglichkeiten machen werden, sind sie doch eine Erwähnung wert:

- Die bisher provisorisch standardisierten Document Type Definitions (DTD) und die neueren XML-Schemas sind Dateien, in denen auf vom XML-Standard definierte Weise genau festgelegt wird, welche Elemente in einem XML-Dokument vorkommen dürfen, welche Elemente in welchen anderen Elementen enthalten sein dürfen (in der TreeDesigner-XML-Datei dürfen beispielsweise *Edge*-Elemente, welche die Verbindungen zwischen Grafikobjekten angeben, in *GraphicElement*-Elementen enthalten sein, aber nicht umgekehrt), welche Attribute erlaubt sind und welche Typen diese haben. Mit Hilfe einer solchen DTD bzw. eines solchen Schemas kann der Parser ein Dokument beim Einlesen auf die so genannte *Gültigkeit* überprüfen, noch bevor die Anwendung über das DOM auf den Dokumentinhalt zugreift. Wie beim Parsen eines Object-Pascal-Quelltextes durch den Kylix-Compiler können auch beim Parsen des XML-Dokuments verschiedene Fehler vorkommen, etwa Attributwerte, die einen falschen Typ aufweisen.

  XML definiert zwei Kriterien, nach denen sich ein XML-Dokument auf Korrektheit überprüfen lässt: Wenn es die Regeln einer DTD/eines XML Schemas komplett erfüllt, wird es wie erwähnt als gültig (*valid*) bezeichnet. Gibt es keine DTD, aber erfüllt das Dokument einige Grundregeln, kann es noch als wohlgeformt (*well-formed*) gelten. Zu diesen Grundregeln gehört etwa, dass jedes Element durch korrespondierende Tags in spitzen Klammern eingeschlossen sein muss (z. B. `<Zahl>1</Zahl>`) oder dass sich Elemente bei der Schachtelung nicht überschneiden, wie es etwa bei den Elementen von HTML-Dokumenten möglich ist.

- Auch ein *Style-Sheet* befasst sich mit den verschiedenen in einem XML-Dokument vorkommenden Elementtypen und Attributen, jedoch aus einer anderen Perspektive: Das Style-Sheet definiert, wie die Daten des Dokuments darzustellen sind. Von HTML ist mit den Cascading Style Sheets (CSS) eine ähnliche Technik bekannt: Dort werden Absatzformate definiert, die jeweils für einen bestimmten Typ von Absatz gelten. Der Typ eines Absatzes wird auch in HTML durch Tags angegeben, so ist etwa *H1* der Typ des Absatzes `<H1>Überschrift</H1>`. Einen ähnlichen Zweck verfolgen auch die XML-Stylesheets, mit dem Unterschied, dass sie viel allgemeiner sind und etwa dazu dienen können, ein und dasselbe XML-Dokument einmal als Tabelle von Zahlen anzuzeigen und ein anderes Mal (unter Verwendung eines zweiten Style-Sheets) als Balkengrafik.

Neben DTD, XML-Schema und Style-Sheets umfasst XML noch zahlreiche andere Technologien und wird auch noch fortlaufend weiter entwickelt. Für weitere Informationen sei hier auf die umfangreiche Spezial-Literatur zu XML verwiesen. Der Bereich des XML-Standards, der in diesem Buch am meisten Aufmerksamkeit geschenkt bekommt, ist das DOM.

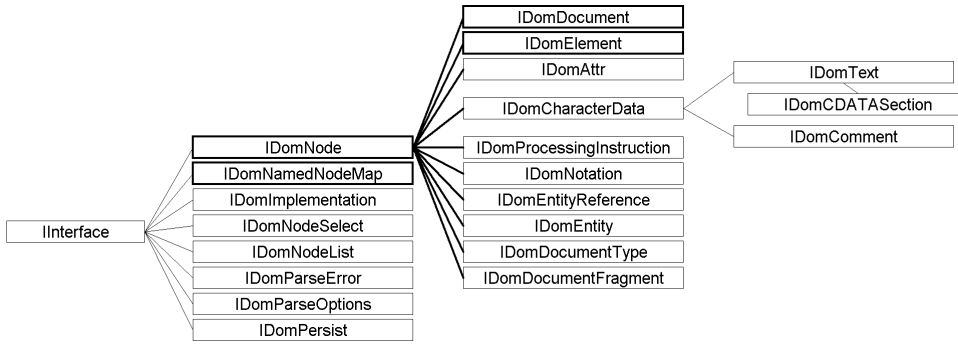


Abbildung 6.15: Die DOM 2.0 Interface-Hierarchie von Delphi 6. Die hervorgehobenen Schnittstellen werden im TreeDesigner verwendet und unter Linux soweit erforderlich mit Hilfe von Qt-Objekten nachgebildet.

XML-DOM

Das *Document Object Model* (DOM) beschreibt eine Programmierschnittstelle, über die ein Programm auf alle Elemente und Daten eines XML-Dokuments zugreifen kann. Auf diese Weise kann ein Programm beispielsweise mit einer einzigen Anweisung ein theoretisch beliebig großes XML-Dokument einlesen und danach über DOM-Klassen auf dessen einzelne Bestandteile zugreifen. Das Parsen der XML-Textdatei liegt alleine in der Verantwortung der verwendeten Bibliothek. Die Klassen, die im DOM-Standard die Namen *Node*, *Element*, *Document*, *Attr* usw. tragen, werden in Delphi 6 durch Interfaces repräsentiert, in deren Name noch ein »IDom« vorangestellt wurde, also etwa

IDomNode, *IDomElement* etc. Bei den Klassen von Qt wird entsprechend ein Präfix von QDom« verwendet. Wir verwenden unter Kylix für die Anbindung an die Qt die gleiche Schnittstelle wie Delphi, also Interfaces mit dem Präfix »IDom«.

Dokumentation

Unter Kylix bieten sich zwei grundlegende Online-Dokumentationen zum DOM an:

- Die Qt-Dokumentation enthält auch eine Beschreibung aller Methoden der *QDom...*-Klassen. Wenn Sie etwa Informationen zum *nodeValue*-»Property« eines *Node*-Objekts suchen, schlagen Sie unter *QDomNode* nach. Sie finden dort die Funktion *nodeValue*, die den Wert zurückliefert, und die Methode *setNodeValue*, die ihn setzt.
- Ultimative Referenz über den DOM-Standard finden Sie auf der Website des W3-Konsortiums unter `www.w3.org`. Da der hier ständig neue Dokumente hinzu kommen, ist es nicht möglich, in diesem Buch eine konkrete, aktuelle Adresse anzugeben. Die in der Abbildung erwähnten Interfaces von DOM 2.0 finden Sie beispielsweise unter `www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html`.

Wenn Sie weitergehende XML-Funktionen verwenden wollen, können Ihnen diese Referenzen allerdings wenig helfen, hier wäre dann ein spezielles, einführendes Buch zu empfehlen, das einen Überblick über den gesamten XML-Bereich gibt und die Grundprinzipien der verschiedenen Technologien erläutert.

Grundlegende Elemente von IDomNode

Wie Abbildung 6.15 zeigt, sind die DOM-Interfaces in einer Vererbungshierarchie angeordnet, deren Wurzel von *IDomNode* gebildet wird. *IDomNode* repräsentiert einen Knoten in der Baumstruktur eines XML-Dokuments und bietet allgemeine Properties an, um durch diese Baumstruktur zu navigieren:

Property	**Bedeutung**
parentNode: IDomNode	gibt den übergeordneten Knoten (Elternknoten) des aktuellen Knotens zurück.
firstChild: IDomNode	gibt den ersten Unterknoten (das erste Kind) des aktuellen Knotens an.
nextSibling: IDomNode	gibt den »Geschwisterknoten« an, der dem aktuellen Knoten in der Kinderliste des Elternknotens folgt.
wnerDocument: IDomDocument	weist auf einen Knoten, der das gesamte Dokument repräsentiert.

Mit diesen Methoden kann man, wenn man nur einen Knoten des Dokuments kennt, alle anderen Knoten erfragen: Es genügt, das Dokument-Objekt (*ownerDocument*) in Erfahrung zu bringen, bei diesem mit *firstChild* das erste Kind zu ermitteln und von diesem ausgehend mit *nextSibling* so lange das nächste Kind zu erfragen, bis *nextSibling nil* zurückgibt. Dies wird rekursiv für jeden Knoten wiederholt, der weitere Kinder hat[4], ein Beispiel dafür finden Sie in Kapitel 6.9.2.

Die bisher genannten Properties sind nur lesbar, jedoch gibt es unter anderem die folgenden Methoden, um die XML-Datenstruktur zu verändern:

Methode	**Wirkung**
function insertBefore(const newChild, refChild: IDomNode): IDomNode;	fügt den Knoten *newChild* vor dem existierenden Knoten *refChild* ein. (Entsprechend gibt es auch eine *insertAfter*-Methode.)
function replaceChild(const newChild, oldChild: IDomNode): IDomNode;	ersetzt den existierenden Knoten *oldChild* durch *newChild*.
function removeChild(const childNode: IDomNode): IDomNode;	entfernt den existierenden Knoten *childNode*.
function appendChild(const newChild: IDomNode): IDomNode;	hängt einen neuen Knoten als letzten Knoten an die Liste der Kind-Knoten an.

Die Rückgabewerte der Funktionen sind im Fehlerfall *nil*, ansonsten wird eine der als Parameter angegebenen *IDomNode*-Referenzen zurückgegeben, was aber im Beispiel-Code dieses Kapitels ignoriert wird.

Schließlich hat jeder Knoten einer DOM-Struktur auch noch bis zu vier wichtige Daten, die über weitere Properties von *IDomNode* angesprochen werden:

Property	**Bedeutung**
nodeType	enthält eine Kennung für die verschiedenen Knotentypen. Zu jedem Knotentyp definiert das DOM eine spezielle Klasse, von *Node* abgeleitete Klasse und entsprechend gibt es auch ein spezielles, von *IDomNode* abgeleitetes Interface. Einige dieser speziellen Klassen werden im folgenden Abschnitt behandelt, dabei wird auch erwähnt, durch welche *nodeType*-Kennung sich diese Klassen auszeichnen.

4. Der Knoten *ownerDocument* hat immer nur ein einziges Kind, das *root*-Element.

Property	Bedeutung
nodeName	Alle im eingangs gezeigten TreeDesigner-Beispiel vorkommenden Markup-Strings sind Knotennamen, die über *nodeName* abgefragt werden können. Der Knotenname kann je nach Art des Knotens auf verschiedene Weisen im XML-Dokument erscheinen: Als Name eines Elements wie *TreeViewerDocument* und *ELEMENT*, als Name eines Attributs wie *ShapeText* oder *Left* oder auf viele andere Weisen, für die auf die spezielle XML-Literatur verwiesen sei. Was jedoch hier noch erwähnt werden soll, ist, dass nicht alle Knoten eines XML-Dokuments über einen individuellen Namen verfügen. So haben beispielsweise alle Text-Knoten den Namen *#text*, alle Kommentarknoten den Namen *#comment* und der Dokumentknoten ist immer mit *#document* benannt.
nodeValue	Auch der Wert dieses Properties hängt von der konkreten Art des Knotens ab. Viele Knotentypen haben gar keinen Wert, bei ihnen ist *nodeValue* immer *nil*, so z.B. bei Element- und Dokument-Knoten. Bei Attribut-Knoten hingegen entspricht der Wert jeweils dem Datenelement, welches im TreeDesigner-Beispiellisting oben in Anführungszeichen angegeben wurde.
text	XML-Elemente können außer Attributen und weiteren Elementen auch noch Text enthalten, der über das Property *text* abgefragt werden kann. Dieser Text wird im XML-Dokument beispielsweise wie folgt angegeben: `<Element>Text des Elements</Element>`.

IDomDocument

Von dem, was *IDomDocument* den von *IDomNode* geerbten Elementen hinzufügt, benötigen wir in diesem Kapitel nur eine Methode und ein Property:

```
// Aus der Deklaration von IDomDocument:
function createElement(const tagName: DomString): IDomElement;
property documentElement: IDomElement
```

Mit *createElement* wird ein neues Element erzeugt, das danach beispielsweise mit *IDomNode.appendChild* an ein bereits existierendes Element angefügt werden kann. Das neue Element wird schon von *createElement* mit einem Namen versehen, die weiteren Eigenschaften des Elements müssen Sie getrennt über die *IDomElement*-Schnittstelle des Elements setzen (z.B. den Text über das *Text*-Property und die Attribute über die im nächsten Abschnitt genannten Methoden). Viele weitere Methoden zum Erzeugen von Elementen spezieller Typen finden Sie außerdem in der Qt-Online-Dokumentation beschrieben.

Jedes XML-Dokument muss genau ein Wurzel-Element enthalten, in dem dann alle anderen Elemente enthalten sind. Dieses Element ist nicht das *IDomDocument*-Objekt selbst, sondern es wird über dessen Eigenschaft *documentElement* angegeben. Für jedes XML-Dokument gibt es also ein *IDomDocument*-Objekt und ein *IDomElement*, welches das Wurzel-Element repräsentiert. In diesem sind dann je nach Dokument weitere Elemente und Attribute enthalten. Abbildung 6.16 zeigt die sich hieraus ergebenden Besitzverhältnisse der DOM-Objekte.

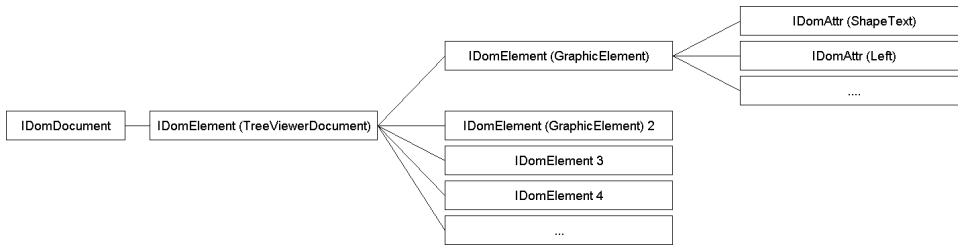


Abbildung 6.16: Die Besitzverhältnisse der DOM-Objekte am Beispiel eines TreeDesigner-Dokuments: Das Wurzelelement (TreeViewerDocument) ist in einem IDomDocument-Objekt enthalten; es enthält seinerseits die einzelnen Grafikobjekte als Unterelemente.

Hinweis: Vielleicht fragen Sie sich, warum es überhaupt ein separates *IDomDocument*-Objekt gibt, wenn doch dessen Inhalt in einem eigenständigen *IDomElement*-Objekt enthalten ist. Der Grund dafür liegt darin, dass *IDomDocument* einige weitere Daten enthält, die auch in der XML-Datei noch außerhalb des Wurzel-Elements angegeben werden. Dazu gehören beispielsweise Kommentare, Angabe der XML-Versionsnummer und die Verbindung des Dokuments mit einer separat gespeicherten DTD-Datei. Natürlich können Sie über das DOM auch auf diese Daten zugreifen, jedoch können diese weiter führenden Eigenschaften des *IDomDocument*-Objekts bzw. die darin enthaltenen Unter-Objekte nicht im Rahmen dieses Buches behandelt werden.

IDomElement

Der folgende Auszug aus der Deklaration von *IDomElement* zeigt einige der wesentlichen Eigenschaften, die ein XML-*Element* von anderen Knotentypen unterscheidet: Ein Tag, das über einen Namen verfügt (*tagName*, der allerdings auch über *IDomNode.nodeName* abgefragt werden kann), sowie eine Menge von Attributen, die über ihren Namen gesetzt und abgefragt werden können:

```
IDomElement = interface(IDomNode)
  ...
  function getAttribute(const name: DomString): Variant;
  procedure setAttribute(const name: DomString;value: Variant);overload;
  function Get_attributes: IDomNamedNodeMap;
  property tagName: DomString read get_tagName;
end;
```

Die von *Get_attributes* zurückgelieferte *IDomNamedNodeMap*-Schnittstelle wird uns in Kapitel 6.9.2 die Möglichkeit geben, alle Attribute abzufragen, ohne deren Namen zu kennen. Sie soll an dieser Stelle nicht weiter untersucht werden.

XML in Kylix

Während Delphi 6 für Windows es leicht hat, Anschluss an eine passende DOM-Bibliothek zu finden (Microsofts MSXML dürfte auf den meisten Systemen installiert sein und ist auch aufgrund ihrer COM-Schnittstellen von Delphi aus über Methoden und Interfaces sehr leicht zu verwenden), ist die Verwendung einer DOM-Programmierschnittstelle in Kylix bisher nicht vorgesehen: Weder werden mit Kylix Units mitgeliefert, die sich mit dem DOM befassen, noch ist unter Kylix ein Mechanismus verfügbar, mit dem der Anschluss an C++-Bibliotheken so leicht ist wie unter Windows mit Delphis COM-Typenbibliotheks-Importfunktion.

otzdem ist XML-Unterstützung in Kylix nicht fern, denn zu dem Füllhorn an Funktionen, welche die Qt-Bibliothek dem Programmierer bietet, gehören auch Klassen, die das DOM implementieren. Wie schon angekündigt, verwenden die Beispielmethoden des TreeDesigners eine Schnittstellenbibliothek, über die alle für diese Methoden notwendigen Qt-XML-Methoden aus Object Pascal heraus angesprochen werden können. Die Schnittstellenbibliothek heißt `libqtxml_wrapper.so` und soll hier nicht ausführlich behandelt werden, sondern als Provisorium angesehen werden, das durch eine vollständige DOM-Schnittstelle für Object Pascal ersetzt werden soll, sobald eine solche von Borland oder einem anderen Anbieter verfügbar ist. Die Bibliothek stellt C-Funktionen der folgenden Art zur Verfügung:

```
QDomNode *QtDomNode_appendChild(QDomNode *self, QDomNode &newChild);
```

Diese C-Funktionen rufen die Qt-Methoden auf, in diesem Beispiel die Methode *appendChild* der Klasse *QDomNode*:

```
QDomNode *QtDomNode_appendChild(QDomNode *self, QDomNode &newChild)
{
  return CreatePointerFromObject(self->appendChild(newChild));
}
```

Der Aufruf von *CreatePointerFromObject* ist eine Besonderheit dieser Bibliothek, die hier nicht von Bedeutung ist. Ansonsten verläuft die Anbindung von C++ zu Object Pascal wie in Kapitel 5.4 beschrieben.

XML über Interfaces

Eine Ebene über `libqtxml_wrapper.so` angesiedelt ist die Object-Pascal-Unit *QXML-Bridge*. Sie importiert zunächst die C-Funktionen auf die ebenfalls schon in Kapitel 5.4 beschriebene Weise:

```
const
  qtxmlintfname = 'libqtxml_wrapper.so.0.1';

function QtDomNode_appendChild(self: QObjectH; newChild: QObjectH)
                              : QObjectH; external qtxmlintfname;
```

Darüber hinaus stellt sie jedoch Klassen und Interfaces zur Verfügung, die im Programm so verwendet werden können wie die mit Delphi 5 und 6 verwendbaren DOM-Interfaces, die beispielsweise die XML-Bibliothek von Microsoft verwenden. Der mit XML arbeitende TreeDesigner-Code kann daher ohne Änderung sowohl unter Delphi als auch unter Kylix übersetzt werden. Alle XML-Funktionen des TreeDesigners stehen damit sowohl unter Windows als auch unter Linux zur Verfügung.

Um diese Kompatibilität herzustellen, wurden zunächst die Interface-Definitionen der Delphi-Unit auf das Notwendige gekürzt und in die Unit *QXMLBridge* übernommen. Der oben als Beispiel angegebenen Funktion *QDomNode::appendChild* entspricht etwa die folgende Deklaration des Interfaces *IDomNode*:

```
type
  IDomNode = interface(IUnknown)
    ['{2BF4C0E1-096E-11D4-83DA-00C04F60B2DD}']
    { Property Acessors }
    ...
    { Methods }
    function appendChild(const newChild: IDomNode): IDomNode; cdecl;
    ...
  end;
```

Randbemerkung: Diese Übernahme von Interfaces aus einer Delphi-Unit stellt keine Copyright- oder Lizenz-Verletzung dar, da die DOM-Schnittstelle zur allgemeinen Verwendung bestimmt ist und die übernommenen Object-Pascal-Interface-Deklarationen maschinell von Delphi erzeugt werden können, beispielsweise über einen Import der Typenbibliothek der `msxml.dll`. Bei den übernommenen Interface-Deklarationen handelt es sich also um eine maschinell angefertigte Übersetzung eines offenen Standards.

Was leider nicht maschinell angefertigt werden kann, sind die Klassen, die unter Linux zur Implementierung der Interfaces notwendig sind, weshalb die Implementierung auf das notwendige Minimum beschränkt wurde. Unsere Beispielmethode *appendChild* etwa wird wie folgt in *QXMLBridge* implementiert:

```
type
  TQtDomNode = class(TInterfacedObject, IDomNode)
  private
    FQtObject: QObjectH;
  public
    function appendChild(const newChild: IDomNode): IDomNode; cdecl;
    ...
  end;

implementation

function TQtDomNode.appendChild(const newChild: IDomNode): IDomNode;
```

```
begin
  Result := GetNodeWrapper(
     QtDomNode_appendChild(self.QtObject, newChild.QtObject));
end;
```

Die Funktion *GetNodeWrapper* ist wieder eine Eigenheit dieser Unit, sie gibt ein *IDomNode*-Interface für das von *QtDomNode_appendChild* zurückgelieferte *QDOMNode*-Objekt zurück (dafür muss im Hintergrund ein Object-Pascal-Objekt erzeugt werden, dessen Klasse die *IDomNode*-Schnittstelle unterstützt, mit anderen Worten: ein neues Objekt der Klasse *TQtDomNode*).

Das folgende Kapitel stellt alle benötigten Interfaces und Methoden vor, wie sie sowohl von *QXMLBridge* als auch von den mit Delphi 6 mitgelieferten Bibliotheken zur Verfügung gestellt werden.

6.9.2 Ein grafischer XML-Betrachter

Aufgrund ihrer Baumstruktur und der Tatsache, dass sie in reinem Textformat mit all den Tags noch nicht gerade bequem zu lesen sind, bieten sich XML-Dateien dem TreeDesigner als ein neues Betätigungsfeld an. Über den Menüpunkt DATEI | XML | BELIEBIGE XML-DATEI ALS GRAFIK ANZEIGEN können Sie eine Datei Ihrer Wahl einlesen. Die später abgedruckte Funktion *CreateTreeDesignerTree* durchläuft den gesamten DOM-Baum des Dokuments und erzeugt für jeden Element-Knoten ein eigenes Grafikelement, wobei natürlich auch die Verbindungen von jedem Element zu seinen Kindelementen in die Grafik übernommen werden. Anschließend werden diese neuen Grafikelemente mit der TreeDesigner-Methode *ArrangeFromObject* zu einem sichtbaren Baum angeordnet (siehe Abbildung 6.17). *ArrangeFromObject* wird auch zur Anordnung der Klassenhierarchiebäume verwendet und soll *nicht* das Thema dieses Kapitels sein.

Eine neue Grafikelement-Klasse für den TreeDesigner

Damit sich mit der XML-Grafik auch etwas anfangen lässt, sollen für die einzelnen Elemente mehr Informationen dargestellt werden, als sich sinnvoll in einem normalen reeDesigner-Grafikelement darstellen lassen. Daher wurde eine neue Grafikelement-Klasse namens *TMultiLineElement* eingeführt, deren Deklaration Sie im Folgenden lesen können:

```
type
  TParagraphAlignment = (pfLeft, pfCenter, pfRight, pfBlock);
  TMultiLineElement = class(TGraphicElement)
  private
    FLongText: String;
    FParagraphAlignment: TParagraphAlignment;
    procedure SetParagraphAlignment(const Value: TParagraphAlignment);
    procedure SetLongText(const Value: String);
  public
```

```
    procedure LoadFromStream(Stream : TStream); override;
    procedure SaveToStream(Stream : TStream); override;
    procedure PaintText(CanvasRef : TVirtualCanvas); override;
    procedure AttrDialog; override;
    function CountNodeTextLines: integer; override;
    property LongText: String read FLongText write SetLongText;
    property ParagraphAlignment : TParagraphAlignment
       read FParagraphAlignment write SetParagraphAlignment;
  end;
```

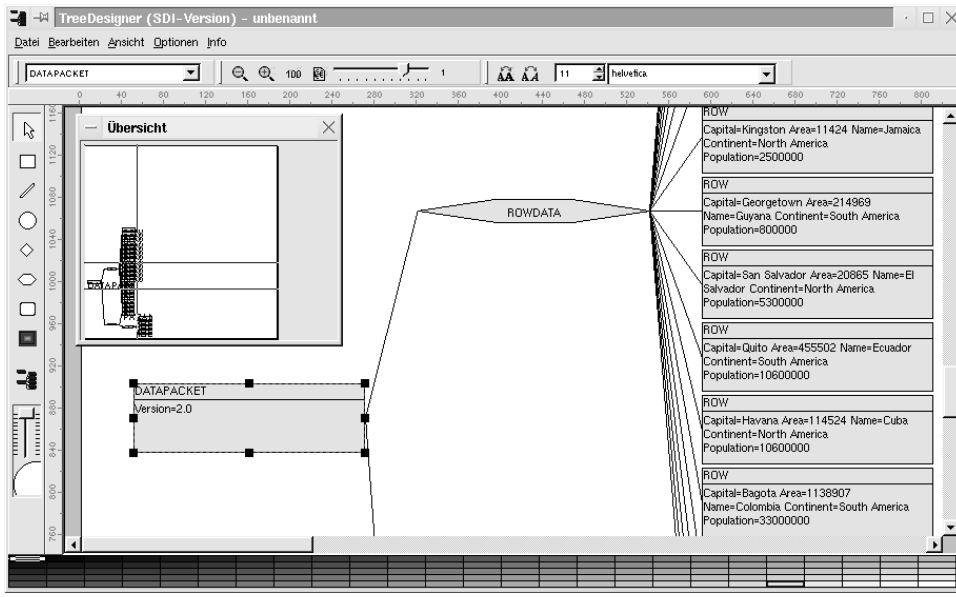


Abbildung 6.17: Die XML-Datenbanktabelle »country.xml« von dbExpress, dargestellt als Grafik im TreeDesigner

Entscheidende Neuerung dieser Klasse ist das Property *LongText*, in dem sich zusätzlich zu dem von *TGraphicElement* geerbten Text-Attribut ein längerer Text speichern lässt, der nicht in der Mitte des Elements angezeigt, sondern von links oben beginnend in mehrere Zeilen umgebrochen wird. Das Property *ParagraphAlignment* ist für zukünftige Erweiterungen vorgesehen und wird normalerweise auf *pfLeft* eingestellt, um den Text linksbündig darzustellen.

Um die korrekte Funktionsweise des Elements zu gewährleisten, überschreibt *TMultiLineElement* außerdem fünf virtuelle Methoden von *TGraphicElement*:

- *LoadFromStream* und *SaveToStream* speichern die beiden neu eingeführten Properties und sorgen dafür, dass die im TreeDesigner bearbeiteten XML-Grafiken in `tvf`-Dateien gespeichert werden können.

- *PaintText* ersetzt die von *TGraphicElement* bekannte zentrierte Textausgabe. Wenn *LongText* nur einen leeren String enthält, ruft *PaintText* die geerbte *PaintText*-Methode zur zentrierten Grafikausgabe aus. Andernfalls sorgt es selbst für die Ausgabe von *LongText* mit automatischem Zeilenumbruch. Wenn zusätzlich zu *LongText* auch das von *TGraphicElement* »Standardtext«-Property *ShapeText* gesetzt ist, schreibt *PaintText* diesen in die erste Zeile, *LongText* in die folgenden Zeilen und trennt die beiden noch durch eine Linie, wie sie etwa in Abbildung 6.17 in einigen Elementen sichtbar ist.

  Wie *PaintText* im Gesamtvorgang des Zeichnens eingesetzt wird, können Sie dem Listing der Methode *TGraphicElement.Paint* aus Kapitel 6.5.2 entnehmen. Da dort bereits ein rechteckiger Clipping-Bereich für die Textausgabe eingestellt wird, kann auch der mehrzeilige Text der neuen Klasse nicht über die Grenzen des Elements hinauswachsen (der automatische Zeilenumbruch sollte dies horizontal sowieso verhindern, doch für das Abschneiden der letzten Zeile, die evtl. nicht mehr ganz in das Element hineinpasst, ist das Clipping ideal).

- *AttrDialog* ruft eine neue Version des Attributdialogs auf, in dem Sie den vom Element angezeigten Text editieren können (aufzurufen über das Popup-Menü des Grafikelements, siehe Kapitel 6.8.3).

- *CountNodeTextLines* wird für die automatische Baum-Anordnungsfunktion benötigt, um die Zeilenzahl eines Elements zu bestimmen. Sie ist für zukünftige Erweiterungen vorgesehen, da bisher immer nur Platz für drei Zeilen reserviert wird (wenn Sie dies ändern möchten, ändern Sie einfach diese Zahl in der Implementierung dieser Funktion!).

Dies soll zur Vorstellung der neuen Klasse genügen. Im Folgenden wird *TMultiLineElement* als gegebenes Werkzeug verwendet, die Implementierung der Methoden von *TMultiLineElement* kann aus Platzgründen nicht hier abgedruckt werden.

Darstellung der XML-Knoten durch Grafikelemente

Durch eine kleine Vorbereitung kann die Umsetzung der XML-Knoten in Grafikelemente erheblich vereinfacht werden. Und zwar enthält die Unit *TDXML* eine Art Übersetzungstabelle, die jedem Typ von XML-Knoten eine bestimmte grafische Form und eine Farbe zuordnet sowie außerdem einen String, der in der Grafik als Typbezeichner erscheinen soll.

```
type
  // Liste der im DOM möglichen Knotentypen:
  TNodeType = (ntInvalid, ntElement, ntAttribute, ntText, ntCData,
    ntEntityRef, ntEntity, ntProcessingInst, ntComment, ntDocument,
    ntDocumentType, ntDocumentFrag, ntNotation);
  // Grafik-Attribute, die mit den Knotentypen verknüpft werden können:
  TGraphicAttributes = record
```

```
      TypeText: string;
      ShapeType: TShapeType;
      Color: TColor;
    end;

// "Übersetzungstabelle" für die Knotentypen:
const
  GraphicAttributesTable : array[ntInvalid..ntNotation]
    of TGraphicAttributes = (
    (TypeText: 'ungültig'; ShapeType: stRect; Color: clWhite),
    (TypeText: 'Element'; ShapeType: stHexagon; Color: clYellow),
    (TypeText: 'Attribut'; ShapeType: stRect; Color: clBlue),
    (TypeText: 'Text'; ShapeType: stRect; Color: clGreen),
    (TypeText: 'CData'; ShapeType: stRect; Color: clWhite),
    (TypeText: 'Entity-Referenz'; ShapeType: stEllipse; Color: clRed),
    ...
  );
```

So werden beispielsweise normale *Element*-Knoten mit dem String »Element« versehen, als Sechseck gezeichnet und mit gelber Hintergrundfarbe versehen. Durch Ändern der Einträge in dieser Tabelle können Sie den Stil der XML-Grafik leicht Ihren eigenen Wünschen anpassen. Wenn Sie weitere »Spalten« in die Tabelle einfügen wollen, erweitern Sie zuerst den Typ *TGraphicAttributes* und berücksichtigen das Attribut auch in der Funktion *CreateTreeDesginerTree* nach dem Muster der schon vordefinierten Attribute.

Hinweis: Bei der automatischen Erzeugung der Grafikelemente wird der in der Tabelle angegebene *ShapeType* ignoriert, falls das Grafikelement mit einem *LongText* versehen wird. Da dieser Text linksbündig im Rechteck des Elements angezeigt wird, würden sich nämlich alle Grafikformen außer dem Rechteck mit dem Text überschneiden.

Erzeugen des XML-Dokument-Objekts

In Kapitel 6.9.1 wurde bisher angenommen, dass bereits irgendein *IDomNode*-Objekt verfügbar war, von dem aus man sich die anderen *IDomNode*- und *IDomElement*-Objekte liefern lassen konnte. Die Erzeugung des ersten DOM-Objekts, des Dokument-Objekts, ist noch offen. Da wir hier mit Qt arbeiten, ist naheliegenderweise ein C++-Create-Konstruktor für die Zurücklieferung eines *QDomDocument*-Objekts zuständig. Da der XML-Code im TreeDesigner jedoch auch mit Delphi für Windows funktionieren soll, wurde ein Weg zur Erzeugung eines Dokumentobjekts definiert, der mit der Vorgehensweise von Delphi für Windows kompatibel ist. Und dort wird eine Klasse *CoDomDocument* definiert, deren *Create*-Methode (kein Konstruktor, sondern eine Klassenfunktion!) ein Dokumentobjekt zurückliefert. Der folgende Auszug aus der Unit

QXMLBridge zeigt diese Klasse und die Implementation der Methode *Create*, die die C-Funktion *QtDomDocument_create* aufruft, welche ihrerseits den C++-Konstruktor in Aktion treten lässt:

```
type
  CoDomDocument = class
    class function Create: IDomDocument;
  end;

class function CoDomDocument.Create: IDomDocument;
begin
  Result := GetDocWrapper(QtDomDocument_create);
end;
```

Die Funktion *GetDocWrapper* dient wie schon *GetNodeWrapper* in Kapitel 6.9.1 zur Umwandlung des *QDomDocument*-Zeigers in eine Interface-Variable durch Erzeugung eines Object-Pascal-»Wrapper«-Objekts.

Einlesen eines XML-Dokuments

Nachdem ein *IDomDocument*-Objekt erzeugt wurde, kann über eine spezielle Methode dieses Objekts eine XML-Datei geparst werden. Diese Methode wurde in Kapitel 6.9.1 noch nicht erwähnt, weil sie implementationsabhängig ist. Die im Folgenden aufgerufene Methode *load* führt über die Unit *QXMLBridge* und `libqtxml_wrapper.so` schließlich zum Aufruf von *QDomDocument::load*. In Delphi für Windows steht bei Verwendung der `msxml.dll` die gleiche Methode zur Verfügung, so dass die Kompatibilität gewahrt bleibt:

```
function CreateTreeDesignerTree(FileName: String;
            Doc: TGraphicDoc): TMultiLineElement;
// liefert das Grafikelement für das XML-Wurzelelement zurück.
var
  XMLDoc: IxmlDOMDocument;
  DocElem: IxmlDOMElement;
begin
  XMLDoc := CoDOMDocument.Create;
  XMLDoc.load(FileName);
  DocElem := XMLDoc.documentElement;
  if Assigned(DocElem) then
    Result := CreateSubTree(Doc, DocElem, nil);
end;
```

CreateTreeDesignerTree hat die Aufgabe, die Datei *FileName* einzulesen und für die darin enthaltenen XML-Knoten Grafikelemente in das Grafikdokument *Doc* einzufügen. Das Grafikelement, welches dem XML-Wurzel-Element entspricht, wird als Funktionsergebnis zurückgeliefert.

Nach Einlesen der Datei mit *load* überprüft die Methode zunächst, ob das ein Dokument-Element vorhanden ist (was natürlich nicht der Fall ist, wenn eine Nicht-XML-Datei geladen wird). Dann ruft sie die entscheidende Funktion *CreateSubTree* auf.

Hinweis: Da XMLDoc ein *Interface* ist, brauchen wir uns um die Freigabe nicht zu kümmern. Der vom Compiler erzeugte Code gibt das dahinter stehende Objekt automatisch am Ende der Methode frei (bzw. er verringert den Referenzzähler des Objekts, was wiederum dazu führt, dass dieses sich selbst freigibt, wenn der Zähler bei Null angekommen ist, siehe hierzu auch Kapitel 2.7).

Ein Durchlauf durch den DOM-Baum

Da *CreateSubTree* sich rekursiv selbst aufruft, wird diese Funktion letztlich für *jeden* Knoten im DOM-Baum aufgerufen. Sie erwartet als Parameter das Grafikdokument, den DOM-Knoten, der die Wurzel des zu behandelnden Teilbaums darstellt, und das Grafikelement, mit dem die neuen Grafikelemente verbunden werden sollen (dies ist also das zum Elternknoten gehörende Grafikelement):

```
function CreateSubTree(Doc: TGraphicDoc; Node: IDomNode;
    ParentGraphicElem: TMultiLineElement): TMultiLineElement;
```

CreateSubTree erzeugt immer nur ein einzelnes Grafikelement, nämlich das Grafikelement für den Wurzel-Knoten des Teilbaums (*Node*). Alle weiteren Grafikelemente werden in den rekursiven Aufrufen von *CreateSubTree* erzeugt.

Für die Erzeugung des Grafikelements für *Node* wird zunächst der Typ des Elements abgefragt: Bei einem Elementknoten werden die Werte der Attribute in *TMultiLineElement.LongText* gespeichert, bei Textknoten stattdessen der Textinhalt des Knotens (*Text*-Property). Bei allen anderen Knotentypen soll nur der Typbezeichner des Knotens (aus der oben gezeigten *GraphicAttributesTable*) im Grafikelement angezeigt werden (die Variable *ShortText* wird später an das *ShapeText*-Property des Elements übergeben).

Die im folgenden ersten Teil des Listings verwendeten DOM-Methoden/Properties sind *nodeType*, *tagName* und *Text*. Die Bezeichner *NODE_ELEMENT* und *NODE_TEXT* wurden der Kompatibilität mit Delphi wegen verwendet; sie stehen lediglich für die Typkonstanten »1« und »3«, entsprechend der weiter oben gezeigen Typdeklaration von *TNodeType*, aus der auch die anderen möglichen Knotentypen hervorgehen, die in *CreateSubTree* nicht gesondert berücksichtigt werden.

```
function CreateSubTree(Doc: TGraphicDoc; Node: IDomNode;
    ParentGraphicElem: TMultiLineElement): TMultiLineElement;
var
  ElementAttributes: TGraphicAttributes;
  Connection: TEdge;
  subNode: IDomNode;
  ShortText, LongText: string;
```

```
begin
  ElementAttributes := GraphicAttributesTable[TNodeType(Node.nodeType)];
  LongText := '';
  case Node.nodeType of
    NODE_ELEMENT: begin
      ShortText := (Node as IDomElement).tagName;
      LongText := GetAttributeString(Node as IDomElement); // s.u.
    end;
    NODE_TEXT: begin
      ShortText := ElementAttributes.TypeText;
      LongText := Node.Text;
    end;
    else ShortText := ElementAttributes.TypeText;
  end;
```

Nach diesem ersten Teil der Methode geht es mit der Erzeugung des *TMultiLineElement*-Objekts und seiner Verbindung zum Elternelement weiter. Hierzu werden Methoden und Properties verwendet, die teilweise in Kapitel 6 erläutert sind, da sie aber nichts mit dem Thema XML zu tun haben, sind sie im Folgenden ohne weitere Erläuterung aufgeführt:

```
  Result := TMultiLineElement.Create(Doc, Rect(0,0,0,0),
                ElementAttributes.ShapeType);
  // für langen Text muss die Grafikform ein Rechteck sein:
  if LongText <> '' then Result.ShapeType := stRect;
  Result.Text:=ShortText;
  Result.Brush.Color := ElementAttributes.Color;
  Result.LongText := LongText;
  if Assigned(ParentGraphicElem) then
    Connection:=TEdge.StandardConnection(ParentGraphicElem, Result);
  Doc.Add(Result); // Das fertige Grafikelement zum Dokument hinzufügen
```

Den Abschluss der Methode bildet der rekursive Aufruf von sich selbst für alle Unterknoten von *Node*. Dabei kommen die DOM-Methoden *firstChild* und *nextSibling* zum Einsatz:

```
  // Unterknoten des aktuellen XML-Knotens durchlaufen:
  subNode := Node.firstChild;
  while subNode <> nil do begin
    CreateSubTree(Doc, subNode, Result);
    subNode := subNode.nextSibling;
  end;
end;
```

Was in dem obigen Listing noch offen geblieben ist, ist die Funktion *CreateAttributeString*. Sie hat die Aufgabe, alle Attribute eines Elements zu erfragen und mit Attributname und -wert in einen String zu schreiben. Der Grund, warum diese Funktion hier gesondert und am Schluss behandelt wird ist der, dass sie ein weiteres, bisher nicht erwähntes DOM-Objekt benötigt, die *NamedNodeMap*. Aus dieser lassen sich mit mit

get_item die einzelnen Attribute abfragen. Jedes einzelne Attribut wird als *IDomNode* zurückgeliefert und gewährt über *get_nodeName* und *get_nodeValue* Zugriff auf Name und Wert des Attributs:

```
function CreateAttributeString(Node: IDomElement);
var
  AttrMap: IDomNamedNodeMap;
  AttrNode: IDomNode;
  i: integer;
begin
  Result := '';
  AttrMap := Node.get_attributes;
  for i := 0 to AttrMap.Get_length-1 do begin
    AttrNode := AttrMap.get_item(i);
    Result := Result + AttrNode.get_nodeName + '='
                + AttrNode.get_nodeValue+' ';
  end;
end;
```

6.9.3 XML-Dateiformate

Dieses Kapitel zeigt, wie eine Kylix-Anwendung Daten in einer XML-Datei selbst definierten Formats speichern kann. Als Beispiel dient der TreeDesigner, der normalerweise binäre `tvf`-Dateien erzeugt, aber über das Menü DATEI | XML seine Dokumente auch in XML-Dateien speichern und daraus wiederherstellen kann (das Lesen einer TreeDesigner-XML-Datei ist nicht zu verwechseln mit dem Einlesen einer beliebigen XML-Datei und der Anzeige dieser Datei als TreeDesigner-Grafik!)

Hinweis: Da der TreeDesigner bereits über eine komplette Speicherfunktion für Binär-Dateien verfügt, die allen Einsatzzwecken des TreeDesigners gerecht werden sollte, bleibt der XML-Speicherfunktionen nur der Demonstrationszweck. Es wurden daher nicht die gesamten binären Speichermethoden auch für XML bereitgestellt, sondern nur die Methoden zum Speichern einfacher Elemente der Klasse *TGraphicElement* ohne zusätzlichen Bildinhalt. Beim Speichern eines Dokuments mit anderen Grafikelementen oder Inhalten in einer XML-Datei werden diese zusätzlichen Inhalte ignoriert; beim Laden der XML-Datei erscheint dann ein entsprechend »reduziertes« Grafikdokument.

Selbst definierte XML-Dateiformate *R117*

Auf die Vorteile der Speicherung von Daten im XML-Format wurde schon in Kapitel 6.9.1 eingegangen; kurz zusammengefasst zeichnen sich XML-Dateien dadurch aus, dass sie

- menschenlesbar (also nicht binär) sind,
- sich aufgrund der Standardisierung von XML zwischen beliebigen Arten von Computern und zwischen allen denkbaren Arten von Anwendungen übertragen lassen,
- erweiterbar unter Beibehaltung von Aufwärts- und Abwärtskompatibilität des Datenformats sind.

Die Definition eigener Dateiformate kann ganz sorgfältig über eine der ebenfalls in Kapitel 6.9.1 erwähnten DTDs erfolgen, jedoch ist eine solche DTD im Fall des Tree Designers nicht notwendig. Die wichtigste Definition für das TreeDesigner-Dateiformat stellt daher die Festlegung der Namen für die XML-Elemente und die Attribute dieser Elemente dar. Sie erfolgt in der Unit *GrDoc* über einen Block von Konstanten. Im Programm-Code werden diese Namen nie direkt als Strings angegeben, sondern immer nur über Verwendung der Konstantenbezeichner wie *xmlAttrFontColor*. Dies stellt sicher, dass keine Tippfehler unbemerkt neue Namen kreieren, die dann nicht mehr mit dem anderen Code übereinstimmen (bei falsch geschriebenen Konstanten-Bezeichnern meldet ja der Compiler einen Fehler).

```
(******** Bezeichner für das TreeDesigner-XML-Dateiformat *********)
const
  (*** geschrieben von TGraphicDocument ***)
  xmlTypeGraphicDocument = 'TreeViewerDocument';
  xmlAttrDocVersion = 'Version';
  xmlGraphicElementOPClass = 'ObjectPascalClass';
  (*** geschrieben von TGraphicElement ***)
  xmlTypeGraphicElement = 'GraphicElement';
  xmlAttrShapeText = 'ShapeText';   xmlAttrFontName = 'FontName';
  xmlAttrLeft = 'Left';             xmlAttrRight = 'Right';
  xmlAttrTop = 'Top';               xmlAttrBottom = 'Bottom';
  xmlAttrShape = 'ShapeType';       xmlAttrPenColor = 'PenColor';
  xmlAttrBrushColor = 'BrushColor'; xmlAttrFontColor = 'FontColor';
  xmlAttrPenWidth = 'PenWidth';     xmlAttrFontHeight = 'FontHeight';
```

Abbildung 6.18 zeigt die grafisch geringfügig aufbereitete Darstellung einer TreeDesigner-XML-Datei; aus ihr können Sie die Anwendung der oben gezeigten Namen erkennen. Einige weitere Namen für die *Edge*-Elemente, die in der Abbildung auftauchen, wurden im obigen Listing der Übersichtlichkeit halber weggelassen; sie kommen auch in den folgenden Listings nicht vor.

Erzeugen einer XML-Datei

Da eine XML-Datei eine Textdatei ist, können Sie sie theoretisch auch wie eine normale extdatei erzeugen, z.B. über die Standard-Pascal-Dateioperationen. Sie müssten sich dann selber darum kümmern, dass die XML-Syntax, also beispielsweise die Klammeungen und die Positionen der Tags, mit dem XML-Standard konform gehen.

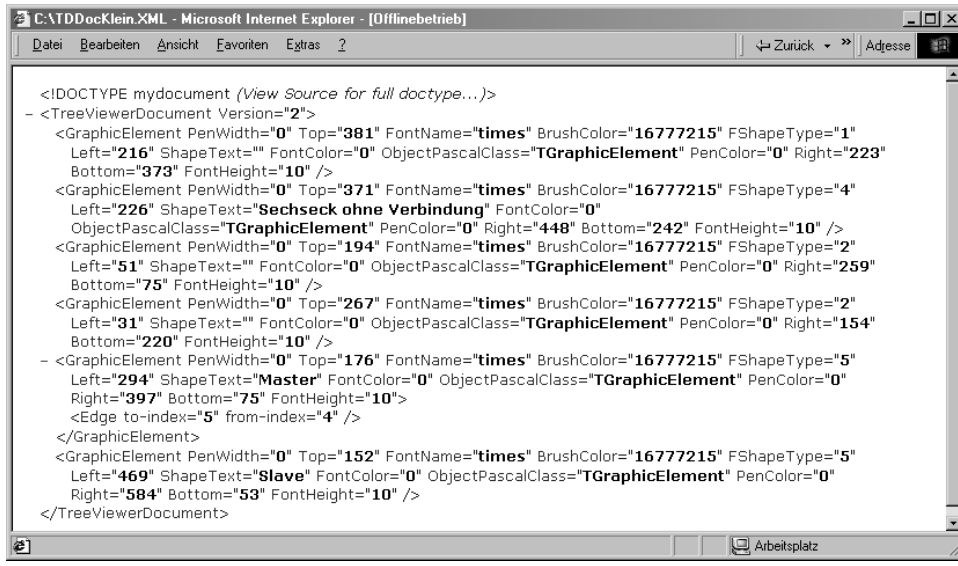


Abbildung 6.18: Um Verwirrung zu vermeiden, wird die TreeDesigner-XML-Datei nicht im TreeDesigner selbst als XML-Grafik angezeigt, sondern mit einem weit verbreiteten Windows-Browser, der die XML-Dateien auch schön anzeigen kann.

Dies entfällt, wenn Sie das XML-Dokument im Speicher mit Hilfe des bereits in Kapitel 6.9.2 verwendeten *Document*-Objekts aufbauen und dieses dann beauftragen, sich selbst in eine Datei zu schreiben.

Das Dokumentobjekt wird im TreeDesigner wie in Kapitel 6.9.2 wieder über einen Aufruf von *CoDOMDocument.Create* erzeugt. Dieses Dokument ist zunächst vollkommen leer, enthält also noch keinen XML-Elementknoten. Ein solcher wird nun in der folgenden Methode mit der schon in Kapitel 6.9.1 erwähnten Methode *createElement* erzeugt. Als Parameter ist der Tag-Name des Elements anzugeben, im TreeDesigner wird dafür eine der oben definierten Konstanten (*xmlTypeGraphicDocument*) verwendet. Das neu erzeugte Element muss nun mit *appendChild* an einen bestehenden Knoten angehängt werden, wobei zu Anfang ja nur das Dokumentobjekt selbst als Elternknoten in Frage kommt.

Das so vorbereitete Dokumentobjekt kann nun gefüllt werden, wofür im TreeDesigner die separate und später besprochene Methode *SaveToXML* zuständig ist. Wir beginnen hier mit einem Listing der Methode *SaveToXMLFile*, die das Dokument-Objekt erzeugt, *SaveToXML* aufruft und damit endet, das fertige Dokument mit der *save*-Methode zu speichern:

```
procedure TGraphicDoc.SaveToXMLFile(FileName: String;
  OnlyMarked: boolean); //OnlyMarked: nur die markierten Objekte speichern
var
```

```
  DocElem: IDomElement;
  n: IDomNode;
  Doc: IDomDocument;
begin
  Doc := CoDOMDocument.Create;
  DocElem := Doc.createElement(xmlTypeGraphicDocument);
  Doc.appendChild(DocElem);
  SaveToXML(DocElem, OnlyMarked);
  Doc.save(FileName);
end;
```

Hinweis: Auch die Save-Methode ist nicht im XML-Standard definiert, wird aber oben in einer zu `msxml.dll` unter Windows kompatiblen Weise aufgerufen. Die Implementierung dieser Methode unter Linux verwendet die Qt-Methode *QDomDocument::toString*, welche keine Datei erzeugt, sondern das Abbild einer »möglichen« Datei in einem String ablegt. Dieser String kann dann »per Hand« in eine Datei geschrieben werden. Diese Details sind auf der CD in der Methode *TQtDomDocument.save* (Unit *QXMLBridge*) zu finden.

Speichern der Grafikelementdaten in einem XML-Element

Oben wurde bereits die Methode *SaveToXML* aufgerufen, die das gesamte Grafikdokument als Unterbaum an ein bereits vorbereitetes XML-Element anhängen soll. Da das Grafikdokument aus seinen einzelnen Grafikelementen besteht, werden wir nun vom Kleinen zum Großen schreiten und mit der *SaveToXML*-Methode für ein einzelnes Grafikelement beginnen. Auch sie erwartet im ersten Parameter ein XML-Element, das als Elternelement für die neu erzeugten XML-Elemente dienen soll.

Die erste Etappe im Arbeitsablauf der Methode bildet die Erzeugung eines neuen Knoten-Objekts. Dieses wird wie in der zuletzt gezeigten Methode mit *IDomDocument.eateElement* erzeugt und über ein *IDomElement*-Interface angesprochen. Als Tag-Name dieses Elements wird *xmlTypeGraphicElement* gesetzt, so dass in der letztlich resultierenden XML-Datei dafür ein Block nach dem Muster

```
<GraphicElement ...>
</GraphicElement>
```

entsteht, wobei der Teil »...« noch durch die Attribute gefüllt wird. Diese Attribute werden von *SaveToXML* über mehrfache Aufrufe von *setAttribute* gesetzt:

```
function TGraphicElement.SaveToXML(ParentElem: IDomElement): IDomElement;
var
  Doc: IDomDocument;
  newElem: IDomElement;
  i : Integer;
begin
  Doc := ParentElem.ownerDocument;
  if Doc = nil then Doc := ParentElem as IDomDocument;
```

```
    newElem := Doc.createElement(xmlTypeGraphicElement);
    newElem.setAttribute(xmlAttrShapeText, FText);
    newElem.setAttribute(xmlAttrLeft, Points.Left);
    newElem.setAttribute(xmlAttrRight, Points.Right);
    newElem.setAttribute(xmlAttrTop, Points.Top);
    newElem.setAttribute(xmlAttrBottom, Points.Bottom);
    newElem.setAttribute(xmlAttrShape, integer(FShapeType));
    newElem.setAttribute(xmlAttrPenColor, Pen.Color);
    ... (Speichern weiterer Attribute und der Edge-Liste)...
    ParentElem.appendChild(newElem);
    Result := newElem;
  end;
```

Ein Beispiel für ein auf diese Weise erzeugtes vollständiges XML-Element finden Sie in Abbildung 6.18.

XML-Schreibmethoden

Nun lässt sich die Schreibmethode für das gesamte Dokument auf den wiederholten Aufruf der Schreibmethode für ein einzelnes Element zurückführen. Als einziges Attribut des XML-Elements für das gesamte TreeDesigner-Dokument wird eine Versionskennung gesetzt, die jedoch im TreeDesigner bisher nicht weiter verwendet wird:

```
procedure TGraphicDoc.SaveToXML(XMLDocElem: IDomElement;
  OnlyMarked: boolean);
var
  i: Integer;
begin
  XMLDocElem.SetAttribute(xmlAttrDocVersion, XMLFileVersion);
  for i := 0 to Count-1 do
    if (not OnlyMarked) or Items[i].Marked then begin
      Items[i].SaveToXML(XMLDocElem);
    end;
end;
```

Hintergrund: Gegenüber der binären Schreibmethode *SaveToStream* muss hier nicht die Zahl der Elemente in die Datei geschrieben werden. Dafür werden im vollständigen Code die Elemente vor dem Speichern durchnummeriert, damit »Zeiger« von einem Element auf andere Elemente möglich werden (schließlich sollen auch die Verbindungen zwischen den Elementen gespeichert werden, was in der vollständigen Methode *TGraphicElement.SaveToXML* stattfindet).

Zurücklesen der XML-Datei

Aus Platzgründen können die Methoden zum Lesen einer TreeDesigner-XML-Datei nicht ebenfalls so ausführlich abgedruckt werden wie die Schreibmethoden. Zunächst muss wieder ein Dokument-Objekt erzeugt werden, das mit der *load*-Methode zum Einlesen und Parsen der Datei aufgefordert werden kann (siehe Kapitel 6.9.2). Dann

wird der gesamte XML-Baum durchlaufen, um für jedes XML-Element `<GraphicElement>` ein *TGraphicElement* zu erzeugen. Dieses überträgt dann die Attribute des XML-Elements in seine eigenen Properties. In Analogie zu den oben gezeigten *setAttribute*-Aufrufen sind nun *getAttribute*-Aufrufe angesagt:

```
procedure TGraphicElement.LoadFromXML(Elem: IDomElement);
begin
  FText := Elem.getAttribute(xmlAttrShapeText);
  Points.Left := Elem.getAttribute(xmlAttrLeft);
  FShapeType := Elem.getAttribute(xmlAttrShape);
  Pen.Color := Elem.getAttribute(xmlAttrPenColor);
  ...
```

Abschließend soll noch auf einen möglichen Nachteil der großen Flexibilität von XML-Dokumenten hingewiesen werden: Auf die gerade skizzierte Weise können auch Dokumente gelesen werden, die mehr Informationen enthalten, als verwertet werden. Z.B. könnte die Datei von einer zukünftigen TreeDesigner-Version geschrieben worden sein und Attribute enthalten, die oben nicht mit *getAttribute* ausgelesen werden. Dass diese Datei auch von früheren Versionen des TreeDesigners verstanden werden kann, ist zunächst einer der Vorteile von XML.

Beim Speichern dieser Datei im XML-Format würden jedoch die zusätzlichen Informationen verloren gehen. Auch Kommentare, die vielleicht von einem Menschen in die XML-Datei eingefügt wurden, werden beim maschinellen Erzeugen der XML-Datei durch eine Anwendung nicht wiederhergestellt, es sei denn, die Anwendung liest die Kommentare aus den Element-Objekten des DOM aus und merkt sie sich (was theoretisch ja möglich wäre).

ür dieses Problem bieten sich unter anderem die beiden folgenden Lösungswege an:

- Zum Teil entsteht das Problem gar nicht erst, da viele XML-Dokumente nur als Nachricht zwischen Computern ausgetauscht werden und nach der Verarbeitung der Nachricht beim Empfänger-Computer ohnehin gelöscht werden. (Das SOAP-Protokoll verwendet beispielsweise Nachrichten im XML-Format. Dabei sendet ein Sender ein XML-Dokument zu einem Empfänger, der unter Umständen nicht alle Informationen der Nachricht verwerten kann, jedoch als Antwort eine komplett neues XML-Dokument generiert.)
- Wenn XML-Dokumente für Benutzerdateien verwendet werden, können die Techniken verwendet werden, die auch schon vor der Erfindung von XML für Benutzerdokumente Anwendung fanden, z.B. Berücksichtigung der Dateiformat-Versionsnummer und Sicherheitsabfrage vor dem Überschreiben einer Datei in einem neueren Format; oder Warnhinweise am Anfang der Datei, dass keine manuellen Erweiterungen (wie Kommentare) hinzugefügt werden sollen, da diese beim Speichern vom Programm gelöscht werden (in Delphi für Windows finden Sie solche Warnungen z.B. in den Units, die beim Importieren von Typenbibliotheken erzeugt werden).

7 Datenbankanwendungen

Neben der neuen CLX bekam Kylix von Borland gleich auch noch eine neue Datenbankarchitektur namens *dbExpress* auf den Weg; die unter Windows noch verfügbare Borland Database Engine (BDE) wurde nicht nach Linux portiert. Während Sie mit *dbExpress* auf professionelle Server-Datenbanken zugreifen können und Sie damit weit entwickelte und bewährte Technologien benutzen können, steckt die Datenbankunterstützung der Kylix-IDE selbst noch in den Kinderschuhen – vergleicht man sie etwa mit Delphi 5 –, denn zahlreiche von dort bekannten Funktionen stehen in Kylix (noch) nicht zur Verfügung, dazu gehören in die Delphi-IDE integrierte Tools wie Datenbank-Explorer, Datenbank-Formularexperte und Datenmodul-Designer und auch externe Hilfsmittel zur Verwaltung von Datenbanken wie eine Datenbankoberfläche für »kleine« Datenbanken und die *isql*-Oberfläche für die Interaktion mit Interbase (welche in der Linux-Version von Interbase nur über ein Terminalfenster möglich ist). Für den Funktionsumfang der mit Kylix entwickelten Datenbankanwendungen bedeutet das jedoch kein Hindernis, und wer auch Nutzer von Delphi 6 ist, kann theoretisch dort eine *dbExpress*-Anwendung mit den weiter entwickelten Delphi-Tools erstellen und sie dann unter Kylix für Linux kompilieren.

Dieses Kapitel befasst sich sowohl mit den unter Kylix zur Verfügung stehenden Entwurfs-Hilfsmitteln für Datenbankanwendungen als auch mit der internen Funktionsweise einer solchen Anwendung, die mit der Funktionsweise einer *dbExpress*-Anwendung unter Delphi 6 übereinstimmt und auch der einer BDE-Anwendung unter Windows ähnlich ist – abgesehen von fundamentalen Unterschieden bei dem, was vor der »Datenquelle« abläuft (mit »Datenquelle« sind die *TDataSource*-Komponenten gemeint, die Ihnen bald vorgestellt werden). Die Teilkapitel sind im Einzelnen:

- Kapitel 7.1 zeigt, wie Sie mit den Bordmitteln von Kylix Tabellen und Datenbanken im Interbase- und im MyBase-Format erzeugen und wie Sie Desktopdatenbanken, die Sie mit Delphi für Windows verwendet haben, nach Linux portieren können.
- Kapitel 7.2 steigt dann in die Kylix-IDE ein und erläutert den grundlegenden Aufbau eines Datenbank-Formulars so wie die automatischen Abläufe innerhalb einer Datenbankanwendung. Als Beispielprogamme werden zwei Browser-Anwendungen vorgestellt, mit denen Sie MyBase-Dateien bzw. Interbase-Datenbanken untersuchen können.

- Kapitel 7.3 befasst sich im Detail mit Feldern und den dafür von Kylix bereitgestellten Komponenten; zu den Themen gehören beispielsweise das programmgesteuerte Untersuchen der Spaltentypen und das Setzen der Feldwerte in einem Record. Als neue Beispielanwendung verwendet dieses Kapitel eine Dateidatenbank, die ganz programmgesteuert aufgebaut wird; der Benutzer muss nur das einzulesende Verzeichnis angeben.
- In Kapitel 7.4 wird die Dateidatenbank durch Verwendung von Indizes, die das Hauptthema dieses Kapitels sind, erweitert und vervollständigt.
- Ein praxisnäheres Beispiel für eine Datenbankanwendung mit mehreren Formularen und Datenmengen, in denen auch Benutzereingaben verarbeitet werden, finden Sie in Kapitel 7.5. Anhand einer Terminverwaltungs-Anwendung wird dort die Arbeit mit SQL-Abfragen und Datenbank-Updates demonstriert.

7.1 Datenbank-Variationen und Datenzugriff

Auch wenn in diesem Buch sehr wenig Platz für die Erläuterung von allgemeinen Datenbankgrundlagen ist, können an dieser Stelle grundlegende Begriffe der Datenbankprogrammierung zusammengefasst werden.

Grundbegriffe und Klassen

Als *Datenbanken* sollen im Folgenden Sammlungen von einer oder mehreren *Tabellen* gelten. Jede Tabelle hat eine genau festgelegte Struktur: Sie besteht aus einer oder mehreren Spalten, die jeweils einen Namen und einen Datentyp besitzen, und beliebig vielen Datensätzen, den Records. Die einzelnen Spalten werden auch als *Felder* bezeichnet. Die Tabellen lassen sich als Spezialfall des abstrakten Konzepts der *Datenmengen* auffassen. Eine Datenmenge hat den gleichen Aufbau wie eine Tabelle, muss aber nicht Teil einer Datenbank sein, sondern kann auch dynamisch aufgebaut werden, z. B. als Ergebnis einer *Abfrage*, die z. B. lauten könnte: »Liste alle Datensätze auf, in denen das Feld Energiebedarf kleiner als 2 ist.«

Das wichtigste Verknüpfungsmerkmal von Tabellen in relationalen Datenbanken ist die *Relation* zwischen zwei Tabellen. Eine Relation verknüpft die Einträge aus einer Tabelle mit Zusatzinformationen aus einer anderen Tabelle. Dabei dient jeweils ein Feld als Schlüssel, der in beiden Tabellen übereinstimmen muss, um die »Gesamtinformation« zu ergeben. Ein Beispiel: In der Dateidatenbank aus Kapitel 7.3.6 werden in einer Tabelle die Namen aller Dateien aufgeführt, die in einem bestimmten Verzeichnisbaum enthalten sind. Dabei soll neben dem Dateinamen auch der komplette Pfad abgespeichert werden. Nun wäre es jedoch nicht besonders platzsparend, wenn etwa für 100 Dateien eines Verzeichnisses 100-mal der komplette und immer gleiche Verzeichnispfad abgespeichert werden müsste. Daher speichert die Dateidatenbank die

Verzeichnispfade in einer eigenen Tabelle, versieht sie mit Nummern und legt in der Dateitabelle nur noch die Nummer des Pfades ab anstatt den kompletten String. Die Verknüpfung der beiden Tabellen geschieht also über das Feld »Verzeichnisnummer«, das in beiden Tabellen enthalten sein muss.

Das Standardbeispiel für Relationen aus Geschäftsanwendungen ist wohl das einer einfachen Kundendatenbank, in der die Adressen der Kunden in einer Tabelle aufbewahrt und mit den berühmten Kundennummern versehen werden. Zu vielen Kunden gibt es nun in einer anderen Tabelle eine oder mehrere Bestellungen. Wenn die Größe einer Bestellung nicht von vornherein abschätzbar ist, wird normalerweise noch eine Tabelle für die einzelnen Bestellpositionen angelegt. Diese Tabelle enthält alle Bestellpositionen aller Bestellungen aller Kunden, was auf den ersten Blick etwas unübersichtlich erscheinen mag. Doch über den Schlüssel der Bestellnummer können genau die Positionen gefunden werden, die zu einer bestimmten Bestellung eines bestimmten Kunden gehören.

Allgemein empfiehlt sich die Bildung einer Relation immer dann, wenn zu einer Menge von Informationen mehrere gleich aufgebaute Mengen von Zusatzinformationen gehören (mehrere Dateien zu einem Verzeichnis, mehrere Bestellungen pro Kunde, mehrere Bestellpositionen pro Bestellung). Dieses Grundprinzip lässt sich auch in formalere Regeln fassen und wird in der Datenbank-Theorie bei der Definition der so genannten *Normalformen* verwendet, die jedoch in diesem Kaptiel nicht weiter erörtert werden können.

In der CLX finden Sie einige der genannten Konzepte durch grundlegende Klassen repräsentiert: *TDataSet* und davon abgeleitete *DataSet*-Klassen stehen für Datenmengen, die Objekte der *TField*-Klassen für einzelne Felder. Für die Datenbanken selbst gibt es keine Klasse, da diese von externen Datenbanksystemen oder Treibern verwaltet werden. Auf der Seite von Kylix treten Datenbanken stattdessen durch *Connection*-Objekte in Erscheinung, die eine von *TCustomConnection* abgeleitete Klasse haben. Sie sorgen für eine Verbindung zum dbExpress-Datenbanktreiber, der dann gegebenenfalls mit dem externen Datenbank-Sever Kontakt aufnimmt. Die Relationen zwischen Tabellen begegnen uns in den Komponenten in den Properties *MasterSource* und *MasterFields* der Datenmengen-Komponenten wieder (siehe hierzu Kapitel 7.4.5).

Physikalische Struktur

Was die physikalische Struktur der Datenbank betrifft, so gibt es zwei grundlegende Arten, eine Datenbank auf Datenträger zu speichern. Die einfachere Art besteht darin, für jede Tabelle eine eigene Datei anzulegen. Dies können Sie in Kylix mit den Komponenten *TClientDataSet* und *TSQLClientDataSet* erreichen. Beide können sowohl binäre als auch XML-Dateien erzeugen. In Delphi werden Desktop-Datenbanken im Paradox- und dBase-Format ebenfalls in der Form »eine Tabelle – eine Datei« abgelegt. Auf diese

Tabellen können Sie jedoch – zumindest mit den von Borland gelieferten Treibern – nicht von dbExpress aus zugreifen. Wie Sie Tabellendateien mit dbExpress erzeugen önnen, zeigt Kapitel 7.1.1.

Die andere Organisationsform besteht darin, dass die gesamte Datenbank in einer einzigen Datei abgelegt wird. Dies ist bei den von Interbase angelegten Datenbanken der Fall, die üblicherweise in Dateien mit der Endung `.gdb` gespeichert werden. Wie Sie eine solche Datenbank erzeugen können, erfahren Sie in den Kapitel 7.1.2 und 7.1.3.

7.1.1 MyBase-Datenbanken

Die bereits in der Einleitung erwähnten lokalen Dateien, die Sie mit *TClientDataSet* und *TSQLClientDataSet* erzeugen können, wurden von Borland mit dem schönen kurzen Namen »MyBase« versehen, der auch gleich zum Ausdruck bringt, dass sich diese abellen nicht mit professionellen Datenbanken vergleichen wollen. Beachten Sie jedoch den Unterschied zwischen MyBase und *MySQL* (einem SQL-Datenbanksystem, auf das Sie ebenfalls mit dbExpress zugreifen können, das aber seine eigene Dateiverwaltung hat und mit MyBase-Tabellen nichts anfangen kann!)

Im Lieferumfang von Kylix befinden sich bereits eine ganze Reihe von MyBase-Tabellen, die sich gut zu Demonstrations- und Lernzwecken verwenden lassen, denn zur erwendung einer MyBase-Tabelle in einer Kylix-Anwendung müssen Sie weder ein Datenbanksystem installieren oder konfigurieren noch eine Verbindungskomponente ür den Zugriff auf eine Datenbank einrichten. Sie finden diese Tabellen im Verzeichnis `kylix/demos/db/data` jeweils im Binärformat (Dateiendung `.cds`) und im XML-Format (Dateiendung `.xml`).

Auch im Bereich der professionellen Datenbankanwendungen gibt es durchaus Verwendung für MyBase-Tabellen. Das Standardbeispiel dafür geben Anwendungen nach dem Aktenkoffermodell (briefcase model), bei denen eine Datenbank zwar auf einem Server gelagert ist, die Anwendung aber nicht immer mit diesem Server verbunden ist und daher für die Zeit der Trennung einen für den Anwender wichtigen Teilbereich der Daten auf dem lokalen PC verwaltet – eben in den MyBase-Dateien. Der »Anwender« kann z.B. ein Außendienstmitarbeiter sein, der auf seinen Reisen zusätzliche Daten sammelt, die zunächst in der MyBase-Datei aufbewahrt und später dann bei einem Wiedersehen des Servers zu diesem weitergeleitet werden.

Zwei wichtige Grenzen von MyBase-Dateien sollten Sie jedoch kennen:

- Die ClientDataSet-Komponenten halten immer die gesamte Datenmenge im Speicher und verwalten sie dort in einem eigenen Datenpaket namens *Data*. Zwar hat sich die Speicherausstattung der Computer seit der Erfindung der Datenbank um Zehnerpotenzen gesteigert, jedoch kann man auch den zu verwaltenden Daten nicht jegliches Wachstum absprechen und manche Datenbanken werden wohl immer zu groß sein, um sie komplett in den Speicher zu laden.

- Die ClientDataSet-Klassen sind zwar von *TDataSet* abgeleitet und erben damit zahlreiche nützliche Funktionen zum Bearbeiten, Filtern und Suchen von Daten, jedoch können Sie auf eine MyBase-Datei keine SQL-Abfrage anwenden. Die SQL-Funktionen der ClientDataSet-Klassen sind dafür vorgesehen, das *Ergebnis* einer SQL-Abfrage von einem externen Datenbank-Server aufzunehmen (dieses Ergebnis kann dann natürlich wieder in einer MyBase-Datei gespeichert, von dort aber nicht mehr weiter per SQL bearbeitet werden, denn die SQL-Ausführung findet immer in einem externen Datenbank-System statt).

Wenn diese Grenzen nicht stören und Sie nicht die speziellen Fähigkeiten eines professionellen Datenbanksystems wie Transaktionskontrolle, Sicherheitsfunktionen etc. benötigen, dann geben Ihnen die MyBase-Datenbanken einen guten Ansatz zur Entwicklung einer Anwendung, die sich später auch leicht auf ein größeres System skalieren lässt. Bei der Verwendung von MyBase benötigen Sie neben der ausführbaren Datei Ihrer Anwendung übrigens nur die Datei `midas.so`.

MyBase-Tabellen erzeugen

Delphi-Benutzer werden vielleicht in Kylix die Datenbankoberfläche vermissen, mit der unter Windows schnell Tabellen erzeugt werden können (und sei dies auch wieder nur zu Experimentierzwecken), doch auch unter Kylix brauchen Sie nicht auf eine Möglichkeit zu verzichten, Tabellen mit Hilfe von Dialogen erzeugen zu können. Diese Möglichkeit fällt allerdings, da sie sich stark auf die Bordmittel der IDE – insbesondere den Objektinspektor – stützt, etwas weniger komfortabel aus als bei der Datenbankoberfläche.

Um eine neue Tabelle zu erzeugen, benötigen Sie zunächst eine *TClientDataSet* oder eine *TSQLClientDataSet*-Komponente (letztere verursacht zusätzlichen Ressourcen-Verbrauch, weshalb sich ihre Verwendung alleine für das Erzeugen von MyBase-Tabellen nicht lohnt). Fügen Sie eine neue davon in ein Formular ein (mit bestehenden Komponenten aus Beispielprogrammen, die bereits mit irgendwelchen Tabellen verknüpft sind, funktioniert der folgende Ablauf nicht!) und starten mit einem Doppelklick auf das Icon der Komponente (oder über ihr Popup-Menü) den Felder-Editor (Abbildung 7.1), der zunächst noch keinerlei Einträge enthält.

Aus seinem Popup-Menü können Sie nun den Punkt NEUE FELDER aufrufen, um in das ebenfalls in der Abbildung dargestellte Dialogfenster zu gelangen. Hier können Sie nun vier Eigenschaften eines neuen Feldes (d.h. einer neuen Tabellenspalte) festlegen: Den Namen des Feldes in der Datei, den Namen der Komponente, die dieses Feld auf Seiten der Kylix-Anwendung vertreten soll, den Datentyp, der in diesem Feld gespeichert werden soll, und je nach Datentyp noch die Größe des Feldes. Wichtig ist, dass der Radio-Schalter FELDTYP auf DATEN gestellt ist, wenn Sie das Feld in der Datei speichern wollen.

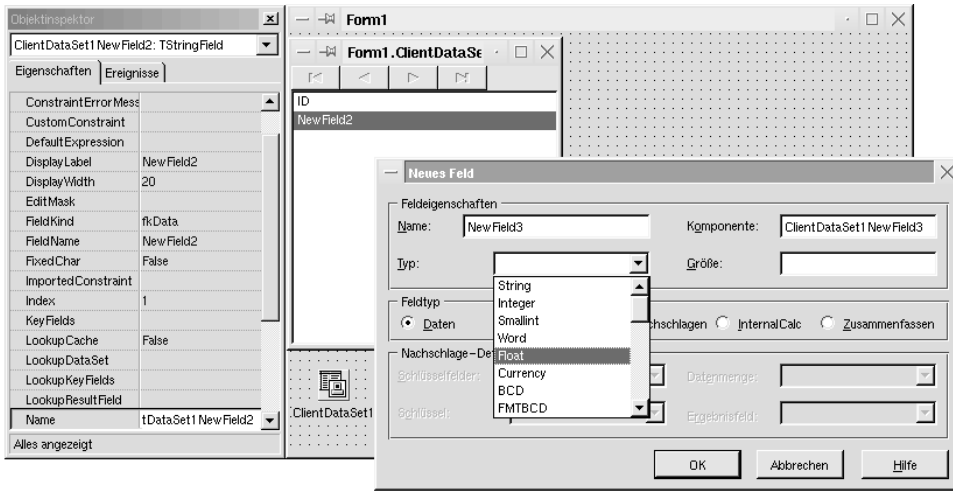


Abbildung 7.1: An der Erzeugung einer neuen MyBase-Tabelle sind der Feldereditor, ein Dialog und optional (z.B. für spätere Korrekturen) der Objektinspektor beteiligt.

Auf diese Weise können Sie nun für jede Spalte der neuen Tabelle ein neues Feld definieren. Feldname, Komponentenname, Feldtyp (der im Dialog auf DATEN eingestellt sein sollte) und Datengröße sowie den Index des Feldes in der Felderliste können Sie nachträglich auch im Objektinspektor ändern, wenn Sie das Feld im Felder-Editor selektieren. Der Datentyp des Feldes lässt sich nachträglich nicht mehr ändern, weil er nicht durch ein Property, sondern durch die Klasse der Feldkomponente repräsentiert wird. Für ein String-Feld erhalten Sie beispielsweise eine Feldkomponente der Klasse *TStringField*. Um dies nachträglich zu ändern, müssen Sie also die Feldkomponente aus dem Felder-Editor löschen und ein neues Feld erzeugen.

Wenn Sie die Definition der Spalten abgeschlossen haben, führen Sie zwei Punkte des lokalen Menüs der ClientDataSet-Komponente aus:

- Mit DATASET ERSTELLEN bewirken Sie, dass die Komponente im Speicher eine Struktur aufbaut, die den von Ihnen definierten Tabellenspalten entspricht.
- Der Erfolg dieses ersten Schritts zeigt sich darin, dass das Popup-Menü danach einige zusätzliche Einträge enthält. Wählen Sie nun je nach dem gewünschten Dateiformat den Punkt IN MYBASE-XML-TABELLE SPEICHERN... oder IN BINÄRE MYBASE-DATEI SPEICHERN.

Den Namen der so erzeugten Datei können Sie nun optional, wenn Sie die ClientDataSet-Komponente weiterverwenden wollen, in deren *FileName*-Property angeben, so dass die Komponente zukünftig ihre Daten automatisch aus dieser Datei bezieht bzw. sie darin speichert. In Kapitel 7.2.4 finden Sie ein Beispielprogramm zu *TClientDataSet* und MyBase-Dateien.

Erzeugen der Tabellen zur Laufzeit

Der beschriebene Ablauf im Felder-Editor entspricht exakt dem Ablauf, mit dem Sie eine MyBase-Tabelle auch zur Laufzeit programmgesteuert erzeugen können. Sie nehmen eine leere Felderliste (*TClientDataSet.FieldDefs*) und fügen der Reihe nach die einzelnen Spalten hinzu (*Add*-Methode), wobei Sie den Datentyp und bei bestimmten Typen die Datengröße als Parameter angeben. Danach rufen Sie statt des Menüpunkts DATASET ERSTELLEN die Methode *CreateDataSet* auf.

Für die in Kapitel 7.3.6 verwendete Dateidatenbank würde dies etwa wie folgt aussehen:

```
with Files do begin
  Active := False;
  with FieldDefs do begin
    Clear;
    Add('Verzeichnisnummer', ftInteger);
    Add('Dateiname', ftString, 12);
    Add('Groesse', ftInteger);
    Add('Datum', ftDateTime);
  end;
  CreateDataSet;
end;
```

Auf ähnliche Weise können Sie auch Indizes definieren (hier ist dann *IndexDefs* statt *FieldDefs* involviert). Beachten Sie, dass *TClientDataSet* nur dann eine Datei auf dem Datenträger erzeugt, wenn Sie explizit die *SaveToFile*-Methode aufrufen oder das *FileName*-Property gesetzt haben (die automatische Speicherung findet im letzteren Fall allerdings erst beim Schließen der Anwendung statt).

Hinweis: Das erwähnte Beispielprogramm aus Kapitel 7.3.6 verwendet keine MyBase-Tabellen, sondern eine Interbase-Datenbank. Daher kann der obige Code in diesem Programm *nicht* verwendet werden!

7.1.2 Interbase-Datenbanken

Mit Interbase steht ein leistungsfähiger Datenbank-Server unter Windows und Linux zur Verfügung, der in der Version 6 sogar als Open Source vorliegt. Die auf der CD von Kylix befindliche Version 5.6 ist noch lizenzpflichtig, kann aber 90 Tage lang evaluiert werden. Die auf der CD zu diesem Buch enthaltenen Beispieldatenbanken sind mit Interbase 5.6 erstellt worden, funktionieren aber auch mit der Version 6, ebenso wie natürlich die Beispielprogramme.

Da es im Rahmen dieses Kapitels nicht möglich ist, Sie zum Datenbankadministrator auszubilden, gehen die Beispielprogramme dieses Kapitels, die mit Interbase arbeiten, von einer lokalen Installation des Interbase-Servers aus. Das Installationsprogramm

sorgt automatisch dafür, dass Interbase bei jedem Start des Systems als Hintergrundprozess geladen wird und in Kylix-Anwendungen sofort angesprochen werden kann.

ür einen ersten Funktionstest genügen die von Borland mitgelieferten Beispieldatenbanken, als da wären die `dbdemos.gdb` im Verzeichnis `kylix/demos/db/data` (ihr Inhalt entspricht ungefähr den auch im MyBase-Format vorliegenden Tabellen desselben erzeichnisses) und die `employee.gdb` im `Verzeichnis interbase/examples`.

In Kapitel 7.1.4 wird erläutert, wie Sie eine Verbindung zu einer solchen Datenbank herstellen können, in Kapitel 7.2.5 finden Sie eine Browser-Anwendung, mit der Sie den Inhalt dieser Datenbanken komfortabel untersuchen können, und in diesem Kapitel soll es zunächst um die Erzeugung einer Interbase-Datenbank gehen.

Interaktives SQL

Während Sie in der Windows-Version von Interbase noch die Möglichkeit haben, eine Datenbank dialoggesteuert im Utility *wisql32.exe* zu erzeugen, ist unter Linux nur der Weg über das Terminal-Programm *isql* vorgesehen (dies ist kein so großer Nachteil, da Sie auch unter *wisql* die eigentlichen Tabellen *ohne* Hilfe von Dialoge erstellen). Sicher wird es bald Tools von Drittanbietern geben, mit denen sich Interbase komfortabler ansprechen lassen wird.

Isql gibt Ihnen die Möglichkeit, Kommandos einzugeben, die dann von *isql* an den Interbase-Server-Prozess geleitet werden. Die Antwort von Interbase wird wieder von *isql* angezeigt (siehe Abbildung 7.2).

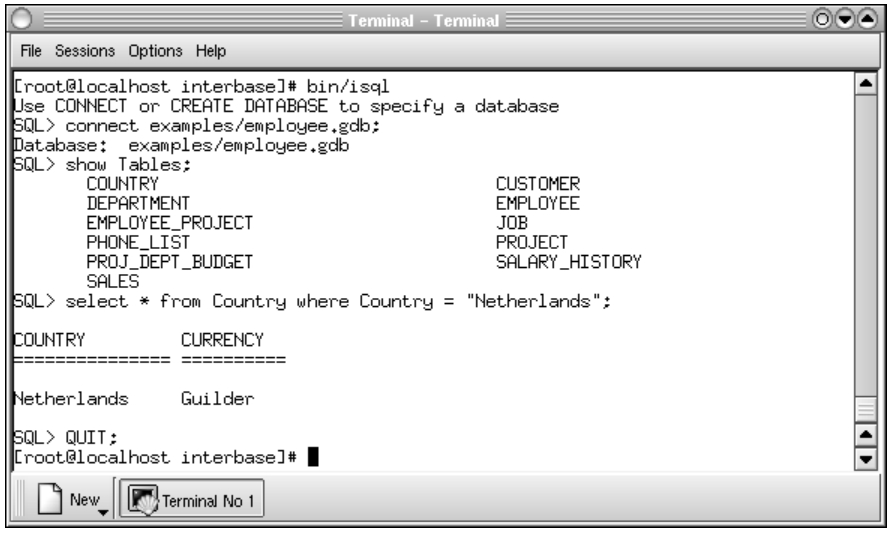


Abbildung 7.2: Im Dialog mit Interbase per isql

Auf diese Weise können Sie sowohl normale SQL-Abfragen durchführen, die als Ergebnis Datenmengen aus bestehenden Tabellen haben, als auch die *Metadaten* einer Datenbank definieren (Daten, die Informationen über die Datenbank selbst liefern, werden als Metadaten bezeichnet). Zu diesen Metadaten gehört vor allem die Struktur der Datenbank, also etwa der Aufbau der Tabellen, jedoch gibt es aufgrund der umfangreichen Funktionalität von Interbase noch viele weitere Arten von Metadaten wie etwa Trigger, Generatoren und Stored Procedures. Dies alles zu dokumentieren ist Aufgabe spezieller Bücher; eine komplette Referenz finden Sie in der *Language Reference*, die in der Interbase-Installation in der Datei *langref.pdf* installiert ist.

Während Sie im Dialog mit *isql* gut Wartungsarbeiten, Tests und Experimente durchführen können, sollten Sie die eigentliche Definition von Datenbanken und ihren Strukturen über SQL-Skripte durchführen, die sich später zu beliebiger Zeit wiederholt ausführen lassen.

Datenbanken erzeugen

Um Ihnen einen ersten Eindruck von der SQL-Data Definiton Language (DDL) zu vermitteln, ist im folgenden das Skript zur Erzeugung der Dateidatenbank aus Kapitel 7.3.6 komplett abgedruckt.

```
CREATE DATABASE "FileDB.gdb" USER 'SYSDBA' PASSWORD 'masterkey'
  PAGE_SIZE 1024;

CREATE TABLE Directories(DirID INTEGER NOT NULL, Path VARCHAR(80)
  NOT NULL);
CREATE TABLE Files(DirID INTEGER NOT NULL, Filename VARCHAR(80)
  NOT NULL, FileSize INTEGER, FileTime DATE);

CREATE UNIQUE INDEX DIRECTORIESO ON Directories(DirID);
CREATE INDEX Path ON Directories(Path);

CREATE UNIQUE INDEX FILESO ON Files(DirID, FileName);
CREATE INDEX BySize ON Files(DirID, FileSize);
CREATE INDEX ByDate ON Files(DirID, FileTime);
```

Es besteht aus mehreren Anweisungen, die jeweils mit einem Semikolon abgeschlossen werden (dieses Semikolon ist übrigens im Dialog mit *isql* erforderlich, damit *isql* das Kommando absendet, das Drücken von ↵ genügt nicht, sondern es bewirkt nur, dass Sie das Kommando der letzten Zeile in einer neuen Zeile fortsetzen können!)

In der ersten Anweisung wird eine Datenbank angelegt und eine neue Datei namens `FileDB.gdb` erzeugt. Als Benutzername und Passwort sind im obigen Listing die Standard-Zeichenketten angegeben, die auch in den Beispieldatenbanken von Interbase verwendet werden.

In den Anweisungen zwei und drei werden die beiden Tabellen der Dateidatenbank definiert; beide Anweisungen enthalten eine Liste von Feldern, für die jeweils Name, Datentyp und evtl. Datengröße angegeben werden. Für die einzelnen Felder können Sie bei Interbase auch einige zusätzliche Eigenschaften festlegen, die es bei MyBase-Dateien nicht gibt. Im obigen Listing wird lediglich die Eigenschaft »NOT NULL« angegeben, die besagt, dass dieses Feld niemals leer sein darf (alle Datensätze der Tabelle müssen in diesem Feld also einen Wert aufweisen).

Hinweis: Diese Eigenschaft ist zwar im Dialog zum Erstellen eines neuen Feldes in ClientDataSet-Komponenten (siehe Kapitel 7.1.1) nicht berücksichtigt, kann dort aber nachträglich über das Property *Required* der Feldkomponente eingestellt werden.

Die restlichen Zeilen des Listings sorgen für die Erzeugung verschiedener Indizes, die in dieser Datenbank nützlich sein *könnten*, wenn das Beispielprogramm die Such- und Sortierfunktionen von Interbase nutzen und nicht immer die gesamten Tabellen in das ClientDataSet laden würde, wo Such- und Sortierfunktionen dann unabhängig von Interbase von der Kylix-Komponente durchgeführt werden (zur Verwendung von Indizes in den ClientDataSets des Programms siehe Kapitel 7.4.4 und 7.4.5).

ür die Syntax der im Listing verwendeten DDL-Anweisungen und die verfügbaren Optionen sei auf die Interbase-Sprachreferenz verwiesen.

Skripte ausführen

Um ein Skript wie das oben gezeigte per *isql* auszuführen, gibt es zwei Möglichkeiten. Die einfachste ist, *isql* in einem Terminal-Fenster zu starten und mit dem Skript als Parameter aufzurufen:

```
isql -i CreateFileDB.sql
```

Die für wiederholte Anwendung bequemste Lösung besteht allerdings im programmgesteuerten Aufruf. Dies könnte beispielsweise mit der Linux-Funktion *system* erfolgen, für die Sie die Unit *Libc* in Ihre Anwendung einbinden müssen und *isql* etwa wie folgt aufrufen können.

```
Libc.system('/opt/interbase/bin/isql -i CreateFileDB.sql');
```

Allerdings arbeitet *system*, indem es zuerst eine Shell startet. Schon dies kann je nach Linux-Konfiguration problematisch sein, abgesehen von weiteren Sonderfällen, in denen *system* laut *man*-Dokumentation nicht verwendet werden soll.

Das Beispielprogramm *FileDB* verwendet daher statt *system* die in Kapitel 5.3.2 im Detail vorgestellte Funktion *RunProgram*, um das Skript auszuführen. Prinzipiell genügt ein Aufruf wie der folgende:

```
RunProgram('isql', ['-i', 'CreateFileDB.sql'], 'script.output', True);
```

Hierbei wurde vorausgesetzt, dass *isql* über die Pfadeinstellungen des Systems gefunden werden kann. Ist das nicht der Fall, müssen Sie wie im obigen *system*-Beispielaufruf eine komplette Pfadangabe verwenden.

RunProgram startet *isql* direkt, also ohne Shell. Die Ausgabe von *isql* wird in die Datei `script.output` umgeleitet, so dass Sie im Fehlerfall aus dieser Datei evtl. weitere Angaben entnehmen können. Wenn die Datei ganz leer ist, so ist dies ein Hinweis darauf, dass *isql* nicht gestartet werden konnte, denn auch bei erfolgreicher Skript-Ausführung gibt *isql* zumindest eine kurze Meldung aus.

Eine komplette Fehlerbehandlung für den Aufruf von *RunProgram* finden Sie im vollständigen Code von *FileDB*. Die folgende Methode bearbeitet dort den Menüpunkt DATENBANK | DATENBANK UND TABELLEN PER SKRIPT ERZEUGEN:

```
procedure TMainForm.CreateDBClick(Sender: TObject);
var
  ReturnCode: Integer;
  isqlPath, ScriptPath: String;
begin
  isqlPath := IncludeTrailingPathDelimiter(FileDBOptions.isqlPath.Text)
              + 'isql';
  ScriptPath := IncludeTrailingPathDelimiter(
                      ExtractFilePath(Application.ExeName))
                      + 'CreateFileDB.sql';
  ReturnCode := RunProgram(isqlPath, ['-i', ScriptPath], 'script.output',
                           True, FileDBOptions.gdbPath.Text);
  if WIFEXITED(ReturnCode) and (WEXITSTATUS(ReturnCode) = 255) then
    ShowMessage('isql konnte nicht ausgeführt werden. Sind die '+
                'Pfadeinstellungen korrekt?')
  else if (ReturnCode <> 0) then
    ShowMessage('isql gab einen Fehler-Code zurück. Möglicherweise '+
                'konnte das Skript nicht fehlerfrei ausgeführt werden. '+
                'Nähere Informationen in der Datei "script.output".');
end;
```

Mit den unter Linux üblichen Funktionen *WIFEXITED* und *WEXITSTATUS* wird ermittelt, ob die Ausführung eines Prozesses normal beendet und dabei ein Code 255 zurückgegeben wurde. Der Code 255 ist ein spezieller Code von *RunProgram*, der besagt, dass das angeforderte Programm nicht gestartet werden konnte. Alle anderen Codes müssen folglich von *isql* selbst zurückgeliefert werden. Und hier gilt, dass der Code 0 für eine erfolgreiche Ausführung steht, während alles andere einen Fehler signalisiert.

FileDB verfügt außerdem noch über eine Dialogbox *FileDBOptions*, über die Sie das Verzeichnis von *isql* angeben können (Editierfeld *isqlPath*) und das Verzeichnis, in dem die Datenbank erzeugt werden soll (*gdbPath*); beides wird ebenfalls in der obigen Methode berücksichtigt.

7.1.3 Portierung von Delphi-Desktop-Datenbanken

Umsteiger von Desktop-Datenbanken unter Delphi (dBase oder Paradox) haben unter Kylix die Chance, viel Neues zu lernen, denn sie sind nicht nur von den Änderungen der Datenzugriffskomponenten betroffen und müssen sich in die neue *dbExpress*-Architektur einarbeiten, sondern sie müssen sogar die zugrunde liegenden Datenbank-Dateien in ein neues Format konvertieren (sofern nicht jemand anderes als Borland einen dBase- oder Paradox-Treiber für Kylix entwickelt).

Als Ziel der Konvertierung kommen theoretisch alle unter Kylix unterstützten Formate in Frage, im Folgenden ist jedoch nur die Konvertierung in das Interbase-Format näher untersucht. Delphi 5 bringt ab der Professional-Ausgabe bereits ein Utility zur Konvertierung von Desktop-Datenbanken mit, mit dem unter anderem auch Interbase-Datenbanken erzeugt werden können. Die Konvertierung der Daten kann so unter Windows stattfinden, wo Sie auch die Anwendung selbst (die Delphi-Formulare und Object-Pascal-Dateien) auf Interbase umstellen können, bevor Sie dann für Windows und Linux denselben Code bzw. dieselben Formulare und Datenmodule verwenden können.

Hinweis: Für Konvertierungen, die sich mit Data Pump nicht durchführen lassen, können Sie auch selbst in kleines Delphi-Programm schreiben. In den BDE-Komponenten von Delphi finden Sie als kleine Hilfe dabei die Komponente *TBatchMove*, die automatisch den Inhalt einer Quell- in eine Zieltabelle schreibt.

Das Data Pump-Tool

Sie finden das Data Pump-Tool bei einer vollständigen Installation von Delphi Professional in Delphis Programmgruppe unter dem Namen *Data Pump*. Das Tool ist in der Form eines Wizards gehalten, besteht also aus mehreren Dialogseiten, zwischen denen Sie mit den Schaltern WEITER und ZURÜCK wechseln können. Wenn Sie auf diesen Seiten alle erforderlichen Angaben gemacht haben, drücken Sie auf der letzten Seite (siehe Abbildung 7.3) den Button UPSIZE, um aus den gewählten Einzel-Tabellen eine einzige Datei im Interbase-Format (Endung .GDB) zu erzeugen.

Die grundlegenden Angaben finden auf den ersten drei, in der Abbildung nicht gezeigten Seiten statt:

- Quell-Alias: Hier geben Sie das Verzeichnis an, in dem sich die zu konvertierenden Tabellen befinden. Sie haben die Wahl, das Verzeichnis über einen Alias-Namen oder über die Angabe eines Verzeichnispfades zu spezifizieren.
- Ziel-Alias: Das Zielverzeichnis kann nur über einen Alias-Namen, nicht aber als Verzeichnispfad angegeben werden. Vor diesem Schritt müssen Sie also eventuell in der BDE-Verwaltung oder mit dem Datenbank-Explorer ein entsprechendes Alias

für die zu verwendende Datenbank erzeugen. Auch die Datenbankdatei selbst muss schon vorher erzeugt werden, z.B. über einen *Create Table*-Aufruf, wie er in Kapitel 7.1.2 gezeigt wurde, oder dialoggesteuert über das *isql*-Tool für Windows.

- Tabellen zum Verschieben: Hier wählen Sie aus einer Liste aller Tabellen im Quell-Verzeichnis die Tabellen aus, die konvertiert werden sollen.

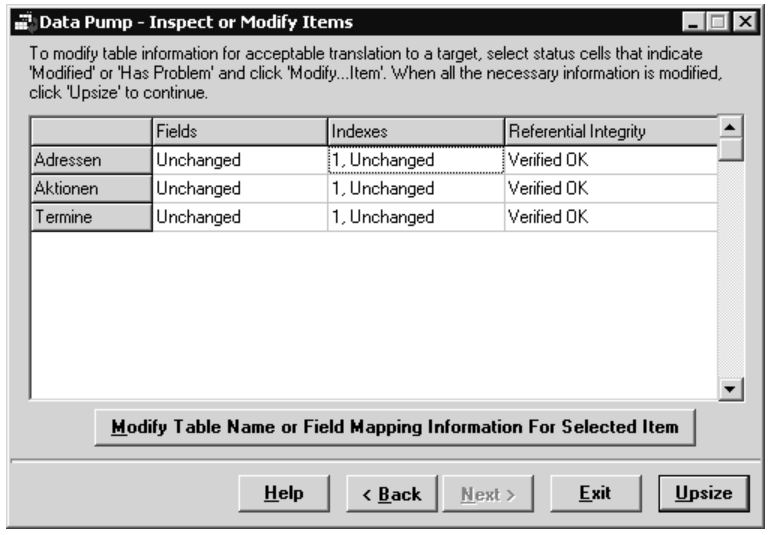


Abbildung 7.3: Konvertieren einer Desktop-Datenbank in das Interbase-Format

Auf der in Abbildung 7.3 gezeigen vierten Seite des Dialogs geht es darum, eventuelle Probleme beim Konvertieren aufzuzeigen und zu lösen. Für jede für die Konvertierung gewählte Tabelle werden die Felddefinitionen, die Indizes und die referenzielle Integrität überprüft:

- Bei der Übersetzung der Felddefinitionen können Umbenennungen erforderlich werden, wenn in der Desktop-Datenbank ein Feldname verwendet wurde, der unter Interbase nicht erlaubt ist, weil es sich um ein reserviertes Wort handelt. Das Data Pump-Utility versieht diese Felder automatisch mit einem neuen Namen; so erhält etwa das Feld SIZE den Namen »SIZ_«. Diesen Namen sowie weitere Übersetzungsparameter (verwendeter Interbase-Datentyp, Minimal- und Maximalwert etc.) können Sie jedoch im Dialog von Data Pump einstellen.
- Auch die für die Tabelle geführten Indizes versucht Data Pump zu übertragen. Im Dialog können Sie die mit Interbase erzeugten Indizes anpassen, wenn auch auf eine etwas komplizierte Weise (zwischen den einzelnen Indizes kann nur mit WEITER und ZURÜCK hin- und hergeschaltet werden).

- Die Referentielle Integrität kann unter anderem verletzt werden, wenn Sie nicht alle zusammengehörigen Tabellen gewählt haben und so die Spalte einer gewählten Tabelle mit der Spalte einer nicht gewählten Tabelle verknüpft ist. Dieser Fehler lässt sich dadurch beheben, dass Sie im Dialog zurück zur Tabellenauswahl gehen und die fehlende Tabelle hinzufügen.

Hinweis: Da Data Pump für die Konvertierung auf eine Interbase-Datenbank zugreifen muss, muss der Interbase-Dienst gestartet sein. Falls Sie diesen nicht für einen automatischen Start konfiguriert haben, können Sie den Start manuell über das Interbase-Untermenü aus dem Start-Menü nachholen (die korrekte Installation von Interbase vorausgesetzt).

est der neuen Interbase-Datei

Nachdem Sie einzelne Tabellen mit Data Pump in eine Interbase-Datenbank umgewandelt haben, müssen natürlich auch noch die Verweise innerhalb der Delphi-Anwendung von den lokalen Tabellen auf die neue Datenbank umgeleitet werden. Dies ist allerdings nur für einen ersten Test sinnvoll, denn die in Delphi verwendeten *TTable-* und *TQuery*-Komponenten stehen unter dbExpress nicht mehr zur Verfügung. Folgende Änderungen sind notwendig, um die alte Anwendung *vor* der Umstellung auf dbExpress mit der Interbase-Datenbank zusammenarbeiten zu lassen:

- Zunächst müssen die *DatabaseName*-Properties der *TTable-*, *TQuery-* oder *TDatabase*-Komponenten auf den Alias der Interbase-Datenbank weisen. Falls Sie auch für die alten Tabellen bereits ein Alias verwendet haben, können Sie dieses löschen (oder umbenennen) und für die neue Interbase-Datenbank ein Alias des alten Namens erzeugen, denn dann brauchen Sie die *DatabaseName*-Properties, in denen dieses Alias genannt wird, *nicht* zu ändern.
- Anders bei den *TableName*-Properties: Bei lokalen Datenbanken enthalten die Tabellen-Namen auch eine Dateiendung wie etwa `.db`. Diese Endung muss jeweils entfernt werden, denn sie gehört nicht zum Namen der Tabelle in der Interbase-Datenbank.

Wenn Sie die Tabellen- oder Datenbanknamen noch an anderer Stelle in Ihrem Projekt verwendet haben (z.B. kann der Tabellenname in SQL-Queries vorkommen), müssen Sie natürlich auch diese anpassen. Häufig reichen diese Änderungen bereits aus, um Ihre Anwendung mit Interbase laufen zu lassen. Sie müssen dann nur noch – wie schon für die Benutzung von Data Pump – sicherstellen, dass der lokale Interbase-Server läuft.

Auch die Änderung von Feldnamen in Data Pump kann sich natürlich auf eine Delphi-Anwendung auswirken. Wenn Sie für ein geändertes Feld eine statische *TField*-Komponente erzeugt haben, passen Sie deren *FieldName*-Property an oder entfernen die

ganze Feldkomponente aus dem Felder-Editor der zugehörigen Datenmengen-Komponente und fügen es dann mit FELDER HINZUFÜGEN wieder ein (dadurch ändert sich aber auch der Name der Komponente und Ihre speziellen Property-Einstellungen an der alten Komponente müssen wiederholt werden). Falls der Name des Feldes auch direkt im Object-Pascal-Quelltext genannt wird, sind eventuell einige weitere eher mechanische Umbenennungs-Handgriffe erforderlich.

Hinweis: Für die Autoinkrement-Felder von Paradox gibt es in Interbase keine entsprechenden Feldtypen, so dass diese Eigenschaft bei der Konvertierung verloren geht. Anwendungen, die sich daher beispielsweise auf automatisch erzeugte Datensatznummern verlassen, werden mit der so konvertierten Interbase-Datei nicht mehr korrekt funktionieren. In Server-Datenbanken wie Interbase werden Autoinkrement-Felder durch flexiblere Mechanismen nachgebildet wie etwa durch Trigger und Generatoren. Kapitel 7.5.1 gibt ein Beispiel für den Ersatz von Autoinkrement-Feldern durch Trigger.

Datenzugriffskomponenten für dbExpress und BDE

dbExpress bringt neue Datenzugriffskomponenten, arbeitet aber mit denselben Datenanzeigekomponenten wie Delphi 5. Da auch die *TDataSet*-Klasse von Delphi 5 »geerbt« wird, gestaltet sich die Portierung von Delphi-5-Datenbankanwendungen leichter, als man von einer neuen Architektur vielleicht erwarten würde. dbExpress ersetzt unter Linux vollständig die BDE (Borland Database Engine), die in Delphi für Windows natürlich auch in der Version 6 noch weiter verwendet werden kann.

Die Portabilität der Datenanzeigekomponenten bedeutet grob gesagt, dass Sie den Teil Ihrer Anwendung, der die *TDataSource*-Komponenten als Datenquelle verwendet, unverändert übernehmen können. Für dbExpress geändert werden müssen nur die Komponenten, die sich zwischen Interbase und der DataSource befinden.

Die dbExpress-Komponenten, die von Borland als Ersatz für die BDE-Komponenten *TTable*, *TQuery* und *TStoredProc* vorgeschlagen werden, sind nur sehr eingeschränkt verwendbar, da sie unidirektional sind und Sie mit diesen Komponenten beispielsweise immer nur einen Datensatz gleichzeitig anzeigen können. Um die volle Funktionalität der Anwendung zu erhalten, müssen *TTable*, *TQuery* und *TStoredProc* daher meistens durch *TSQLClientDataSet*-Komponenten ersetzt werden. In dessen Property *CommandType* wählen Sie dann aus *ctTable*, *ctQuery* und *ctStoredProc* den genauen Typ der Datenmenge. In *CommandType* machen Sie die restlichen Angaben, die etwa bei *ctTable* aus dem Namen der Tabelle, bei *ctQuery* aus der SQL-Anfrage, die in den BDE-Komponenten noch unter *TQuery.SQL* angegeben wurde, besteht.

Weder in *TSQLClientDataSet* noch in den anderen Datenmengen-Komponenten von dbExpress werden Sie das von Delphi bekannte *DatabaseName*-Property wiederfinden, da die verwendete Datenbank in *dbExpress* auf eine der drei folgenden Arten angegeben wird:

- In *FileName* können Sie eine MyBase-Datei angeben (siehe Kapitel 7.2.4).
- Eine Verbindung zu einem Datenbank-Server kann über *ConnectionName* ausgewählt (vergleichbar mit einem Alias-Namen der BDE)
- oder über eine eigene *TSQLConnection*-Komponente definiert werden. Diese Komponente entspricht in etwa der *TDatabase*-Komponente einer BDE-Anwendung von Delphi.

Der Verbindungsaufbau über *ConnectionName* oder eine *TSQLConnection*-Komponente wird in Kapitel 7.1.4 näher erläutert.

Nach dem Austausch der Datenzugriffskomponenten sind noch einige weitere Umstellungen zu erwarten. Diese können hier nicht im Detail beschrieben werden und werden auch in der Online-Hilfe von Kylix ausführlich dokumentiert. Für einfache Programme wie etwa die Dateidatenbank aus Kapitel 7.3.6 genügt es fast schon, die statischen Felder für die ClientDataSets im Felder-Editor neu zu erzeugen, die DataSource-Komponenten der Anwendung mit den ClientDataSets zu verbinden und die Komponenten nach Bedarf zu aktivieren (*Active*-Property der ClientDataSets und das *Connected*-Property der Verbindungskomponente).

Eine Überprüfung ist in jedem Fall auch bei den Update-Mechanismen notwendig, da nderungen des Benutzers in *TClientDataSet*-Komponenten nicht ebenso automatisch in die Datenbank übernommen werden wie unter der BDE (siehe hierzu Kapitel 7.2.6).

7.1.4 Datenzugriff mit Verbindungskomponenten

Damit die Kylix-Anwendung nicht nur auf Interbase, sondern theoretisch auf beliebige Datenbank-Server zugreifen kann, ist eine flexible Architektur gefragt. Der Zugriff auf den externen Server (»extern« meint hier lediglich: außerhalb der Kylix-Anwendung) wird in dbExpress durch Verbindungskomponenten gekapselt, die in *TCustomConnection* eine gemeinsame Basisklasse haben. Speziell dient etwa *TSQLConnection* (in der Komponentenpalette auf der Seite *dbExpress*) dazu, Verbindung zu allerlei SQL-Servern aufzunehmen. An diese Verbindungskomponenten können Sie dann andere Komponenten der *dbExpress*-Seite wie etwa *TSQLClientDataSet* anschließen.

Definition der Verbindungen

Im Borland-Ressourcenverzeichnis (z. B. `/root/.borland`) befindet sich eine Datei namens *dbxconnections*, in der alle Verbindungen definiert sind, die etwa in *TSQLClientDataSet* genutzt werden können. Diese Datei ist zwar als Textdatei in einem einfachen Format

(Ini-Format) leicht zu bearbeiten, jedoch können Sie über eine *TSQLConnection*-Komponente auch dialoggesteuert neue Verbindungen definieren oder bestehende Verbindungen ändern (Abbildung 7.4); die Komponente aktualisiert dann entsprechend die Datei *dbxconnections*.

Um beispielsweise eine Verbindung für eine neu erzeugte Interbase-Datenbank zu definieren, fügen Sie zunächst eine *TSQLConnection*-Komponente in ein Formular ein, rufen dann aus dem Popup-Menü dieser Komponente oder per Doppelklick den Verbindungseditor auf, klicken auf den »+«-Schalter, wählen als Treiber »Interbase« und nehmen dann in der in Abbildung 7.4 zu sehenden Tabelle auf der rechten Seite des Fensters die notwendigen Einstellungen vor. Im Minimalfall brauchen Sie hier nur im Feld DATABASE die Datenbank anzugeben; empfehlenswert ist hier eine absolute Pfadangabe, auch wenn relative Pfadangaben funktionieren, solange Sie die Verbindung nur aus einem festen Verzeichnis heraus nutzen.

Login-Daten

Wenn Sie die Möglichkeit des automatischen Logins nutzen wollen, geben Sie außerdem noch einen USER_NAME und ein PASSWORD an. Wenn eine Datenbank sowieso nur durch das Standard-Passwort *masterkey* für den Benutzernamen *SYSDBA* geschützt ist, ist dies sicher eine gute Lösung, um ständige Login-Dialoge zu vermeiden. Für kritischere Datenbanken ist die Angabe der Login-Daten in der Verbindungsdefinition jedoch nicht geeignet, da sie im Klartext innerhalb der Datei *dbxconnections* gespeichert wird.

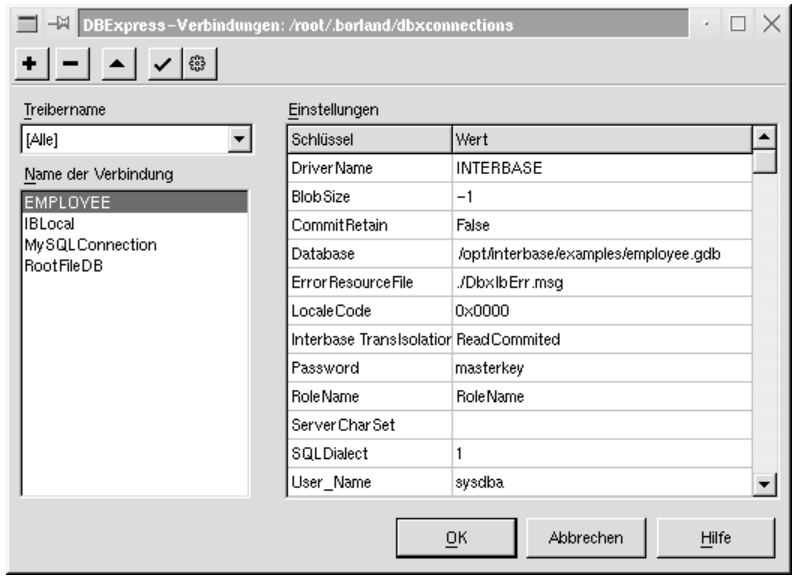


Abbildung 7.4: Definition einer Verbindung für die Dateidatenbank

Um das automatische Login zu aktivieren, müssen Sie außerdem noch das Property *TSQLConnection.LoginPrompt* auf *False* setzen, sonst erhält der Benutzer trotz einer Passwort-Angabe in den Verbindungsdaten einen Login-Dialog.

Hinweis: Alle Verbindungsdaten für eine *TSQLConnection* (*Params*-Property) werden in der Formulardatei des Datenmoduls gespeichert und daraus auch beim Programmstart wieder geladen. Die aktuellen Einstellungen aus *dbxconnections* werden *nicht* geladen, solange Sie das Property *TSQLConnection.LoadParamsOnConnect* bei seinem voreingestellten Wert *False* belassen. Dies bedeutet, dass Sie zur Installation einer Anwendung auf einem anderen PC nur die ausführbare Datei und den Datenbanktreiber installieren, sofern die Verbindungsparameter (wie etwa der Pfad zur Datenbank) sich ohne Änderung auf diesen anderen PC übertragen lassen. In Kapitel 7.2.5 wird eine Möglichkeit vorgestellt, wie Sie Verbindungsdaten programmgesteuert definieren oder ändern können, ebenfalls ohne Verwendung der Konfigurationsdatei.

Herstellen der Verbindung

Um nun eine der so definierten Verbindungen in einer Kylix-Anwendung zu nutzen, können Sie den Namen der Verbindung (in Abbildung 7.4 in der linken Listbox zu sehen) entweder direkt in einer Datenmengen-Komponente wie *TSQLClientDataSet* im *ConnectionName*-Property angeben, oder Sie kapseln die Verbindung über eine eigene Komponente der Klasse *TSQLConnection*, mit deren Verbindungseditor Sie die Verbindung ja vielleicht erst erzeugt haben. In diesem Fall setzen Sie das *ConnectionName*-Property der *TSQLConnection*-Komponente und wählen diese im Property *DBConnection* der Datenmengen-Komponente aus.

Der Vorteil der Kapselung in der eigenen Komponente liegt darin, dass Sie die Verbindung später leichter ändern können. Angenommen, Sie haben fünf *TClientDataSet*-Komponenten, die alle dieselbe Verbindung nutzen. Sie können diese Verbindung direkt im *ConnectionName*-Property dieser Komponenten angeben, müssen dann aber beim Wechsel der Verbindung alle fünf Properties ändern. Wenn Sie dagegen das Property *DBConnection* der fünf ClientDataSets mit einer Verbindungskomponente verknüpft haben, müssen Sie als Einziges das *ConnectionName*-Property dieser Komponente anpassen.

Das Konzept der Datenquelle

Schon seit der ersten Delphi-Version für Windows unterscheidet Delphi zwischen Komponenten für den Datenzugriff (nicht-visuelle Komponenten der Palettenseite *Datenzugriff*) und Komponenten für die visuelle Präsentation und Manipulation der Daten (visuelle Komponenten auf der Palettenseite *Datensteuerung*). Zu Letzteren gehören Editierfelder, Markierungsfelder und Listen, in denen Daten angezeigt, ausge-

wählt und editiert werden können. Die Zugriffskomponenten dienen dazu, den visuellen Steuerelementen die Daten bereitzustellen.

Ein zentrales Konzept dabei ist das der *Datenquelle*, realisiert durch die Komponente *TDataSource*. Alle erwähnten visuellen Daten-Steuerelemente werden an eine solche DataSource angeschlossen, beziehen daraus ihre Daten und reichen sie an diese weiter, wenn sie editiert wurden und in die Datenbank bzw. das Änderungsprotokoll zurückgeschrieben werden sollen. Eine DataSource steht dabei normalerweise für eine einzelne Tabelle bzw. Abfrage.

Der Grund, warum die Daten-Steuerelemente nicht direkt mit der Tabelle bzw. Abfrage verknüft werden, sondern nur über den Umweg über eine DataSource, wird mit der neuen Delphi-Version für Linux eindrucksvoll demonstriert: Wird eine DataSource verwendet, können Sie den gesamten Datenzugriff neu zusammenstellen, ohne etwas an den Daten-Steuerelementen zu ändern. Sie können z. B. 10 verschiedene Steuerelemente an die DataSource anschließen und brauchen doch nur eine einzige DataSource-Komponente zu ändern, um etwa zwischen den folgenden Arten des Datenzugriffs zu wechseln:

- Datenzugriff über die BDE unter Windows, z. B. zur Nutzung lokaler dBase- oder Paradox-Tabellen: Die DataSource kann dazu direkt an eine *TTable* oder *TQuery*-Komponente angeschlossen werden.
- Datenzugriff über die unter Windows zur Verfügung stehenden Interbase-Komponenten für direkten Zugriff auf Interbase ohne die BDE (dies war bis Delphi 5 ein effizienter Weg, Interbase an der behäbigen BDE vorbei zu nutzen; unter Delphi 6 empfiehlt sich dafür jedoch auch dbExpress).
- Datenzugriff über ADO-Komponenten, falls unter Windows die Datenbankschnittstelle von Microsoft genutzt werden soll, etwa zur Verwendung von Microsoft-Datenbanken.
- Datenzugriff über eine *ClientDataSet*-Komponente von Delphi 5, welche mit Hilfe der MIDAS-Bibliothek Daten von einer entfernten, mit Delphi erzeugten Server-Anwendung bezieht.
- Datenzugriff über die neue dbExpress-Bibliothek: Die DataSource wird an ein *ClientDataSet* angeschlossen, welches wiederum auf verschiedene Weise mit weiteren Zugriffskomponenten verbunden werden kann.

Auch innerhalb von dbExpress gibt es noch einmal verschiedene Variationen des Datenzugriffs:

- In den Beispielprogrammen dieses Buchs wird entweder ein *TClientDataSet* verwendet, das an eine MyBase-Datei angeschlossen ist (siehe Kapitel 7.2.4), oder ein *TSQLClientDataSet*, welches über eine *TSQLConnection*-Komponente mit einer Interbase-Datenbank verbunden ist.

- Eine Neuerung von dbExpress gegenüber der BDE stellen die unidirektionalen Datenmengen dar, zu denen *TSQLDataSet*, *TSQLTable*, *TSQLQuery* und *TSQLStoredProc* gehören (dabei sind die letztgenannten drei nur spezielle Arten von *TSQLDataSet*, die eine gewisse Abwärtskompatibilität zu Delphi-Komponenten wie beispielsweise *TTable*, *TQuery* und *TStoredProc* herstellen sollen). Diese unidirektionalen Datenmengen können Sie anstelle eines ClientDataSets verwenden, wenn Sie eine besonders schnelle Verbindung zum Datenbank-Server benötigen und die Einschränkungen der unidirektionalen Datenmengen keine Rolle spielen (die Datenmenge kann nur in einer vorgegebenen Reihenfolge durchlaufen werden, eine Anzeige in *DBGrid*-Komponenten ist nicht möglich, ebenso wenig wie das Editieren der Daten).
- Das *SQLClientDataSet*, wie es in den Beispielen dieses Buchs verwendet wird, enthält intern ebenfalls eine unidirektionale Datenmenge sowie eine Provider-Komponente, über die der Zugriff auf die unidirektionale Datenmenge erfolgt. Wenn Ihnen die voreingestellte Weise, wie *SQLClientDataSet* mit diesen internen Komponenten umgeht, nicht ausreicht, können Sie das *SQLClientDataSet* selbst in drei Komponenten aufspalten: ein normales *ClientDataSet* (ohne »SQL«), einen Provider (*DataSetProvider* auf der Komponentenpalette *Datenzugriff*) und eine unidirektionale Datenmenge.
- Diese Aufspaltung ist insbesondere dann notwendig, wenn Sie eine mehrschichtige Anwendung schreiben. Die *TDataSetProvider*-Komponente kann nämlich in eine Bibliothek ausgelagert werden, die dann auf einem externen Rechner ausgeführt wird und dort über eine unidirektionale Datenmenge und eine *SQLConnection* auf einen dritten Rechner, den Datenbank-Server, zugreifen. Auf der Seite der Anwendung (auf dem ersten Rechner bzw. der ersten Schicht) befindet sich dann lediglich noch das ClientDataSet, das über eine spezielle Verbindungskomponente auf den externen Rechner zugreifen kann (diese bisher von Delphi unter Windows bekannten Komponenten für externe Verbindungen werden wohl nach und nach auch für die Server-Developer-Ausgabe von Kylix zur Verfügung stehen).

Auch wenn Sie zwischen diesen Variationen wechseln wollen, genügt es prinzipiell, die entsprechenden Datenzugriffskomponenten umzustellen und die DataSource-Komponente mit den unter Umständen neu eingeführten Zugriffskomponenten zu verbinden. Die visuellen Komponenten, an die die DataSource-Komponente ihre Daten weiterleitet, bekommen von dieser Änderung nichts mit.

Nachdem Ihnen dieses Kapitel nun die Grundbegriffe und -Konzepte vorgestellt hat, beginnen wir im nächsten Kapitel mit der Praxis und bauen zunächst ein ganz einfaches Datenbankformular für eine MyBase-Datei auf.

7.2 Von der Tabelle zum Browser

Kylix lässt Ihnen bei der Datenbank-Programmierung, wie bei anderen Anwendungstypen auch, einen großen Spielraum darin, inwieweit Sie Ihre Anwendung bereits zur Entwurfszeit festlegen. So können Sie mit dem Felder-Editor und dem Objektinspektor Datenbankanwendungen konstruieren, die schon mit nur wenigen Zeilen Code sinnvoll einsetzbar sind. Das andere Extrem wäre, dass Sie im Formular außer ein paar Steuerelementen keinerlei Datenbankfunktionalität unterbringen und zur Laufzeit intensiv die Datenbankschnittstelle der CLX verwenden oder sogar direkt auf die dbExpress-Bibliothek zugreifen.

In diesem und in den folgenden Kapiteln werden wir beim erstgenannten Extrem, also bei Anwendungen ohne selbst geschriebenen Code, beginnen, bevor nach und nach verschiedene Möglichkeiten untersucht werden, wie Sie die CLX zur Laufzeit nutzen können, um Ihre Datenbankanwendungen individuell zu gestalten.

7.2.1 Das grundlegende Datenbankformular

Für den Einstieg in den Entwurf einer Datenbankanwendung sollen in diesem Kapitel nur MyBase-Tabellen verwendet werden. Die Datenbankkomponenten von Kylix sind so aufeinander abgestimmt, dass Sie nur einige davon zusammensetzen müssen, um eine komplette MyBase-Datenbankanwendung zu erhalten, die sich auf eine vordefinierte Weise verhält und bereits eine komplette Datenbanktabelle anzeigt, durch die Sie blättern und die Sie editieren können. Um eine solche Anwendung zu erhalten, müssen Sie noch keinen Code schreiben, sondern lediglich einige wenige Komponenten einfügen und ein paar Properties einstellen.

Ein solches Anwendungsgerüst stellt bereits eine wichtige Standard-Verhaltensweisen zur Verfügung und bildet damit eine gute Grundlage für die meisten Datenbankanwendungen. Da Sie nur für Ergänzungen und Änderungen des Standardverhaltens weitere Komponenten einfügen und neuen Code schreiben müssen, bleibt Ihnen viel Routinearbeit erspart.

Hinweis: Wenn Sie statt MyBase eine SQL-Datenbankverbindung nutzen, müssen Sie für das Speichern von Änderungen in der Datenbank zusätzliche Dinge beachten, die in Kapitel 7.2.6 erläutert sind. Auch bei Verwendung einer MyBase-Datei werden Änderungen standardmäßig nicht in den eigentlichen Datenbestand übernommen, sondern nur in ein Änderungsprotokoll; da dieses aber zusammen mit der Datei gespeichert und automatisch wieder geladen wird, hat das Änderungsprotokoll hier zunächst keinen sichtbaren Effekt. Auch hierzu gibt Kapitel 7.2.6 weitere Informationen.

Die grundlegende Komponenten-Pipeline

Grundlegend für alle Datenbankanwendungen, die mit Hilfe der DataCLX Daten anzeigen wollen, ist der Datenfluss zwischen drei Komponenten: dem *DataSet*, der *DataSource* und den visuellen Datensteuerelementen.

Der Weg der Daten von der Datei zum Bildschirm hat – einschließlich Datei und Bildschirm – vier Stationen:

- Die erste Station lässt sich für den Entwickler grob als »außerhalb der Kylix-Anwendung« beschreiben, dies kann eine lokale Datei oder die Datenbank auf einem Netzwerkserver sein, bei dreischichtigen Anwendungen gibt es zwischen dem Datenbank-Server und dem PC, auf dem die Anwendung läuft, noch eine Zwischenschicht, was aber für die Anwendung selbst keine Rolle spielt.
- Die zweite interessante Station befindet sich in der Kylix-Anwendung und wird von den Datenzugriffskomponenten gebildet. Bei lokalen Daten kann dies ein *ClientDataSet* sein, bei Daten von einem SQL-Server die Komponente *TSQLConnection*. Die verschiedenen Varianten, in denen Sie Datenzugriffskomponenten kombinieren können, wurden schon in Kapitel 7.1.4 vorgestellt. All die dort genannten Komponenten (Datenmengen-Komponenten, Verbindungskomponenten, Provider) seien hier als zweite Station der Daten zusammengefasst.
- Für die dritte Station der Daten stellt Kylix die Klasse *TDataSource* zur Verfügung. Der Name von *TDataSource* gibt diesen Komponententyp bereits als »Datenquelle« aus. Abgesehen davon, dass in dieser Daten-Pipeline ja jede Station die Datenquelle für die jeweils nächste Station darstellt, ist *TDataSource* eine besonders wichtige Datenquelle, denn sie trennt die visuellen Komponenten von den Datenzugriffskomponenten ab und macht die visuellen Komponenten dadurch vor Änderungen im Datenzugriff unabhängig (siehe auch Kapitel 7.1.4). *TDataSource* ist also die Datenquelle für die visuellen Komponenten.
- Diese visuellen Komponenten bilden die letzte Station: Sie zeigen die aus einer *TDataSource* stammenden Daten im Formular an. Zum größten Teil handelt es sich um Spezialversionen der Standardkomponenten. So zeigt beispielsweise *TDBEdit* ein bestimmtes Feld aus dem aktuellen Datensatz der zugeordneten *DataSource* an und erlaubt dem Anwender, seinen Inhalt zu editieren.

Manueller Aufbau der Daten-Pipeline

Mit den folgenden Schritten können Sie in effektiver Weise eine Anwendung wie in Abbildung 7.5 zusammenbauen, die eine Tabelle komplett in einer scrollbaren *TDBGrid*-Komponente anzeigt:

- Fügen Sie eine *ClientDataSet*-Komponente von der Seite *Datenzugriff* in das Formular ein und geben Sie im *FileName*-Property den Namen einer Tabelle an, die durch diese Komponente repräsentiert werden soll. Wählen Sie zum Test z. B. eine der mit Kylix mitgelieferten Tabellen aus dem Verzeichnis `kylix/demos/db/data`.
- Als Nächstes fügen Sie eine Komponente des Typs *TDataSource* in das Formular ein. Das einzige Property, das Sie hier verändern müssen, ist *DataSet*, es gibt eine *DataSet*-Komponente an, aus der die *DataSource* ihre Daten bezieht. Geben Sie hier die im ersten Schritt eingefügte *TClientDataSet*-Komponente an, bzw. wählen Sie sie aus der aufklappbaren Liste aus. Solange Sie nur eine *DataSet*-Komponente im Formular haben, genügt auch schon ein Doppelklick auf das Editierfeld des *DataSource*-Properties, um die Tabelle dort einzutragen.
- Als letzte Komponente folgt nun eine, die die Daten zur Laufzeit sichtbar macht: Am informativsten ist hier *TDBGrid*. Wechseln Sie in der Komponentenpalette auf die Seite *Datensteuerung*, fügen Sie eine *TDBGrid*-Komponente in das Formular ein und setzen Sie ihr *DataSource*-Property per Doppelklick auf die im letzten Schritt eingefügte *DataSource*-Komponente.

  Das Formular in Abbildung 7.5 enthält zusätzlich zwei *TDBEdit*-Komponenten, deren *DataSource*-Property gleichermaßen angepasst wurde. Bei solchen Komponenten, die nur ein Feld der Tabelle anzeigen, müssen Sie außerdem das Property *DataField* setzen, siehe hierzu Kapitel 7.2.2.
- Wenn Sie vorerst keine Eingaben im Quelltexteditor machen und die Daten der Tabelle bereits zur Entwurfszeit sehen wollen, nehmen Sie die Pipeline nun in Betrieb, indem Sie das *Active*-Property der *TClientDataSet*-Komponente auf *True* setzen. Dies können Sie auch schon im ersten Schritt vor dem Einfügen der *TDBGrid*-Komponente tun, allerdings würden Sie dann den interessanten Einschalt-Effekt, durch den ein bisher leeres *TDBGrid* mit Daten gefüllt wird, verpassen.

Hinweis: Wenn Sie den Inhalt einer MyBase-Datei zur Entwurfszeit in ein ClientDataSet laden (indem Sie das *Active*-Property auf *True* setzen), gehört die gesamte Datenmenge zu den Eigenschaften der Komponente, die in der Formulardatei gespeichert werden, wodurch diese plötzlich auf eine nahezu beliebige Größe anwachsen kann. Dies können Sie vermeiden, indem Sie vor dem Speichern das *Active*-Property des ClientDataSets wieder abschalten.

Wenn Sie diese Anwendung starten, haben Sie die Möglichkeit, durch die Datentabelle zu blättern, die Spaltenbreiten zu ändern und sogar, die Daten zu editieren. Ein wichtiges Ergänzungselement für solch ein grundlegendes Formular wäre eine Komponente der Klasse *TDBNavigator*, sie wird in Kapitel 7.2.3 besprochen.

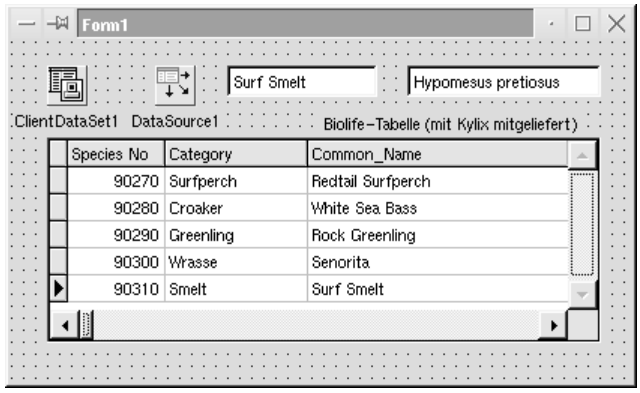


Abbildung 7.5: Ein Datenbankformular noch ohne selbst geschriebenen Quelltext (zur Anzeige des abelleninhalts wurde das Active-Property der Datenmenge eingeschaltet, jedoch nur temporär – siehe Hinweis im Text)

Das Konzept der Datenmodule

Ein Datenmodul ist ein Container für nicht-visuelle Komponenten, die in mehreren Formularen verwendet werden können. Zur Laufzeit ist ein Datenmodul in jedem Fall unsichtbar. Zur Entwurfszeit wird es als formularähnliches Fenster mit weißem Hintergrund dargestellt; die von Delphi 5 bekannte Ansicht im Datenmodul-Designer ist unter Kylix nicht verfügbar.

Um ein Datenmodul manuell zu erzeugen, wählen Sie im DATEI | NEU-Dialog das ATENMODUL. Danach können Sie alle nicht-visuellen Komponenten (Tabellen, Timer, Bilderlisten etc.) im Fenster des Moduls platzieren wie in einem Formular. Im Objektinspektor können Sie die Datenmodul-Properties *Tag* und *Name* sowie die Ereignisse *OnCreate* und *OnDestroy* bearbeiten.

Der große Vorteil der Datenmodule kommt ins Spiel, wenn Sie die in dem Modul aufbewahrten Komponenten in den verschiedenen Formularen Ihrer Anwendung verwenden wollen, z.B. um aus mehreren Formularen auf dieselbe Datenbank zuzugreifen. Statt jedes Formular, das diese Datenbank verwendet, mit einem eigenen Satz von Datenzugriffs-Komponenten zu versehen, brauchen Sie alle Datenzugriffskomponenten nur einmal in einem Datenmodul unterzubringen. Sie können diese Komponenten dann in allen Formularen Ihres Projekts benutzen.

Datenmodule verwenden

Um den Inhalt eines Datenmoduls in einem anderen Formular zu benutzen, führen Sie die folgenden Schritte aus:

- Holen Sie das Formular in den Vordergrund und wählen Sie den Menüpunkt DATEI | UNIT VERWENDEN..., wählen Sie den Namen der Datenmodul-Unit aus dem folgenden Dialogfenster aus und drücken Sie *OK*. Auf diese Weise binden Sie die Unit des Datenmoduls in der Unit des Formulars ein, so dass das Formular die Komponenten des Datenmoduls »sehen« kann.

  Dieser Schritt ist nur einmal notwendig, und zwar bevor Sie die erste Komponente des Datenmoduls im Formular benutzen.

- Nun können Sie die Komponenten des Moduls wie gewohnt im Formular benutzen. Um beispielsweise eine *DataSource*-Komponente eines Datenmoduls mit einem *DBGrid* eines Formulars zu verbinden, wählen Sie die *DataSource* einfach aus der aufklappbaren Liste des Properties *TDBGrid.DataSource* aus. In dieser Liste ist dem Namen der Komponente noch der Name des Datenmoduls vorangestellt, so wie es die übliche Pascal-Syntax verlangt (z.B. *DataModule.DataSource*).

Ansonsten ändert sich durch Datenmodule nichts in der Art, wie die Komponenten verwendet werden. Insbesondere wird bei Datenbankanwendungen der Aufbau der Daten-Pipeline nicht beeinflusst.

Zur Laufzeit ist ein Datenmodul für den Benutzer unsichtbar, nicht aber für Ihren Programmcode: Dieser kann – abgesehen von der direkten Nennung eines Moduls bei seinem Namen – die Module über Properties des *Screen*-Objekts erkennen (*DataModules* und *DataModuleCount*, zu *TScreen* siehe Kapitel 3.2.4).

Formularverknüpfung

Die Verwendung von Komponenten eines anderen Formulars durch *Formularverknüpfung* ist eine grundlegende Funktion, die unabhängig von den oben beschriebenen Datenmodulen ist. Sie steht Ihnen nicht nur zwischen Datenmodulen und Formularen, sondern auch zwischen mehreren Formularen zur Verfügung und wird allgemein als Formularverknüpfung bezeichnet. Erst das Vorhandensein der Formularverknüpfung macht die Datenmodule möglich. Diese sind zwar nicht unbedingt notwendig, da Sie gemeinsam benutzte Komponenten auch in einem beliebig ausgewählten Formular unterbringen könnten, sie fördern aber die Übersichtlichkeit eines Projekts erheblich.

7.2.2 Datensensitive Steuerelemente

TDBGrid ist nur eine von vielen datensensitiven Komponenten auf der Seite DATENSTEUERUNG. Die meisten anderen Komponenten unterscheiden sich zuerst einmal darin, dass sie weder mehrere Datensätze noch mehrere Felder eines Datensatzes anzeigen, sondern nur genau ein Feld des gerade aktuellen Datensatzes (das ist der Datensatz, der in der *TDBGrid*-Komponente selektiert ist, falls eine solche vorhanden ist). Wie schon in Kapitel 1 erwähnt, bedeutet die Bezeichnung *Datensteuerung* eher *Datensteuerelemente*, sollte also nicht mit dem Begriff *Datenzugriff* verwechselt werden.

Diese auf ein einzelnes Feld spezialisierten Datensteuerelemente geben Ihnen zwei wichtige Möglichkeiten:

- Sie können damit Masken aufbauen, die eine komfortablere Dateneingabe erlauben als eine *TDBGrid*-Tabelle.
- Darüber hinaus gibt es einige Feldtypen, die in einem *TDBGrid* nicht dargestellt werden können, und zwar sind das die *Binary Large Objects* (BLOBs), zu denen im wesentlichen Bilder und Texte variabler Länge gehören. Für diese Feldtypen sind die Komponentenklassen *TDBMemo* und *TDBImage* zuständig.

erbindung zum Feld

Sie bauen eine solche Komponente fast genauso in das Formular ein wie das *TDBGrid*, erforderlich ist zuerst einmal das Setzen des Properties *DataSource*, um die Herkunft der Daten aus der »Pipeline« anzugeben. In einem zusätzlichen Schritt müssen Sie nun noch das Feld auswählen, das angezeigt werden soll. Hierzu finden Sie im Objektinspektor beim Property *DataField* wieder eine Liste aller zur Auswahl stehenden Felder, die über die von Ihnen gewählte *DataSource* erreichbar sind.

In einem Formular mit einer *TDBGrid*-Komponente, die über eine *DataSource* mit einer abelle verbunden ist, können Sie die grundsätzliche Funktionsweise der datensensitiven Steuerelemente einfach testen. Ergänzen Sie das Formular wie in Abbildung 7.5 beispielsweise um eine *TDBEdit*-Komponente, setzen Sie deren Property *DataSource* auf denselben Wert wie die *DataSource* der *TDBGrid*-Tabelle und wählen Sie ein Feld ür *DataField* aus. Wenn Sie die Anwendung starten und im *DBGrid* einen anderen Datensatz auswählen, wechselt auch der Inhalt der *DBEdit*-Komponente, wie im folgenden Abschnitt beschrieben.

Automatische Navigation

Der Inhalt der Datensteuerelemente ist immer synchron zum aktuellen Datensatz in einer *TDBGrid*-Tabelle. Wenn Sie den Feldinhalt im Datensteuerelement editiert haben und zur Tabelle wechseln, wird die Änderung automatisch gespeichert und in das *DBGrid* übernommen.

Die Tatsache, dass der Inhalt des Editierfelds und die aktuelle Position im DBGrid immer übereinstimmen, beruht nicht auf einer geheimen Verbindung zwischen Editierfeld und DBGrid, sondern darauf, dass beide von der gleichen Datenbank-Tabellen (allgemein: von der gleichen Datenmenge) abhängen. Wenn Sie sich in einem DBGrid bewegen, ändert dieses die aktuelle Position in der Datenmenge, woraufhin auch alle anderen über *TDataSource*-Komponenten an diese Datenmenge angeschlossenen Datensteuerelemente angepasst werden.

Ein Datensteuerelement, das nur ein einzelnes Feld anzeigt, kann also nicht aus eigener Kraft zu einem anderen Datensatz wechseln, sondern benötigt die Unterstützung durch eine andere Komponente, mit deren Hilfe der Anwender durch die Tabelle blättern kann. Neben einer *TDBGrid*-Komponente kommt hier als einfachere Möglichkeit die Komponente *TDBNavigator* in Betracht, die in Kapitel 7.2.3 besprochen wird.

Verbindung zu BLOB-Feldern

Felder, die sich nicht innerhalb einer *TDBGrid*-Tabelle darstellen lassen, können Sie nur mit Hilfe eines eigenständigen Datensteuerelements darstellen, z. B. mit der Grafikkomponente *TDBImage*. Auf der Basis eines grundlegenden Datenbankformulars wie in Abbildung 7.5 genügen wenige Schritte, um diese Komponente auszuprobieren:

- Verbinden Sie eine ClientDataSet-Komponente mit einer der mit Kylix mitgelieferten Beispieldatenbanken, die auch Bilddaten enthalten (*Animals*, *Biolife* oder *Clients*).
- Setzen Sie eine *DBImage*-Komponente in das Formular.
- Setzen Sie deren *DataSource*-Property auf die *DataSource* Ihres Formulars.
- Setzen Sie schließlich das Property *DataField* auf den Namen des Grafikfelds (in den drei oben genannten Tabellen heißen diese Felder *BMP*, *Graphic* und *IMAGE*).

Diese drei Schritte können Sie genauso mit einer Memo-Komponente *TDBMemo* für Datenbanken mit einem Memo-Feld durchführen. (Der Browser aus Kapitel 7.3.4 verwendet eine *TDBImage*- und eine *TDBMemo*-Komponente, um diese zur Laufzeit dynamisch an eventuell passende Tabellenfelder zu binden.)

Steuerelement-Komponenten für Datenfelder

Die folgende Tabelle listet alle in Kylix zur Verfügung stehenden Datensteuerelement-Komponenten auf. Die weniger komplexen sind jeweils mit einer Komponente der Standardseite verwandt (»verschwistert«), die bis auf das »DB« den gleichen Namen hat.

Komponente	Aufgabe
TDBNavigator	stellt die in Kapitel 7.2.3 beschriebenen Bedienfunktionen zur Navigation durch die Tabelle und die Einstellung bestimmter Modi über eine Schalterleiste zur Verfügung.
TDBText	zeigt ein Feld des aktuellen Datensatzes als Text an (ist wie *TLabel* abgeleitet von *TCustomLabel*).
TDBEdit	erlaubt zusätzlich zu *TDBText* das Editieren des angezeigten Textes.

omponente	Aufgabe
TDBCheckBox	zeigt den Wert von booleschen Feldern des aktuellen Datensatzes an (editierbar).
TDBListBox, TDBRadioGroup	bieten eine Auswahl an möglichen Werten für ein einzelnes Feld des aktuellen Datensatzes. Der Feldinhalt kann nur durch Auswahl eines der vorgegebenen Werte geändert werden.
TDBComboBox	lässt durch ein zusätzliches Editierfeld die Beschränkung der *TDBListBox* fallen, dass keine nicht vorgegebenen Werte eingegeben werden können.
TDBGrid	zeigt mehrere Datensätzen komplett an, wobei es die einzelnen Felder als Text darstellt, und verfügt auch über eine Editiermöglichkeit.
TDBMemo	zeigt den Inhalt eines BLOB-Feldes als Text an (editierbar).
TDBImage	zeigt ein Feld des aktuellen Datensatzes mit Bilddaten an und erlaubt über Zwischenablageoperationen (Ausschneiden und Einfügen) sogar in geringem Umfang das Editieren des Bildes.
TDBLookupListBox	funktioniert wie *TDBListBox*, die Listeneinträge sind jedoch nicht fest vorgegeben, sondern werden von der Komponente zur Laufzeit des Programms aus einer anderen Tabelle ausgelesen.
TDBLookupComboBox	ergänzt die von *TDBLookupListBox* bereitgestellte Funktionalität durch das Editierfeld einer *TComboBox*.

7.2.3 Elementare Funktionen

Die Komponente *TDBNavigator* eignet sich sehr gut zur weiteren Einführung in die Datenbankprogrammierung, denn sie stellt dem Anwender elementare Funktionen, mit denen auch der Programmierer arbeitet, über Schalter zur Verfügung (in der nächsten Abbildung 7.6 ist ein solcher Navigator zu sehen). Diese Schalter betreffen die Navigation durch die Tabelle und das Editieren derselben. Selbst wenn Sie meinen, dass Sie diese Komponente niemals verwenden werden, können Sie von ihr viel über die Funktionsweise der Komponente *TTable* erfahren, denn jeder Schalter entspricht einer ihrer wichtigen Methoden.

TDBNavigator

Die Aufgabe von *TDBNavigator* ist es lediglich, die üblichen Bedienfunktionen für eine Datenbank in einer Komponente zusammenzufassen und die einzelnen Schalter zu verwalten (wenn Sie das *ShowHint*-Property einschalten, zeigt *TDBNavigator* sogar Hinweise zu diesen an). Mit der internen Funktion der Datenbankanwendung hat diese Komponente nichts zu tun. Für jeden Schalter ruft sie nämlich lediglich eine gleichbedeutende *TDataSet*-Methode auf, nur beim Löschen führt sie eine zusätzliche Sicherheitsabfrage durch, wenn das Property *ConfirmDelete* eingeschaltet ist (wie es die oreinstellung ist).

Um eine Basis zum praktischen Experimentieren zu haben, erweitern Sie eine Beispielanwendung (*FileDB* oder *DBBrowse*) oder ein grundlegendes Datenbankformular um eine *TDBNavigator*-Komponente und setzen ihr *DataSource*-Property auf die DataSource-Komponente für die gewünschte Datenmenge. Ohne eine Zeile Code sollte die Anwendung sofort lauffähig sein.

Grundkonzepte der Datenbankprogrammierung

Für die Navigationsschalter spielen der Positionszeiger und der Modus, in dem sich die Tabelle befindet, eine große Rolle. Der Positionszeiger (auch *Cursor* genannt) weist einen Datensatz aus der Tabelle als *aktuellen Datensatz* aus, der Modus unterscheidet (unter anderem) zwischen dem Editieren und dem Blättern in der Tabelle. Auf den ersten Blick sind diese beiden Eigenschaften zwar nur für die Bedienung der Datenbankanwendung wichtig, denn der Benutzer kann immer nur einen Datensatz gleichzeitig editieren und muss zwischen Editieren und Blättern in der Tabelle wechseln. Tatsächlich müssen Sie sich aber auch bei der Programmierung an diesen aktuellen Datensatz und an den gerade eingestellten Modus halten. Beides sind grundlegende Eigenschaften der Komponente *TDataSet* und damit aller speziellen Datenmengen-Komponenten wie *TClientDataSet*.

Aktueller Datensatz und Modus beeinflussen die Programmierschnittstelle der Datenbankkomponenten entscheidend: Die einzelnen Datensätze einer Tabelle sind nicht direkt und beliebig über ein indiziertes Array-Property ansprechbar (so können Sie beispielsweise nicht irgendeinen Datensatz mit einer Anweisung wie *Table.Datensatz[100].Name* := '*niemand*' setzen). Stattdessen können Sie nur den Datensatz an der aktuellen Position erreichen. Um auf beliebige Datensätze zugreifen zu können, müssen Sie für jeden Datensatz zuerst die aktuelle Position verändern. Dazu geben Sie jedoch keine Datensatznummer ein, sondern bewegen sich relativ zum aktuellen Datensatz, wie es im Folgenden beschrieben wird, oder suchen über einen Schlüssel gezielt einen bestimmten Datensatz auf (siehe Kapitel 7.4.3).

Wenn Sie bereits Erfahrung mit Datenbanken oder deren Programmierung haben, wird Ihnen dies wahrscheinlich völlig normal vorkommen, da die Position der Datensätze sich innerhalb einer Datenbank ständig ändern kann und feste Indizes daher nicht sehr hilfreich wären. Diese Art der Programmierung bildet jedoch einen Gegensatz zur einfachen Property-Programmierung in Kylix, wo sich fast alle gleichförmigen Datenmengen in ein Array-Property fassen lassen, wie es die CLX in zahlreichen Standard-Komponenten demonstriert.

Hinweis: Die ClientDataSet-Klassen verfügen über das Property *RecNo*, über das Sie den aktuellen Datensatz auch über seinen Index setzen können, was jedoch normalerweise nicht zu empfehlen ist – schon deshalb, weil sich dieser Index durch Einfügen neuer Datensätze verändern kann.

Die aktuelle Position

Es ist ein wichtiger Vorteil, dass die aktuelle Position nicht etwa eine Eigenschaft von *TDBGrid*, sondern von der Datenmenge selbst ist. Wenn sich also statt oder zusätzlich zu *TDBGrid* andere Komponenten zur Anzeige einzelner Felder im Formular befinden, so müssen Sie nicht jede Komponente einzeln auf die aktuelle Position setzen, sondern brauchen die Position nur einmal, und zwar in der *DataSet*-Komponente, zu ändern. Die CLX passt dann alle über die Daten-Pipeline angeschlossenen Datensteuerelemente an (sofern Sie diese Aktualisierung nicht gezielt unterbinden, siehe Kapitel 7.3.5, Abschnitt *Arbeiten in der Tabelle*).

Die einfachsten Arten, die aktuelle Position anzupassen, sind sowohl über eine Methode von *TDataSet* als auch über einen Schalter von *TDBNavigator* zugänglich und in der folgenden Tabelle zusammengefasst:

***TDataSet*-Methode**	**Positionierung**	***TDBNavigator*-Schalter**
First	zum ersten Datensatz des *DataSets*	nbFirst
Prior	einen Datensatz von der aktuellen Position rückwärts	nbPrior
Next	einen Datensatz weiter	nbNext
Last	zum letzten Datensatz des *DataSets*	nbLast

Die Reihenfolge in der obigen Tabelle entspricht den ersten vier Schaltern des *DBNavigators* von links nach rechts. Sie können im Property *TDBNavigator.VisibleButtons* diese und jeden anderen Schalter einzeln an- und ausschalten, indem Sie die in der Tabelle genannten Unterelemente von *VisibleButtons* (*nb...*) ändern.

Hinweis: Eng mit den obigen Methoden verknüpft sind auch die Properties *BOF* und *EOF*. Sie zeigen an, ob der Beginn (*BOF*) oder das Ende (*EOF*) der Datenmenge erreicht ist. Wenn Sie beispielsweise mit *Next* alle Elemente der Menge durchlaufen, erfahren Sie durch *EOF=True*, dass Sie das Ende erreicht haben.

Funktionsweise des Editierens

Wenn Sie die Properties der Datenbankkomponenten bei ihren Voreinstellungen belassen, können Sie die Daten zur Laufzeit ohne Probleme editieren. Wenn Sie beispielsweise in einem *DBGrid* mit der Eingabe von Zeichen beginnen, wechselt das *DBGrid* automatisch in den Editiermodus. Auch alle anderen Datensteuerelemente sind editierbar. Änderungen am aktuellen Datensatz werden automatisch in den Änderungsspeicher geschrieben, wenn Sie zu einem anderen Datensatz wechseln (zum Speichern der Änderungen in der Datenbank siehe Kapitel 7.2.6).

Sie haben eine Reihe von Möglichkeiten, auf das Editieren Einfluss zu nehmen – jede im bisherigen Beispielformular verwendete Komponente aus der Daten-Pipeline bietet hierzu ein bestimmtes Property:

- So können Sie in den editierbaren Datensteuerelementen das Editieren unterbinden, indem Sie das Property *ReadOnly* auf *True* setzen oder das Flag *dgEditing* im *Options*-Property der *TDBGrid*-Komponente löschen.
- Um die gesamte Tabelle effektiv gegen Veränderungen zu schützen, setzen Sie das *ReadOnly*-Property der *DataSet*-Komponente auf *True*.
- Die Komponente *TDataSource* bietet schließlich das Property *AutoEdit*, das per Voreinstellung eingeschaltet ist, so dass Sie wie oben beschrieben alle Felder direkt editieren können. Wenn Sie *AutoEdit* auf *False* setzen, erreichen Sie, dass der Benutzer zuerst explizit in den Editiermodus wechseln muss, indem er beispielsweise den Editieren-Schalter des *DBNavigators* drückt. Wenn er nach dem Editieren zu einem anderen Datensatz wechselt, werden die Änderungen gespeichert, der Editiermodus aber automatisch wieder abgeschaltet.
- Die niedrigste Ebene, auf der Sie das Editieren einer bestimmten Tabellenspalte verhindern können, ist das *ReadOnly*-Property der Feldkomponenten. Solche sind in unseren aktuellen Experimental-Formular jedoch noch nicht verfügbar. Kapitel 7.3 wird Ihnen die Feldkomponenten im Detail vorstellen.

Hinweis: Hinter den Kulissen bestimmt nicht die Klasse *TDataSource*, dass das Editieren im nicht-automatischen Modus wie beschrieben abläuft, denn theoretisch können die Datensteuerelemente die Tabelle trotz *AutoEdit=False* in den Editiermodus setzen und verändern, genauso wie Sie das mit eigenem Code jederzeit tun können (sofern eben nicht die *DataSet*-Komponente per *ReadOnly* geschützt ist). Die Funktion des Properties *TDataSource.AutoEdit* ist also davon abhängig, dass sich alle Steuerelement-Komponenten daran halten und nur dann das Editieren zulassen, wenn der Editiermodus auf andere Weise aktiviert wird, beispielsweise über den *DBNavigator*-Schalter.

TDataSet.Edit

Unabhängig von allen Einstellungen in *TDataSource*-Komponenten und Datensteuerelementen können Sie die Tabelle jederzeit durch den Aufruf der Methode *Edit* in den Editiermodus versetzen. Dieser Schritt wirkt sich sofort auch auf die anderen Komponenten aus, *DBNavigator* zeigt den Editiermodus automatisch in seiner Schalterdarstellung an und die Datensteuerelemente, bei denen das *ReadOnly*-Property nicht *True* ist, lassen sich editieren.

Sie können also auch diesen Schalter des *DBNavigators* leicht durch einen eigenen Schalter ersetzen, in dessen *OnClick*-Methode Sie einfach die entsprechende *TDataSet*-Methode (hier *Edit*) aufrufen. In allen anderen Fällen werden Sie die Methode *Edit* wahrscheinlich nur aufrufen, um den aktuellen Datensatz durch weitere Programmanweisungen zu verändern. Wie das geschieht, lesen Sie in Kapitel 7.3.5.

Speichern, Verwerfen und Aktualisieren

Mit dem Editiermodus alleine ist es jedoch noch nicht getan, es kommt nun darauf an, wann die Änderungen in die Tabelle geschrieben werden. Es ist klar, dass die Datenbankdatei nicht jedes Mal, wenn der Benutzer ein einzelnes Zeichen verändert, aktualisiert werden kann.

Wie schon erwähnt sorgt die voreingestellte Automatik dafür, dass die Daten jedes Mal, wenn der Benutzer den Datensatz wechselt, in ein Änderungsprotokoll geschrieben werden. Im *DBNavigator* hat der Anwender mit den Schaltern *Cancel* und *Post* zwei Möglichkeiten, diesen automatischen Ablauf zu ändern:

- Als Erstes sollte es natürlich die Möglichkeit geben, die Übernahme einer verunglückten Editieroperation in das Änderungsprotokoll zu verhindern und damit die Änderungen zu verwerfen. *TDataSet* stellt zu diesem Zweck die Methode *Cancel* zur Verfügung und im *DBNavigator* gibt es hierfür den Schalter mit dem großen »X«. In beiden Fällen werden die Steuerelemente automatisch mit den noch unveränderten Werten des Datensatzes neu geladen.
- Das Gegenteil der *Cancel*-Methode ist die Methode *Post*. Sie fordert die Datenmenge explizit auf, die Änderungen in das Änderungsprotokoll zu schreiben, so dass sie nicht mehr über *Cancel* rückgängig gemacht werden können (wohl aber noch über den Aufruf *CancelUpdates*, mit dem das Änderungsprotokoll verworfen wird, siehe Kapitel 7.2.6; oder über einen Aufruf von *UndoLastChange*, mit dem die letzte Änderung aus dem Protokoll rückgängig gemacht wird).

Der letzte Schalter von *TDBNavigator* entspricht der Methode *Refresh*. Sie liest alle in visuellen Komponenten angezeigten Datensätze aus der Tabelle neu ein. Dies ist z. B. dann sinnvoll, wenn sie zwischenzeitlich außerhalb der Kylix-Anwendung verändert worden sein könnten, beispielsweise von einer anderen Person, die auf dieselbe Datenbank zugreift.

Die in diesem Abschnitt besprochenen Funktionen sowie die *Edit*-Methode sind in der folgenden Tabelle zusammengefasst:

***TDataSet*-Methode**	**Funktion**	***TDBNavigator*-Schalter**
Edit	versetzt die Tabelle in den Editiermodus.	nbEdit
ost	schreibt die Änderungen in die Tabelle.	nbPost

TDataSet-Methode	Funktion	*TDBNavigator*-Schalter
Cancel	macht die Änderungen rückgängig, indem sie die Original-Daten erneut aus der Tabelle liest.	nbCancel
Refresh	liest alle Datensätze neu ein und ändert die Anzeige in den Datensteuerelementen, ohne aber einen gerade laufenden Editiervorgang zu beeinflussen.	nbRefresh

Löschen und Einfügen von Datensätzen

Die beiden noch nicht besprochenen Schalter des *DBNavigators* machen dem Benutzer die beiden Funktionen zugänglich, die noch fehlen, um zur Laufzeit theoretisch beliebige Änderungen der Tabelle durchführen zu können:

TDataSet-Methode	Funktion	*TDBNavigator*-Schalter
Insert	einen neuen Datensatz einfügen	nbInsert
Delete	den aktuellen Datensatz löschen	nbDelete

Wie schon erwähnt, macht ein *DBNavigator* vor dem Löschen eines Datensatzes eine Sicherheitsabfrage, während die Methode *TDataSet.Delete* sofort zur Tat schreitet.

Das Einfügen findet immer einen Datensatz vor der aktuellen Position statt. Nachdem der Benutzer (oder das Programm) die Daten eingegeben und bestätigt hat, wird die Position des Datensatzes natürlich an die Sortierreihenfolge angepasst.

7.2.4 Ein Browser für MyBase-Tabellen

Zum Abschluss dieses Einführungskapitels schreiben wir den ersten Teil eines Datenbank-Browsers, einer Beispielanwendung, die demonstrieren wird, wie Sie auch zur Laufzeit flexibel auf verschiedene Tabellen und Datenbanken reagieren können. Der Browser befindet sich auf der CD unter dem Namen cdsBrowser; Abbildung 7.6 zeigt die fertige Version, die die beiden folgenden Aufgaben dynamisch erledigt:

- Aufbau einer Verbindung zu einer MyBase-Datenbank nach Wahl des Benutzers,
- Angabe einer Liste aller Spalten der Tabelle und Auswahl von Grafik- und Memofeldern zur Darstellung in zwei einzelnen Datensteuerelementen.

Die letztgenannte Aufgabe kann erst in Kapitel 7.3.4 gelöst werden, hier bleibt die sehr einfache Aufgabe, Tabellen dynamisch zur Laufzeit anzubinden.

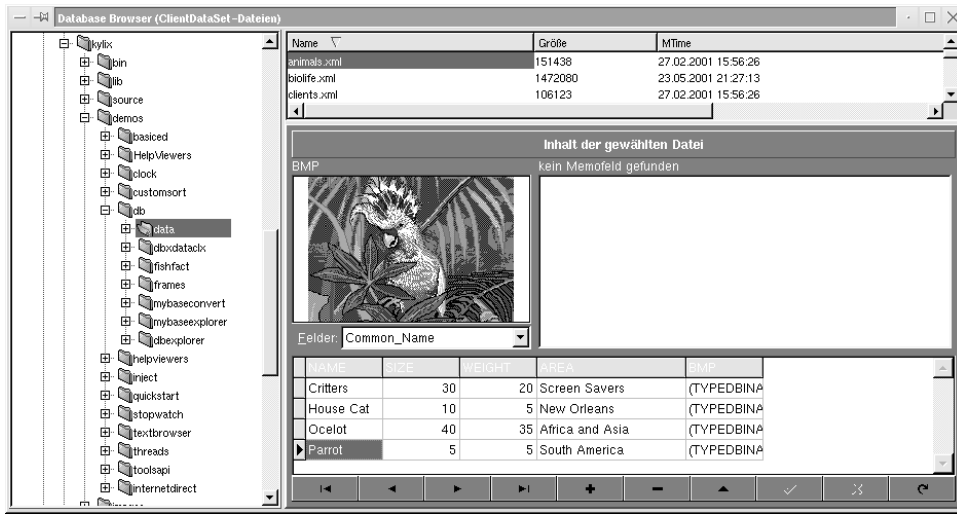


Abbildung 7.6: Der fertige Datenbank-Browser

Browser-Steuerung durch fertige Listenkomponenten

Um dem Benutzer die komfortable Auswahl einer Tabellendatei und den zügigen Wechsel zwischen verschiedenen Dateien (ohne einen *Datei-öffnen*-Dialog) zu ermöglichen, verwendet das Formular des Beispielprogramms eine Kombination aus Verzeichnisbaum und Dateitabelle, die von den Komponenten *TDirectoryTreeView* und *TFileTable* aus der Bibliothek zu diesem Buch dargestellt werden (siehe auch Kapitel 8.8.1). *TDirectoryTreeView.RootDirectory* wurde so eingestellt, dass die Komponente den gesamten Verzeichnisbaum, beginnend beim Wurzelverzeichnis, darstellt. Das *FileTable*-Property der Verzeichnisbaum-Komponente wurde auf die FileTable gesetzt, so dass diese automatisch aktualisiert wird, wenn der Benutzer ein anderes Verzeichnis wählt.

ür das Beispielprogramm selbst ist vor allem das Ereignis von Interesse, wenn der Benutzer einen neuen Eintrag der Dateitabelle wählt. Da *TFileTable* von *TListView* abgeleitet ist, verwendet es dazu das entsprechende *TListView*-Ereignis namens *OnSelectItem*. Welche Datei gewählt ist, wird wie folgt ermittelt:

- Der Name der Datei geht aus dem Property *TFileTable.Selected.Caption* hervor.
- Der Pfad der Datei ist der Pfad, der in der Tabelle angezeigt wird, also *TFileTable.Directory*.

Um den Komfort noch weiter zu steigern, ist auch noch eine *TStateSaver*-Komponente in das Beispielformular eingebaut (siehe Kapitel 4.2.3). Sie bewirkt, dass das gerade gewählte Verzeichnis in der Konfigurationsdatei *cdsBrowser.ini* gespeichert und beim

nächsten Programmstart automatisch wieder selektiert wird. Falls Sie also vorwiegend die Dateien eines bestimmten, tief geschachtelten Verzeichnisses ansehen, erspart Ihnen das, jedes Mal vom Wurzelverzeichnis beginnend die einzelnen Unterverzeichnisse im Baum suchen zu müssen.

Komponenten des Formulars

Neben den besprochenen Komponenten zur Auswahl einer Datei benötigt das Formular lediglich die grundlegenden Datenbankkomponenten: je ein Exemplar der Klassen *TClientDataSet*, *TDataSource*, *TDBGrid* und *TDBNavigator*. Sie werden miteinander verknüpft, wie in den bisherigen Kapiteln besprochen. Über diese Verknüpfung hinausgehende Property-Einstellungen sind nicht erforderlich.

Verbindungswechsel

Der Datenbank-Browser soll also beim *OnSelectItem*-Event der Dateiliste eine neue Datenbank-Verbindung aufnehmen. Dies geschieht durch Setzen des *FileName*-Properties auf die neu zu ladende Datei. Vorher muss der Browser jedoch die aktive Verbindung auflösen, indem er *TClientDataSet*.Close aufruft oder das *Active*-Property auf *False* setzt. Sobald die Daten der neuen Verbindung festgelegt wurden, kann die Datenmenge wieder aktiviert werden:

```
procedure TForm1.FileTable1SelectItem(Sender: TObject; Item: TListItem;
  Selected: Boolean);
begin
  if Selected then begin
    ClientDataSet1.Close;
    ClientDataSet1.FileName :=
             IncludeTrailingPathDelimiter(FileTable1.Directory)
                                     + FileTable1.Selected.Caption;
    ClientDataSet1.Open;
  end;
end;
```

Diese wenigen Veränderungen sorgen schon dafür, dass die Datenbankkomponenten, insbesondere das *DBGrid*, entsprechend aktualisiert werden.

Kapitel 7.3.4 wird diese extrem einfache Aktion stark erweitern, so dass auch Bildfelder der Datenbank automatisch angezeigt werden, wie in Abbildung 7.6 gezeigt.

7.2.5 Ein Browser für Interbase-Tabellen

Nun soll ein ähnlicher Browser wie im letzten Kapitel für MyBase-Dateien auch für Interbase-Dateien entwickelt werden. Das fertige Projekt finden Sie auf der CD unter dem Namen *gdbBrowser*. Das Browserformular des letzten Kapitels bedarf zweier Erweiterungen, um mit Interbase-Dateien zurecht zu kommen:

- Für den Datenzugriff wird das einfache ClientDataSet ausgewechselt durch ein Duo aus *TSQLClientDataSet* und *TSQLConnection*. Die DataSource-Komponente, deren *DataSet*-Property bisher auf das ClientDataSet wies, wird auf das SQL-ClientDataSet umgeleitet. Dessen Property *DBConnection* wird auf die *TSQLConnection*-Komponente eingestellt. Normalerweise würden Sie jetzt in dieser noch eine Verbindung auswählen oder einrichten (siehe Kapitel 7.1.4), jedoch wird dies im aktuellen Beispielprogramm programmgesteuert erledigt. Zur Entwurfszeit wird lediglich der Treibername der Komponente (Schlüssel *DriverName*) im Property *Params* auf *INTERBASE* gesetzt.
- Da Interbase-Dateien nicht nur eine einzelne Tabelle enthalten, sondern eine komplette Datenbank aus mehreren Tabellen, genügt nicht wie bisher die Auswahl einer Datei, um in der DBGrid-Komponente eine Tabelle darzustellen. Der Benutzer braucht auch eine Möglichkeit, eine Tabelle innerhalb der Datei auswählen zu können. Zu diesem Zweck finden Sie in Abbildung 7.7 unter der Dateiliste des neuen Formulars eine Kombobox, die eine Liste der in der aktuellen Datei verfügbaren Tabellen enthält.

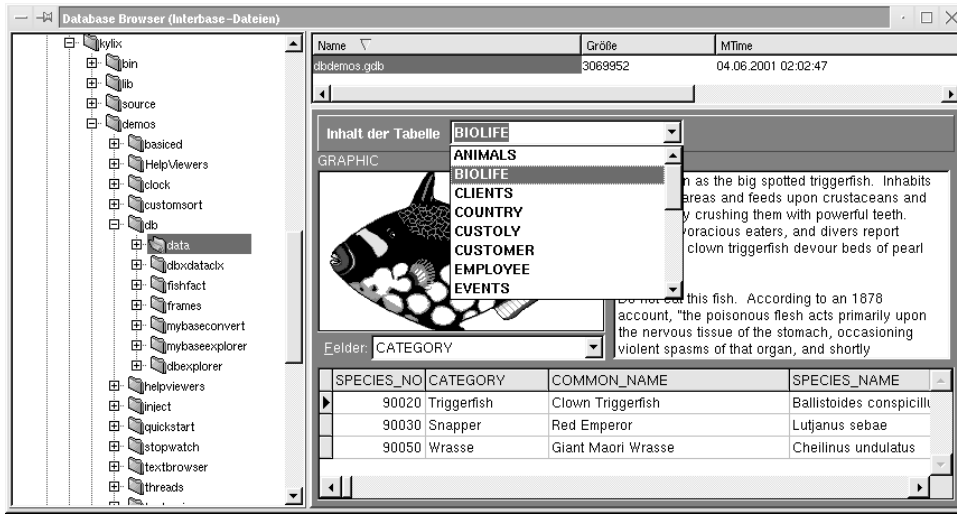


Abbildung 7.7: Der Browser für Interbase-Datenbankdateien

Das Programm hat zwei Benutzerereignisse zu verarbeiten: Die Selektierung eines anderen Dateinamens in der Dateiliste und die Auswahl eines anderen Tabellennamens aus der Kombobox. Wir beginnen mit dem Ereignis der Dateiliste.

or dem Wechsel der Tabellen und der Datenbank müssen die Verbindungen zur alten Datenbank/Tabelle deaktiviert werden; hierzu werden die Properties *TSQLClientDataSet.Active* und *TSQLConnection.Connected* abgeschaltet.

Die Definition der Verbindung geschieht im Programm durch Setzen des Objekts *TSQLConnection.Params*. Dieses enthält exakt die Werte, die Sie zur Entwurfszeit auch im Verbindungsdialog für vordefinierte Verbindungen einstellen können. So sprechen Sie etwa den Eintrag *DriverName* aus dem Dialog im Programm mit dem Ausdruck *Params.Values['DriverName']* an. Da der Treibername bereits zur Entwurfszeit auf *INTERBASE* festgelegt wurde, genügt es für das Programm, den Parameter *Database* auf die vom Benutzer gewählte Datei zu setzen. Der Dateiname selbst wird wie im Programm aus Kapitel 7.2.4 aus der *TFileTable*-Komponente ausgelesen:

```
procedure TForm1.FileTable1SelectItem(Sender: TObject; Item: TListItem;
  Selected: Boolean);
begin
  if Selected then
  begin
    SQLClientDataSet1.Active := False;
    SQLConnection1.Connected := False;
    SQLConnection1.Params.Values['Database'] :=
       IncludeTrailingPathDelimiter(FileTable1.Directory) +
       FileTable1.Selected.Caption;
    SQLConnection1.Connected := True;
```

Nun muss noch die Kombobox *cbTableNames* mit den Tabellennamen gefüllt werden. Die Verbindungskomponente bietet hierfür die Methode *GetTableNames* an, die bereits ein *TStrings*-Objekt wie das *Items*-Property der Kombobox als Parameter erwartet und es mit den Namen der Tabellen füllt. Vorher können wir der Verbindungskomponente über das Property *TableScope* noch mitteilen, welche Arten von Tabellen aufgelistet werden sollen. Die im Folgenden genannten Arten *tsTable or tsView* entsprechen der Voreinstellung, daher ist die Property-Zuweisung hier nicht wirklich notwendig (die weiteren möglichen Tabellenarten sind *tsSysTable* und *tsSynonym*):

```
  SQLConnection1.TableScope := [tsTable or tsView];
  cbTableNames.Items.Clear;
  SQLConnection1.GetTableNames(cbTableNames.Items);
  if cbTableNames.Items.Count > 0 then
    cbTableNames.ItemIndex := 0;
  cbTableNames.DroppedDown := True;
end;
```

Die letzten drei Zeilen der Methode dienen der bequemeren Handhabung der Kombobox: Falls mindestens eine Tabelle aufgelistet wurde, wird der erste Eintrag der Kombobox selektiert. Außerdem wird die Liste bereits aufgeklappt, denn das Programm verlangt in jedem Fall eine Auswahl der Benutzers, bevor es in seinem Datenbereich eine Tabelle anzeigt.

Die Auswahl des Benutzers in der Kombobox (*OnSelect*) ist das zweite zu bearbeitende Ereignis. Es ist der Bearbeitung des *OnSelectItem*-Ereignisses aus dem MyBase-Browser des letzten Kapitels ähnlich, da nur der Inhalt eines ClientDataSets neu festgelegt werden muss. An dieser Stelle handelt es sich jedoch um ein *TSQLClientDataSet*, das seine

Daten nicht aus einer MyBase-Datei (*FileName*-Property) bezieht, sondern über eine SQL-Abfrage, die über die Properties *CommandType* und *CommandText* eingestellt werden will. *CommandType* wird hier auf *ctTable* gesetzt, damit einfach die gesamte Tabelle abgerufen wird. Die dafür notwendige SQL-Anweisung wird automatisch von der Komponente erzeugt; was für uns zu tun bleibt ist, die gewünschte Tabelle im *CommandText*-Property anzugeben:

```
procedure TForm1.cbTableNamesSelect(Sender: TObject);
begin
  SQLClientDataSet1.Active := False;
  SQLClientDataSet1.CommandType := ctTable;
  SQLClientDataSet1.CommandText :=
    cbTableNames.Items[cbTableNames.ItemIndex];
  SQLClientDataSet1.Active := True;
end;
```

Außer diesen beiden Methoden enthält das komplette Beispielprogramm die gleichen Erweiterungen, die in Kapitel 7.3.4 auch dem MyBase-Browser aus dem letzten vorherigen Kapitel hinzugefügt werden (Anzeige von BLOB-Daten in zwei zusätzlichen Datenanzeigekomponenten und ein Zwischenablage-Popup-Menü für die *TDBImage*-Komponente).

Hinweis: Neben der von diesem Beispielprogramm gewählten »Browse-Strategie« gibt es noch viele andere Möglichkeiten, wie Sie durch vorhandene Datenbanken browsen können. Statt der Wahl eines Verzeichnisses und einer Interbase-Datei könnten Sie den Benutzer etwa aus den vordefinierten Verbindungen wählen lassen (wie Sie es in der Kylix-IDE selbst über den Verbindungsdialog tun können; bei einer Benutzerauswahl zur Laufzeit würden Sie dann das *ConnectionName*-Property der Verbindungskomponente auf den Namen der gewählten Verbindung einstellen). Dies hätte beispielsweise den Vorteil, dass Sie nicht nur Interbase-Datenbanken laden könnten wie das aktuelle Beispielprogramm.

Und außer der Anzeige der vorhandenen Tabellen können Sie sich von Interbase noch zahlreiche andere Metainformationen auflisten lassen, wie etwa Systemtabellen, Views, Stored Procedures. Ein ziemlich komplexes Beispiel dafür wird mit Kylix im Verzeichnis `demos/db/dbexplorer` mitgeliefert.

7.2.6 Anwenden und Verwerfen von Updates

Die in Kapitel 7.2.3 bereits vorgestellte *Post*-Methode einer Datenmenge dient als Bestätigung der Änderungen, die am aktuellen Datensatz durchgeführt wurden. *Post* kann durch den Benutzer explizit über einen Schalter einer *DBNavigator*-Komponente oder implizit durch Wechseln des aktuellen Datensatzes aufgerufen werden – oder auch durch das Programm über einen Aufruf der Methode *Post*.

Post bedeutet jedoch nicht, dass die Änderungen auch in der Datenbank gespeichert werden. Für die Speicherung der Daten müssen Sie als Entwickler in jedem Fall gesonderte Vorkehrungen treffen, wobei Sie im einfachsten Fall mit ein oder zwei Zeilen Code auskommen. Dies ist einer der grundlegenden Unterschiede der dbExpress-Architektur gegenüber der alten BDE-Architektur, bei der beispielsweise Änderungen in dBase- und Paradox-Tabellen automatisch gespeichert wurden.

Abbildung 7.8 zeigt den Weg der Änderungen eines Datensatzes in einer Client-Datenmenge von der Entstehung bis zur Speicherung in der Datenbank. Die *Post*-Operation führt zunächst dazu, dass die Änderungen des Datensatzes im Änderungsprotokoll *Delta* der Datenmenge gespeichert werden. Dabei werden die gesamten Felder des alten Datensatzes, die geänderten Felder des neuen Datensatzes und die Art der Änderung gespeichert. Als Änderungsarten gibt es

- das Verändern eines bestehenden Datensatzes,
- das Löschen eines Datensatzes (hier gibt es natürlich keinen neuen Feldwerte, die gespeichert werden könnten) sowie
- das Einfügen eines neuen Datensatzes (hier entfällt die Speicherung der alten Feldinhalte).

Kapitel 7.5.4 wird noch einmal auf dieses Änderungsprotokoll zurückkommen und erläutern, wie Sie es in einem DBGrid anzeigen und wie Sie die drei erwähnten Änderungsarten im ClientDataSet sichtbar machen können.

Vom Änderungsprotokoll gibt es zwei Wege zur festen Speicherung der Änderungen, abhängig davon, ob diese Speicherung in einer MyBase-Datei oder auf dem Datenbank-Server stattfinden soll:

- Die Änderungen der MyBase-Datei können durch einen Aufruf von *TClientDataSet.MergeChangeLog* in den festen Datenbestand übernommen werden, jedoch ist zu beachten, dass *TClientDataSet* die gesamten Daten im Hauptspeicher aufbewahrt. Eine Übertragung der Daten auf Festplatte findet nur zu zwei besonderen Anlässen statt: Falls Sie im *FileName*-Property eine Datei angegeben haben, werden die Daten automatisch gespeichert, sobald die Datenmenge geschlossen wird. Unabhängig davon können Sie durch einen *SaveToFile*-Aufruf jederzeit explizit für eine Speicherung in einer beliebigen Datei sorgen.

  Wenn die Speicherung stattfindet, bevor die Änderungen mit *MergeChangeLog* in den festen Datenbestand übernommen wurden, gehen die Änderungen nicht verloren, denn das ClientDataSet speichert auch das gesamte Änderungsprotokoll in der Datei und stellt es beim Laden komplett wieder her, so dass der Anwender mit den geänderten Daten weiter arbeiten kann und das Änderungsprotokoll fortlaufend erweitert wird, bis irgendwann *MergeChangeLog* aufgerufen wird (häufig wird *MergeChangeLog* allerdings nie aufgerufen wird; siehe Hinweis unten).

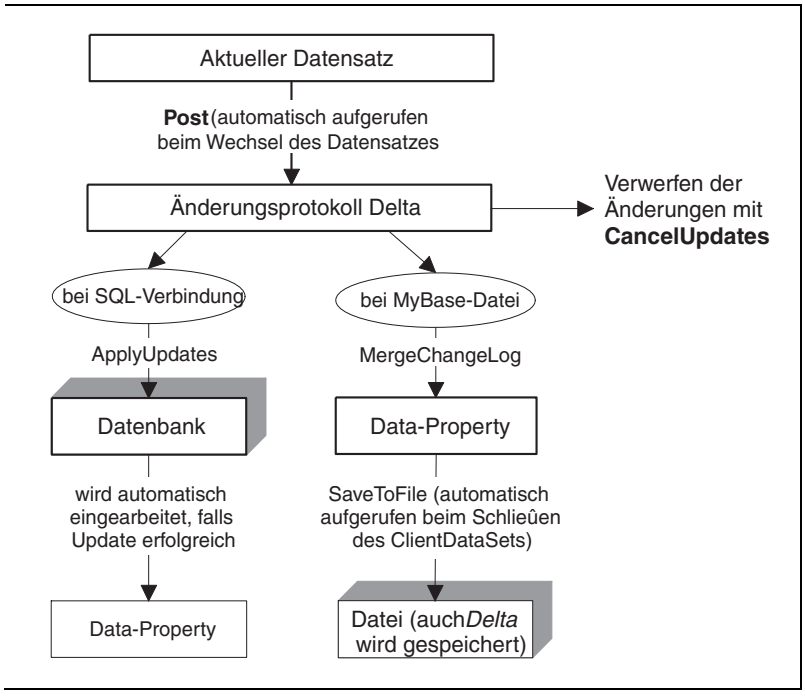


Abbildung 7.8: Verschiedene Ebenen der Änderungsspeicherung in dbExpress

- Sollen die Änderungen an einen Datenbank-Server übertragen werden, muss statt *MergeChangeLog* die Methode *TClientDataSet.ApplyUpdates* aufgerufen werden. Hierbei kann es – vor allem, wenn mehrere Benutzer gleichzeitig auf die Datenbank zugreifen – zu Update-Konflikten kommen, die in Kapitel 7.5.5 behandelt werden. An dieser Stelle genügt es zu wissen, dass die Änderungen nur dann in den im Hauptspeicher gehaltenen Datenbestand der Komponente (*Data*) übernommen werden, wenn sie mit *ApplyUpdates* auch erfolgreich auf dem Server eingetragen werden konnten.

Das Verwerfen des Änderungsprotokolls funktioniert in beiden Fällen gleich: Ein Aufruf von *CancelUpdates* löscht alle noch im Protokoll befindlichen Änderungen, und die an die Datenmenge angeschlossenen datensensitiven Komponenten zeigen automatisch wieder den Ursprungszustand der Datenmenge an (vor der ersten verworfenen nderung).

Hinweis: Beachten Sie, dass *jede* ClientDataSet-Komponente (auch *TSQLClientDataSet*) ihre Daten jederzeit mit *SaveToFile* in eine MyBase-Datei speichern kann, auch wenn die Daten eigentlich von einem Datenbank-Server stammen. Dies ist für Anwendungen erforderlich, die nach dem Aktenkoffermodell arbeiten. Hier wird zunächst – während einer Verbindung zum Server – die ClientDataSet-Menge mit Daten vom Server geladen (die ClientDataSet-Komponente bewahrt das komplette Datenpaket in ihrem *Data*-Property auf).

Nun kann die Verbindung zum Server getrennt werden und die Daten können im ClientDataSet editiert sowie mit *SaveToFile* in lokalen (»MyBase-«)Dateien gespeichert werden. Die Änderungen werden *nicht* mit *MergeChangeLog* eingearbeitet, da das Änderungsprotokoll später, wenn wieder eine Verbindung zum Server besteht, mit *ApplyUpdates* zu diesem übertragen werden muss. Nach einem Aufruf von *MergeChangeLog* wäre diese Übertragung nicht mehr möglich, so dass in dieser Situation die Änderungen mit *MergeChangeLog* quasi »verloren« gingen (zumindest für den Server, auf den es hier ja ankommt).

Bei einem Aufruf von *ApplyUpdates* wird auch das *Data*-Property des ClientDataSets aktualisiert, daher ist danach ein *MergeChangeLog* nicht mehr erforderlich (wenn das ClientDataSet mit neuen Daten vom Server geladen wird, geht der bisherige Inhalt des *Data*-Properties sowieso verloren).

7.3 Programmieren mit Feldern

Für eine Reihe wichtiger Programmieraufgaben genügt es nicht, nur mit den bisher beschriebenen Komponenten (*DataSet*-Komponenten, *DataSource* und Datensteuerelemente) zu arbeiten. Um beispielsweise bestimmte Datensätze (ohne SQL-Anweisungen) zu suchen oder zu editieren, benötigen Sie den Zugriff auf die einzelnen Felder der Tabelle.

7.3.1 TField und die Fields

Bevor eine Datenbankanwendung die Daten einer Tabelle darstellen kann, muss ihr die Struktur der Tabelle bekannt sein. Als Kenner dieser Struktur hat sich bisher besonders die *TDBGrid*-Komponente hervorgetan, denn sie erstellt automatisch alle am Bildschirm sichtbaren Spalten, die zur Darstellung der Tabellenfelder notwendig sind, passt ihre Breite an und richtet sich so auf jede Tabellenstruktur ein.

Bei allen anderen Datensteuerelement-Komponenten mussten wir das Property *DataField* auf das gewünschte Feld der Tabelle setzen. Da wir dieses aus einer Liste aller zur Verfügung stehenden Felder auswählen konnten, liegt der Gedanke nahe, dass diese Feldliste auch unter Object Pascal ansprechbar ist.

Natürlich ist sie das auch, aber das ist noch nicht alles: Die Feldliste ist nicht nur zur Laufzeit ansprechbar, sondern Sie können sie mit Hilfe des Felder-Editors und des Objektinspektors schon zur Entwurfszeit anpassen. Bei dieser Anpassung geht es jedoch nicht darum, die Struktur der Tabelle zu verändern, sondern darum, die Ausgabe der Felder in den Datensteuerelementen zu formatieren.

Die Tabellenstruktur zur Laufzeit

Unter Object Pascal ist die Felderliste aus nahe liegenden Gründen über ein Property des *DataSets* ansprechbar. Das Property heißt *Fields* und ist durch die Vererbung von *TDataSet* in allen Datenmengen-Komponenten vorhanden. Es ist wie ein Array-Property ansprechbar, das für jedes Feld der Tabelle ein *TField*-Objekt enthält. Die Zahl der definierten Felder erhalten Sie über das Property *TDataSet.FieldCount*.

Haben Sie beispielsweise eine ganz einfache Tabelle *Table1* mit den Feldern *Name* und *Vorname*, erhalten Sie mit *Table1.Fields[0]* ein *TField*-Objekt, das Informationen über das Feld *Name* enthält. *Table1.Fields[1]* oder *Table1.Fields[Table1.FieldCount-1]* gibt dementsprechend die Daten des Felds *Vorname* zurück.

Wenn Sie den Index eines Feldes nicht kennen, aber seinen Namen, können Sie das Feld mit der Funktion *FieldByName* ermitteln, beispielsweise mit *Table1.FieldByName('Schraubenzahl')*.

Tatsächlich handelt es sich beim *Fields*-Property jedoch nicht um ein Array, sondern um ein Objekt der Klasse *TFields*. In dieser Klasse finden Sie neben einigen anderen Elementen auch eine Methode *Add* (die dem Hinzufügen eines Feldes im Felder-Editor entspricht), ein Property *Count* (entspricht *TDataSet.FieldCount*) und eine Methode *FieldByName*.

Hintergrund: Zwischen dem Aufruf von *DataSet.Fields.FieldByName* und *DataSet.FieldByName* ist nur dann ein Unterschied feststellbar, wenn das *DataSet* Aggregat-Felder enthält. Diese sind nämlich nicht dem *Fields*-Objekt zugeordnet und deshalb auch nicht von seiner *FieldByName*-Methode ansprechbar.

TField-Properties zur Feldstruktur

Die folgende Tabelle listet die grundlegenden Properties der Klasse *TField* auf:

Property	Bedeutung
FieldName	gibt den Namen der Spalte an, so, wie er in der Tabelle selbst definiert ist.
Calculated	gibt mit *True* an, dass es sich nicht um ein Feld der physikalischen Tabelle handelt, sondern dass der Feldinhalt zur Laufzeit berechnet wird (siehe auch den Abschnitt *Erzeugung persistenter Felder* in Kapitel 7.3.2).

Property	Bedeutung
DataSize	enthält die Zahl der Bytes, die der Inhalt eines Felds im Hauptspeicher benötigt.
DataType	enthält eine der im nächsten Abschnitt beschriebenen Konstanten, die den Datentyp des Felds angeben.
DataSet	weist auf die *TDataSet*-Komponente, zu der das *Field* gehört.
IsIndexField	gibt an, ob der aktuelle Index dieses Feld enthält.
Index	gibt den Index des Feldes in der *Fields*-Liste an.
Lookup	Wenn *True*, handelt es sich um ein Lookup-Feld (siehe hierzu den Abschnitt *Lookup-Felder* in Kapitel 7.3.2). Wie bei *Calculated=True* handelt es sich auch bei einem Lookup-Feld nicht um ein Feld der physikalischen Tabelle. Nur eines der Properties *Calculated* und *Lookup* kann den Wert *True* haben.

TField-Typen

Abbildung 7.9 zeigt den Hierarchiebaum der *TField*-Klassen. Die DataCLX erstellt beim Öffnen einer Tabelle per Voreinstellung automatisch für jedes Feld der Tabelle ein *TField*-Objekt des passenden Typs.

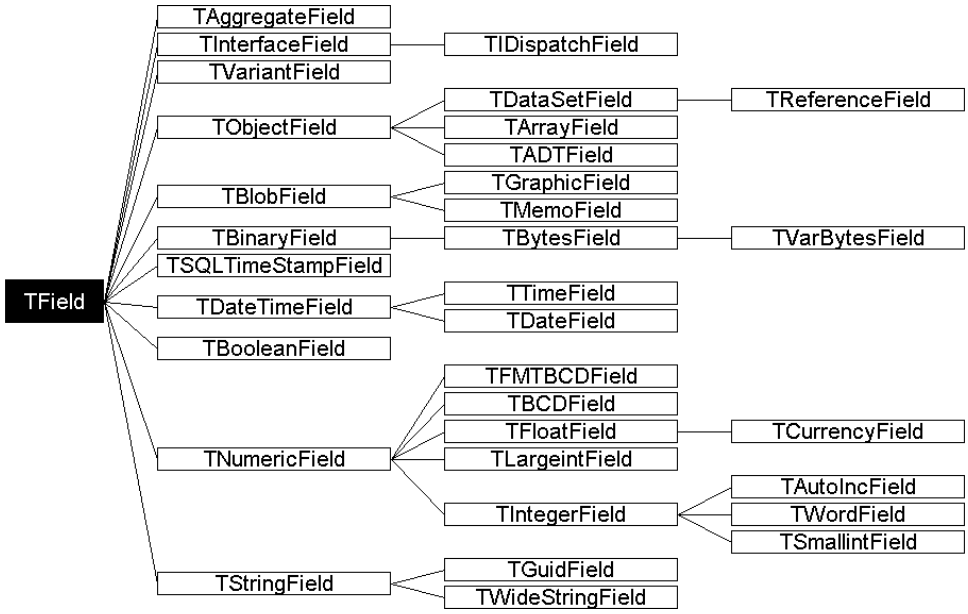


Abbildung 7.9: Die Hierarchie der TField-Klassen (abgeleitet von TComponent)

Wie bereits erwähnt, befindet sich unter den *TField*-Properties auch ein Property *DataType*. In diesem speichert die CLX den Datentyp des Feldes zusätzlich zu der Laufzeittypinformation, mit der Sie auch feststellen könnten, ob ein *TField*-Objekt beispiels-

weise die Klasse »TStringField« aufweist (zur Laufzeit-Typinformation siehe Kapitel 2.3.6). Da das Feld *DataType* jedoch wesentlich schneller anzusprechen ist und extra für die Datentypspeicherung gedacht ist, sollten Sie seine Abfrage der Verwendung der Laufzeittypinformationen vorziehen.

Die folgende Tabelle listet die wichtigsten *TField*-Klassen auf:

Klasse	Feldtyp-Konstante	Typ des Value-Properties	Bemerkung
T(Wide)StringField	ftString	(Wide)String	Textdaten, Länge abhängig von Felddefinition
TIntegerField	ftInteger	LongInt	
TSmallIntField	ftSmallInt	LongInt	Wertebereich des Object-Pascal-Typs *SmallInt* (16-Bit)
TLargeIntField	ftLargeInt	Int64	
TWordField	ftWord	LongInt	Wertebereich entspricht dem Object-Pascal-Typ *Word*
TBooleanField	ftBoolean	Boolean	True oder False
TFloatField	ftFloat	Double	Fließkommazahlen im Bereich von 5.0*10-324 bis 1.7*10308, Genauigkeit 15-16 Ziffern
TCurrencyField	ftCurrency	Double	Feldinhalt ist als Währungswert definiert, wird aber intern wie ein *TFloa-Field* dargestellt
TDateTimeField	ftDateTime	TDateTime	Datum und Uhrzeit im Format *TDateTime*
TDateField	ftDate	TDateTime	wie *TDateTimeField* im Format *TDateTime*, aber mit einer auf 0:00 Uhr eingestellten Zeit
TTimeField	ftTime	TDateTime	ebenfalls im Format *TDat-Time*, aber mit gelöschtem Datumsanteil
TBlobField	ftBlob, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary	String	binäres Datenfeld unbegrenzter Länge

Klasse	Feldtyp-Konstante	Typ des Value-Properties	Bemerkung
TMemoField	ftMemo	String	Interpretation der Blob-Daten als Text
TGraphicField	ftGraphic	- (String, von Blob geerbt)	Blob-Daten, die als Grafik (Bitmap) interpretiert werden

TField-Properties zur Anzeigesteuerung

Neben den bisher besprochenen nicht beschreibbaren Properties verfügt *TField* über eine noch größere Anzahl von Properties, die festlegen, wie die Daten des entsprechenden Feldes in den visuellen Komponenten angezeigt werden. Alle von Kylix bereitgestellten datensensitiven Komponenten richten sich nach diesen Werten.

Die folgende Tabelle behandelt die Properties aller Feldtypen, wobei viele Properties nur bei bestimmten Feldtypen vorhanden sind. So gibt es z. B. *MinValue*/*MaxValue* nur bei den Feldern, die Zahlen beinhalten. Speziell diese beiden Properties haben sogar je nach Feldtyp unterschiedliche Typen. Während ein *TIntegerField* durch *LongInt*-Werte begrenzt ist, benötigen die Grenzen eines *TFloatField* beispielsweise den Datentyp *Double*. Manche der Properties werden innerhalb der CLX nur von *TDBGrid* beachtet.

Property	Bedeutung
Alignment	wählt zwischen Linksbündigkeit, Rechtsbündigkeit und Zentrierung.
Calculated	gibt an, dass der Feldinhalt zur Laufzeit beim *TTable*/*TQuery*-Event *OnCalcFields* berechnet wird.
Currency	gibt an, ob eine datensensitive Komponente den Wert als Währung darstellen soll.
DisplayFormat	gibt für Datums-, Zeit- und Zahlenwerte einen Formatierungs-String an, der letztlich von den *SysUtils*-Funktionen *FormatDateTime* und *FormatFloat* verwendet wird, um die Darstellung des Feldwerts zu erzeugen.
DisplayLabel	gibt die Spaltenüberschrift für ein *DBGrid* an, falls diese nicht mit *FieldName* übereinstimmt.
DisplayWidth	*TDBGrid* verwendet diesen Wert, um die Breite der zum Feld gehörenden Spalte in Zeichen festzulegen.
EditFormat	steht im Gegensatz zu *DisplayFormat* und spezifiziert das Format, das nur zum Editieren des Werts verwendet werden soll.
EditMask	gibt eine Maske an, die die Eingabe beim Editieren des Felds in einer *TMaskEdit*- oder *TDBGrid*-Komponente beschränken und formatieren soll (Details siehe Online-Hilfe).
FieldName	enthält den Namen der Spalte in der Originaltabelle und stellt damit die Verbindung vom *TField* zur tatsächlichen Spalte her.
MaxValue	gibt den maximalen Wert an, der beim Editieren des Feldes eingegeben werden darf.
MinValue	schließt den von *MaxValue* begonnenen Bereich nach unten ab.

Property	Bedeutung
ReadOnly	verhindert das Editieren des Feldes in den Datensteuerelementen, wenn *True*.
Size	steht zwar im Objektinspektor, ist aber von der verwendeten Tabelle abhängig und sollte normalerweise nicht geändert werden.
g	frei verfügbarer Wertespeicher, geerbt von *TComponent*

7.3.2 Persistente Felder und der Felder-Editor

Per Voreinstellung reagiert eine DataSet-Komponente flexibel auf die Struktur einer jeden Tabelle, mit der Sie sie verbinden. Jedes Mal, wenn Sie die Tabelle (*TableName*) wechseln und die Verbindung zu ihr durch das Property *Active* einschalten, liest die DataSet-Komponente die Tabellenstruktur ein und baut so eine neue dynamische *Fields*-Liste auf. Da die *TField*-Komponenten dieser Liste viele Properties beinhalten, insbesondere solche, die sich visuell auswirken, ist es wünschenswert, die *TField*-Komponenten schon zur Entwurfszeit im Objektinspektor anpassen zu können.

Der Felder-Editor

Die Auswahl der Felder zur Anpassung im Objektinspektor zu ermöglichen ist eine Aufgabe des Felder-Editors (Abbildung 7.10), den Sie über ein Popup-Menü von Datenmengen-Komponenten oder über einen Doppelklick auf dieselben aufrufen.

Der Felder-Editor zeigt eine Liste von *TField*-Komponenten an, die zur Entwurfszeit nur über den Felder-Editor selbst ausgewählt und dann im Objektinspektor editiert werden können – ähnlich der Situation des Menü-Designers und der *TMenuItem*-Komponenten. Beim ersten Aufruf des Editors ist die Felderliste noch leer, weil alle Felder dynamisch angelegt werden.

Funktion persistenter Felder

Das entscheidende Merkmal der Liste im Felder-Editor ist, dass sie nicht bei jeder Verknüpfung mit einer anderen Tabelle dynamisch neu aufgebaut wird, sondern dass sie statisch ist, und dies sogar über einen beliebig langen Zeitraum, denn diese Felder werden in der Formulardatei gespeichert und daher auch als *persistent* bezeichnet.

Die *DataSet*-Komponente kann natürlich zur Laufzeit des Programms feststellen, ob bereits *TField*-Komponenten aus der Formulardatei gelesen werden konnten. Ist das der Fall, so zieht sie daraus einen wichtigen Schluss: Sie erzeugt im Folgenden *keinerlei* dynamische Felder mehr, egal, mit welcher Tabelle sie verbunden wird. Die persistenten Felder müssen auf jede Tabellenstruktur, der die *DataSet*-Komponente begegnet, passen. Das heißt, dass zu jeder *TField*-Komponente eine Spalte der Tabelle vorhanden sein muss, die mit dem Property *FieldName* der *TField*-Komponente übereinstimmt und einen passenden Datentyp aufweist.

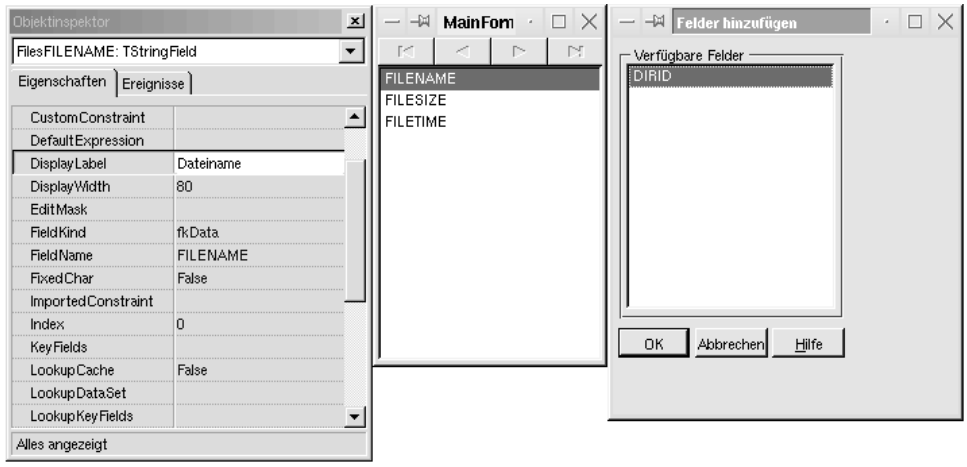


Abbildung 7.10: Der Felder-Editor ist die unscheinbare Liste in der Mitte. Der Objektinspektor zeigt die Attribute des dort gewählten Feldes FILENAME; der Felder-hinzufügen-Dialog zeigt ein weiteres Feld der Tabelle an, das noch nicht in die Liste des Felder-Editors übernommen wurde (Feld DirID).

Die Reihenfolge der Spalten in der Tabelle spielt keine Rolle, sie beeinflusst die Reihenfolge der Feld-Komponenten, die Sie im Felder-Editor festgelegt haben, nicht. Das bedeutet, dass Sie durch die Verwendung des Felder-Editors eine von der tatsächlichen Reihenfolge der Spalten unabhängige Fields-Liste erhalten.

Außerdem ist es möglich, dass die tatsächliche Tabelle mehr Spalten enthält, als *TField*-Komponenten vorhanden sind. Auch in diesem Fall erzeugt die *DataSet*-Komponente keine dynamischen *TField*-Komponenten für die unverknüpften Spalten, was zur Folge hat, dass diese Spalten beispielsweise in einem *TDBGrid* gar nicht erscheinen würden. Kapitel 7.3.3 zeigt, wie Sie diese Tabellenspalten ohne *TField*-Komponenten dennoch zur Laufzeit erkennen können.

Erzeugung persistenter Felder

Sie haben zwei Möglichkeiten, neue persistente Felder zu erzeugen:

- Wenn Sie die *DataSet*-Komponente vor dem Aufruf des Felder-Editors mit einer existierenden Datenmenge verbinden (z.B. *FileName* auf eine MyBase-Datei setzen oder in *ConnectionName* eine Verbindung und in *CommandText* eine Tabelle auswählen), kann der Felder-Editor seinerseits eine dynamische Liste aller in dieser Tabelle verfügbaren Spalten aufbauen, die er in einer seiner Listen zur Wahl stellt.

  Wählen Sie aus dem lokalen Menü des Felder-Editors den Punkt FELDER HINZUFÜGEN... und aus dem folgenden Dialogfenster die hinzuzufügenden *TField*-Komponenten. Wenn Sie einen geschlossenen Bereich wählen (in einer Maus-Ziehaktion oder mit ⇧ und Strg), können Sie mehrere Felder gleichzeitig hinzufügen.

- Sie können auch per Hand *TField*-Komponenten erstellen, indem Sie den Punkt NEUES FELD... wählen und in der folgenden Dialogbox einen Feldnamen, einen Namen für die *TField*-Komponente und einen Datentyp angeben. Auf diese Weise können Sie Felder für Tabellen definieren, die zurzeit nicht auf Ihrem lokalen Computersystem verfügbar sind (geben Sie dazu im Eingabefeld *Feldname* den Namen ein, den das zurzeit nicht verfügbare Feld in der echten Tabelle hat). Vor allem dient der Dialog aber der Definition von Lookup-Feldern, berechneten Feldern oder Feldern einer Tabelle, die Sie neu erstellen wollen (zu Letzterem siehe Kapitel 7.1.1, auf Nachschlagefelder werden wir in Kürze zurückkommen).

Um die Reihenfolge der Felder zu ändern, können Sie ihr *Index*-Property anpassen oder die Felder im Felder-Editor mit Strg + Pfeiltasten verschieben. Das von Borland empfohlene Drag&Drop konnte vom Autor noch nicht als funktionsfähig ermittelt werden.

elder im Beispielprogramm

Das Beispielprogramm *DBFile* (Kapitel 7.3.6) verwendet persistente Felder, um die Formatierung der Datei- und Verzeichnisliste zur Entwurfszeit festzulegen und um über die im Objektinspektor festgelegten Namen der *TField*-Komponenten im Quelltext auf einfache Weise auf die Felder des aktuellen Datensatzes zugreifen zu können.

Funktionen des Felder-Editors

Eine wichtige Funktion des Felder-Editors soll nicht unerwähnt bleiben: Bei der Schalterleiste unter der Feldliste handelt es sich um einen *DBNavigator*, der auf vier Schalter beschränkt wurde. Mit ihm können Sie schon zur Entwurfszeit durch die Datenmenge blättern (falls diese aktiviert und mit datensensitiven Steuerelementen verbunden ist), um etwa die Wirkung der *TField*-Properties auf die Darstellung der einzelnen Felder zu begutachten. Das Blättern ist besonders dann nützlich, wenn das Formular kein *DBGrid* enthält, sondern nur einen einzigen Datensatz gleichzeitig anzeigt.

Hinweis: Möglicherweise verfügt der Felder-Editor einer späteren Kylix-Version über eine weitere sehr nützliche Funktion, die schon seit längerem aus Delphi bekannt ist: Ziehen von Felden per Drag&Drop auf das Formular, damit dort automatisch eine passende Datenanzeigekomponente (z.B. ein *TDBEdit* für ein String-Feld) mit Beschriftung erzeugt wird.

Lookup-Felder *R125*

Lookup-Felder zeigen einmal mehr, dass ein *TField*-Objekt nicht immer mit einem wirklich in der Tabelle (bzw. in einem beliebigen *DataSet*) existierenden Datenfeld bereinstimmen muss, dass die Klasse *TField* also zunächst einmal nur ein abstraktes

Konzept beschreibt. Wie die berechneten Felder (Property *Calculated=True*) enthalten auch Lookup-Felder (*lookup* = nachschlagen, Property *Lookup=True*) keine Werte aus der Tabelle, zu der sie gehören, sondern sie schlagen ihren Wert in einer anderen Tabelle, der Lookup-Tabelle, nach.

In den Lookup-Vorgang sind drei verschiedene Felder einbezogen; in den ersten beiden Feldern können Sie statt eines einzelnen Feldes auch eine Feldliste angeben:

- Die *Schlüsselfelder* (*KeyFields*) sind ein oder mehrere Felder der *DataSet*-Komponente, in der das Lookup-Feld enthalten ist. Beim Lookup-Vorgang wird in der Lookup-Tabelle nach dem Wert gesucht, der im Schlüsselfeld des aktuellen Datensatzes enthalten ist.
- Die *Lookup-Schlüssel* (*LookupKeyFields*) sind Felder (oder ein einzelnes Feld) der Lookup-Tabelle, in denen nach dem Wert der Schlüsselfelder (bzw. des Schlüsselfeldes) gesucht werden soll. Die Lookup-Tabelle wird also quasi über ihre *Lookup-Schlüssel* mit dem Wert der Schlüsselfelder der ersten Tabelle verbunden (Ähnliches geschieht bei einem Haupt-/Detailformular (siehe Kapitel 7.4.5).
- Das *Ergebnisfeld* (*LookupResultField*) ist das Feld der Lookup-Tabelle, das schließlich den Wert des Lookup-Feldes liefert.

Im Felder-Editor können Sie ein Lookup-Feld definieren, indem Sie NEUES FELD... aus dem Popup-Menü auswählen und in der Dialogbox den Schalter NACHSCHLAGEN markieren. Die drei oben beschriebenen Felder(listen) und die Lookup-Tabelle (*LookupDataSet*) geben Sie im unteren Teil der Dialogbox ein. Die Werte des Dialogabschnitts *Feldeigenschaften* müssen außerdem mit dem Typ des *Ergebnisfeldes* übereinstimmen.

Nachdem Sie ein Lookup-Feld über diese Dialogbox erzeugt haben, können Sie alle genannten Parameter unter den in Klammern genannten Namen (*LookupDataSet*, *LookupKeyFields* usw.) auch im Objektinspektor finden, wenn Sie das Lookup-Feld im Feld-Editor auswählen.

Lookup-Beispiel

Ein Beispiel wäre z.B. eine Tabelle mit Dateien, in denen nur der Dateiname als Zeichenkette, der Pfad aber nur als Nummer eines Verzeichnisses in einer Verzeichnistabelle abgelegt ist. Wenn Sie die Dateitabelle nun etwa unverändert in einem DBGrid anzeigen, erhält der Benutzer als Verzeichnisangaben nur schwer interpretierbare Zahlen. Ein zusätzliches Lookup-Feld kann nun den Namen des Verzeichnisses in der Tabelle nachschlagen. Dafür müssen das Verzeichnisnummernfeld der Dateitabelle unter *KeyFields*, das Verzeichnisnummernfeld der Nachschlagetabelle unter *LookupKeyFields* und das Feld mit dem Verzeichnis-String als *LookupResultField* angeben werden.

In der Komponente für das Verzeichnisnummernfeld der Dateitabelle kann nun das *sible*-Property auf *False* geschaltet werden, damit die Verzeichnisnummer nicht mehr im DBGrid angezeigt wird.

Dieses Beispiel finden Sie auf der CD im Projekt der Dateidatenbank aus Kapitel 7.3.6, und zwar im Formular *LookupFieldDemo* (kann zur Laufzeit auch über FENSTER | OOKUP-DEMO aufgerufen werden).

Hinweis: Wenn Sie ein Lookup-Feld nicht wie in diesem Beispiel in einem DBGrid anzeigen, sondern das Feld mit einer *DBLookupCombobox*-Komponente (oder einer *DBLookupListBox*) verknüpfen, kann der Benutzer zur Laufzeit einen der nachgeschlagenen Werte auswählen. Ein Beispiel dafür finden Sie im Termindialog von Kapitel 7.5.3.

7.3.3 Feld-Definitionen

Im letzten Abschnitt wurde beschrieben, dass keine dynamischen Felder erzeugt werden, wenn Sie im Felder-Editor nur ein einziges persistentes Feld definieren. Das bedeutet jedoch nicht, dass Sie zur Laufzeit keinen Zugriff mehr auf die restlichen Spalten der Tabelle haben. Die *Fields*-Liste der *TDataSet*-Komponenten, die zur Laufzeit entweder die persistenten oder die dynamisch erzeugten Felder enthält, ist gar nicht dazu gedacht, die genaue Struktur der Tabelle wiederzugeben, sondern sie soll die Kylix-Anwendung von der tatsächlichen Tabellenstruktur abschirmen und die Festlegung der Darstellungs-Properties zur Entwurfszeit sowie die Definition von berechneten und Lookup-Feldern ermöglichen.

Das *TDataSet*-Property, das die tatsächliche Tabellenstruktur enthält, ist *FieldDefs*. Es sst sich wie ein indiziertes Property ansprechen, so dass Sie mit *TDataSet.FieldDefs[Index]* ein Element der Tabellenstruktur erhalten, und zwar die Definition eines einzelnen Felds, die in Form eines *TFieldDef*-Objekts vorliegt.

TFieldDefs

FieldDefs ist ein Objekt der Klasse *TFieldDefs*, das sich wie ein Array-Property ansprechen lässt. Sein Standard-Property *Items* gibt die einzelnen *TFieldDef*-Werte an und sein Property *Count* die Anzahl dieser Werte.

Im Gegensatz zu einem *TField*, in dem die allermeisten Daten nicht zur eigentlichen Felddefinition, sondern zur Darstellung auf dem Bildschirm gehören, fällt die Property-Auswahl in einem *TFieldDef*-Objekt sehr klein aus: In *Name*, *DataType* und *Size* sind genau die Daten gespeichert, die Sie auch bei der Erstellung der Tabelle in der Datenbankoberfläche angeben. Dazu kommen unter anderem bei Zahlenangaben eine Genauigkeitsangabe (*Precision*), einige Flags im Property *Attributes* (z.B. *faRequired*,

wenn das Feld nicht leer sein darf, oder *faReadOnly*, wenn es schreibgeschützt ist), und ein Property, das den Index des Felds in der Felderliste angibt (*FieldNo*).

Die Klasse *TFieldDefs* enthält außerdem zwei interessante Methoden zum Hinzufügen neuer *TFieldDef*-Objekte, von denen die Methode *Add* in Kapitel 7.1.1 bei der Erzeugung einer neuen Tabelle verwendet wird.

Hintergrund: Programmtechnisch interessant ist das Property *FieldClass*: Es enthält eine Klassenreferenz auf die Klasse der *TField*-Hierarchie, die für den Typ des Felds zuständig ist. Die Methode *TFieldDef.CreateField* hat die Aufgabe, ein *TField*-Objekt dieser Klasse zu erzeugen. Sie wird intern von der CLX verwendet, um bei der Verbindung mit einer Datenmenge, für die keine persistenten Feldkomponenten definiert wurden, dynamische Feldkomponenten zu erzeugen (zu jedem Element von *FieldDefs* wird dann eine Feldkomponente in *Fields* erzeugt).

7.3.4 Ein Beispielprogramm mit dynamischen Feldern

Wir setzen an dieser Stelle das in Kapitel 7.2.4 begonnene Beispielprogramm fort. Ein Nachteil des Datenbank-Browsers war bisher, dass er die Daten zwar sehr übersichtlich in einer *DBGrid*-Tabelle anzeigt, diese aber nicht alle Datenelemente darstellen kann: Grafiken, Memos und andere BLOBs erhalten zwar eine eigene Spalte, diese enthält jedoch immer nur einen ersatzweisen Text wie »(Memo)« oder »(Blob)«.

Mit der hier beschriebenen Erweiterung durchsucht der Browser die Struktur einer beliebigen Tabelle zur Laufzeit nach einem Grafik- und einem Memofeld. Das letzte gefundene Feld jedes dieser Typen stellt er jeweils in einem statisch im Formular eingebauten Datensteuerelement dar (einer *TDBImage*- bzw. einer *TDBMemo*-Komponente).

Einbau der Komponenten

Zu den schon in Kapitel 7.2.4 erwähnten Komponenten kommen in der vollständigen Version des Programms die folgenden hinzu:

- *cbFields* ist eine Kombobox, in der das Programm alle Felder der gerade gewählten Datenbank auflistet: Für jedes in *FieldDefs* angegebene Feld erweitert das Programm die Stringliste (*Items*) dieser Kombobox um den Namen dieses Felds.
- *DBMemo* ist die statische Memo-Komponente, die mit dem letzten Memo-Feld der Feldliste verbunden wird
- *MemoLabel* ist die Beschriftung für dieses Feld (sie gibt den Feldnamen an).
- *DBImage* zeigt dementsprechend das zuletzt gefundene Feld der Datenbank an, das Bilddaten enthält
- *GraphicLabel* zeigt den Namen dieses Feldes an.

Zunächst zerlegen wir die Aufgabe, die bei jeder neuen Dateiauswahl durchgeführt wird, in drei Teile: die gegenüber Kapitel 7.2.4 kaum veränderte Methode für das *OnSelectItem*-Ereignis der Dateiliste und die Methoden *Activate* und *Deactivate*:

```
procedure TForm1.FileTable1SelectItem(Sender: TObject; Item: TListItem;
  Selected: Boolean);
begin
  Deactivate;
  ClientDataSet1.FileName :=
      IncludeTrailingPathDelimiter(FileTable1.Directory) +
      FileTable1.Selected.Caption;
  Activate;
end;
```

Dynamische Abfrage *R131*

Activate ist die wichtigste Methode, denn sie greift intensiv auf die *TFieldDefs*-Daten zu, die sofort nach dem Setzen des *Active*-Properties auf *True* zur Verfügung stehen und die Struktur der in *FileTable1SelectItem* gesetzten Datenmenge beschreiben.

Activate durchläuft alle Felddefinitionen und hängt in jedem Fall den Namen des Felds an die Kombobox an. Dann überprüft Sie den Feldtyp und verbindet das Feld mit dem *DBImage* bzw. dem *DBMemo*, falls es den passenden Typ aufweist (gibt es mehrere Felder desselben Typs, so überschreibt die Methode die bisher aufgebaute Feldverknüpfung einfach mit dem jeweils letzten Feld des Typs):

```
procedure TForm1.Activate;
var
  i: Integer;
begin
  ClientDataSet1.Open;
  with ClientDataSet1 do
    for i := 0 to FieldDefs.Count-1 do begin { für jedes Feld: }
      cbFields.Items.Add(FieldDefs[i].Name);   { Liste erweitern }
      if FieldDefs[i].DataType in [ftGraphic, ftBlob, ftTypedBinary] then
      begin
        { Graphics und Blobs als Grafik darstellen }
        DBImage.DataField := FieldDefs[i].Name;
        GraphicLabel.Caption := FieldDefs[i].Name;
      end
      else if FieldDefs[i].DataType = ftMemo then begin
        { Memofeld gefunden }
        DBMemo.DataField := FieldDefs[i].Name;
        MemoLabel.Caption := FieldDefs[i].Name;
      end;
    end;
end;
```

Lösen der Verbindung

Das vollständige Lösen der in *Activate* aufgebauten Verbindungen besteht nun nicht mehr nur im Abschalten von *Active*, sondern auch im Löschen der *DataField*-Properties der Datensteuerelemente. Dies ist notwendig, damit es beim Öffnen einer anderen Tabelle nicht sofort zu einem Fehler kommt:

```
procedure TForm1.Deactivate;
begin
  ClientDataSet1.Active := False;
  Felder.Clear;
  DBImage.DataField := '';
  DBMemo.DataField := '';
  GraphicLabel.Caption := 'kein Grafikfeld gefunden';
  MemoLabel.Caption := 'kein Memofeld gefunden';
end;
```

TDBImage-Methoden

Die *TDBImage*-Komponente erhält im Beispielprogramm noch ein kleines Popup-Menü, in dem Sie zur Laufzeit mit der Zwischenablage arbeiten und das *Stretch*-Flag der Grafik umschalten können (*DBImage.Stretch*). Die Menüpunkte für die Zwischenablage erfordern jeweils nur den Aufruf einer der *DBImage*-Methoden *CopyToClipboard*, *PasteFromClipboard* und *CutToClipboard*, so dass sich ein Listing erübrigen dürfte.

7.3.5 Die Daten des aktuellen Datensatzes

In Kapitel 7.2.3 wurden zwar schon einige Methoden vorgestellt, die den aktuellen Datensatz auswählen, für den eigentlichen Zugriff auf den Datensatz waren jedoch die Datenbank-Komponenten zuständig: Sie lesen den Inhalt des aktuellen Datensatzes automatisch aus und zeigen ihn an. Auch wenn der Benutzer Daten ändert, werden die Änderungen automatisch verarbeitet, sobald er das Feld verlässt oder den *Post*-Schalter drückt. Sie müssen lediglich sicherstellen, dass die Änderungen aus dem Änderungsprotokoll auch in die Datenbank geschrieben werden, brauchen die Daten aber oft nicht durch eigenen Programm-Code zu ändern.

Wenn Sie einzelne Daten durch das Programm auslesen lassen oder Datensätze programmgesteuert erzeugen wollen, benötigen Sie jedoch Zugriff auf den Inhalt des aktuellen Datensatzes. Wie bereits erwähnt, ist dieser auch schon der einzige Datensatz, den Sie gleichzeitig ansprechen können. Zum Ansprechen von mehreren Datensätzen müssen Sie ähnlich vorgehen wie der Benutzer einer Datenbankanwendung: Datenzugriff, einen anderen Datensatz ansteuern und dann auf dessen Daten zugreifen. Allerdings haben Sie im Programm einige zusätzliche Möglichkeiten, wie Lesezeichen zu setzen und die Aktualisierung der Anzeigen zeitweise auszuschalten.

ür den Zugriff auf den Inhalt des aktuellen Datensatzes benötigen wir ein weiteres ündel von *TField*-Properties.

alue

Zunächst einmal finden Sie in den verschiedenen von *TField* abgeleiteten Typen jeweils ein Property namens *Value*, dessen Typ dem Datentyp des Felds entspricht. Falls Sie B. ein Feld des Typs *ftInteger* beschreiben wollen und dementsprechend ein Feld des Typs *TIntegerField* vorliegen haben, hat dessen Property *Value* den Typ Integer.

Wenn Sie mit persistenten Feldern arbeiten, die Sie im Felder-Editor erzeugt haben, önnen Sie diese besonders einfach über ihre Namen ansprechen, so dass das folgende Beispiel den Wert 16 in ein Feld schreibt:

```
DatenbanknameFeldname.Value := 16;
```

Dabei soll *DatenbanknameFeldname* für den von Kylix automatisch erzeugten Namen des persistenten Felds stehen, den Sie im Objektinspektor ändern können, wie bei jeder anderen Komponente auch.

Auch die weitere Vorgehensweise nach dem Schreiben der Daten entspricht der des Benutzers. Sie können also die *DataSet*-Methode *Post* aufrufen, um die Änderungen sofort zu speichern, oder darauf warten, dass die *DataSet*-Komponente das automatisch tut, wenn der aktuelle Datensatz gewechselt wird.

alue als Variante

Um auf *Value* zugreifen zu können, benötigen Sie kein Objekt einer speziellen *TField*-Klasse, denn auch die abstrakte Klasse *TField* verfügt bereits über ein *Value*-Property, das jedoch keinen speziellen Datentyp hat, sondern eine Variante darstellt. Sie können diesem Property also nahezu beliebige Datentypen zuweisen, und die CLX versucht dann, falls notwendig, diese in den passenden Typ umzuwandeln.

Da das *Fields*-Array einer *DataSet*-Komponente die einzelnen *TField*-Komponenten als *TField*-Objekt zurückgibt, können Sie wie folgt (ohne eine explizite Typenumwandlung) auf das *Value*-Property eines solchen Objekts zugreifen:

```
Table.Fields[i].Value := 16;
```

ormatumwandlungs-Properties

Jedes *TField*-Objekt verfügt über einen Satz von Properties, die es Ihnen ermöglichen, den Feldwert in verschiedenen Datentypen anzugeben und abzufragen: *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger* und *AsString*. Die Feldtypen, die beispielsweise in einem *DBGrid* vollständig als Text dargestellt werden können, können auch über *AsString* geändert und gelesen werden. Die meisten Feldtypen können sehr flexibel ineinander

umgewandelt werden. Nur in wenigen Fällen müssen Sie mit Problemen rechnen, so etwa wenn Sie versuchen, das Property *AsDateTime* bei einem Feld zu verwenden, das weder ein String- noch ein Datums- oder Zeitfeld ist.

Um auf das obige Beispiel zurückzukommen: Mit *AsInteger* können Sie den Feldwert auch ohne Verwendung einer persistenten *TField*-Komponente und ohne explizite Typenumwandlung und Verwendung des Properties *Value* ansprechen:

```
Table.Fields[Feldindex].AsInteger := 16;
```

Das *FileDB*-Beispiel aus dem nächsten Kapitel gibt Beispiele für die praktische Verwendung der Properties *AsInteger* und *AsString*.

Rufen der dynamischen Felder beim Namen

Die letzte Beispielzeile kann noch etwas lesbarer gestaltet werden. Schwierig zu verstehen ist besonders der *Feldindex*, denn normalerweise werden die Felder bei ihrem Namen genannt.

Tatsächlich können Sie die Felder auch in dieser Situation beim Namen nennen, auch ohne sie persistent im Felder-Editor definiert zu haben. *TDataSet* definiert zu diesem Zweck die Methode *FieldByName*, der Sie den Namen des gewünschten Felds als String übergeben und die das gefundene Feld als Ergebnis des Typs *TField* zurückliefert:

```
Table.FieldByName('Feldname').AsInteger := 16;
```

FieldByName erzeugt eine Exception, wenn der angegebene Feldname nicht existiert. Wenn Sie in diesem Fall lieber das Ergebnis *nil* empfangen wollen, verwenden Sie stattdessen die genauso aufrufbare *TDataSet*-Methode *FindField*.

Es ist noch eine weitere Vereinfachung möglich: Sie können den Feldnamen quasi als Index in eckigen Klammern direkt hinter den Tabellennamen schreiben (damit sprechen Sie *FieldValues* an, das Standard-Array-Property von *TDataSet*). Wie schon bei Verwendung des Properties *TField.Value* erhalten Sie auch hier eine Variante. Zuweisung und Auslesen eines Wertes sehen dann wie folgt aus:

```
Table['Feldname'] := 16;
AusgelesenerWert := Table['Feldname'];
```

Arbeiten in der Tabelle

Wenn die Kylix-Anwendung selbst mehrere Datensätze verändert, zeigt sich ein Nebeneffekt der Tatsache, dass es immer nur einen Datensatz gibt, auf den Programm und Anwender gleichzeitig zugreifen können. Wenn das Programm einen anderen Datensatz auswählt, um in diesem einige Änderungen vorzunehmen, verschwindet auch der für den Benutzer aktuelle Datensatz aus dem Blickfeld, stattdessen zeigen die Datensteuerelemente den vom Programm angesprochenen Datensatz an.

Oft ist es jedoch für den Benutzer angenehmer, von diesen Hintergrundaktivitäten des Programms nicht allzuviel mitzubekommen, und so empfiehlt es sich, bei längeren Arbeiten in der Tabelle die Aktualisierung der Datenanzeige-Komponenten zu unterbinden. Hierzu sieht *TDataSet* die Methoden *DisableControls* und *EnableControls* vor. Mit *DisableControls* trennen Sie alle visuellen Elemente, die mit der Tabelle verbunden sind, kurzfristig von dieser ab, so dass das Programm seinen Auftrag hinter den Kulissen ausführen kann. Danach schalten Sie die Komponenten mit *EnableControls* wieder an.

Ein wichtiger Vorteil dabei ist natürlich auch, dass die Aktualisierung einer Vielzahl von Datensätzen spürbar schneller abläuft, wenn die Tabellen im Formular nicht bei jedem Datensatz-Wechsel aktualisiert werden müssen. Im Beispielprogramm *FileDB* (nächstes Kapitel) können Sie diesen Geschwindigkeitsunterschied bei laufender Operation testen, indem Sie die Aktualisierung der Komponenten über einen Markieungsschalter manuell an- und abschalten.

7.3.6 Eine Dateidatenbank

Ein Beispiel für eine Datenbankanwendung, die ausgiebig selbst Datensätze erstellt, ist *FileDB* (Abbildung 7.11). Sie liest komplette Verzeichnisbäume mitsamt den enthaltenen Dateien ein und legt diese in einer Datenbank ab, die aus zwei über eine Relation verknüpften Tabellen besteht:

In der Dateitabelle sollen zu jeder Datei neben dem Namen (Feld *Dateiname*) auch die *Größe* und das *Datum* (der Zeitstempel des Betriebssystems) gespeichert werden. Wichtig ist natürlich auch das Verzeichnis, in dem sich die Datei befindet. Da Verzeichnisse ber hundert Dateien haben und Verzeichnispfade sehr lang werden können, wäre es eine große Platzverschwendung, für jede Datei ein eigenes, 80 Zeichen langes Feld für den Verzeichnispfad zu reservieren. Besser ist es, eine zweite Tabelle zu definieren, die jedes Verzeichnis genau einmal aufführt und mit einer Nummer versieht.

Die Verzeichnistabelle enthält nur zwei Felder: Verzeichnisnummer (Integer) und Verzeichnispfad (vollständige Verzeichnisangabe als String). Da der Pfad jedes Verzeichnisses in dieser Tabelle nachgeschlagen werden kann, beschränkt sich die Angabe des erzeichnisses in der Dateitabelle einfach auf die Nennung der Verzeichnisnummer.

Die folgende Tabelle listet die Namen der Felder auf, wie sie auf Datenbankebene definiert wurden, und durch welche persistenten Feldkomponenten sie in der Kylix-Anwendung repräsentiert werden:

abelle Dirs	**Tabelle Files**
DirID (DirsDirID: TStringField)	**DirID** (FilesDirID: TIntegerField)
Path (DirsPath: TStringField)	**FileName** (FilesFileName: TStringField)
	FileTime (FilesFileTime: TSQLTimeStampField)
	FileSize (FilesFileSize: TIntegerField)

Das SQL-Skript, mit dem diese Datenbank erstellt werden kann, wurde bereits in Kapitel 7.1.2 gezeigt.

Version 1

Die erste Version der Methoden, die hier abgedruckt ist, unterscheidet sich allerdings noch deutlich von der Endversion. Ohne die in Kapitel 7.4.4 hinzugefügten Verbesserungen eignet sich die Datenbank nur zum Einlesen der Verzeichnis- und Dateidaten in eine völlig neue Tabelle, denn sie berücksichtigt in keiner Weise schon vorhandene Einträge, die unter Umständen nur aktualisiert zu werden bräuchten. Wenn Sie mit dieser Version versuchen würden, eine bestehende Datenbank zu aktualisieren, käme es häufig zu Indexfehler-Exceptions, denn das Programm würde versuchen, schon existierende Schlüssel (bestehend aus Verzeichnisnummer und Dateiname) zu verdoppeln. Die notwendigen Erweiterungen können erst hinzugefügt werden, nachdem Kapitel 7.4.2 das Thema der Indizes erläutert hat.

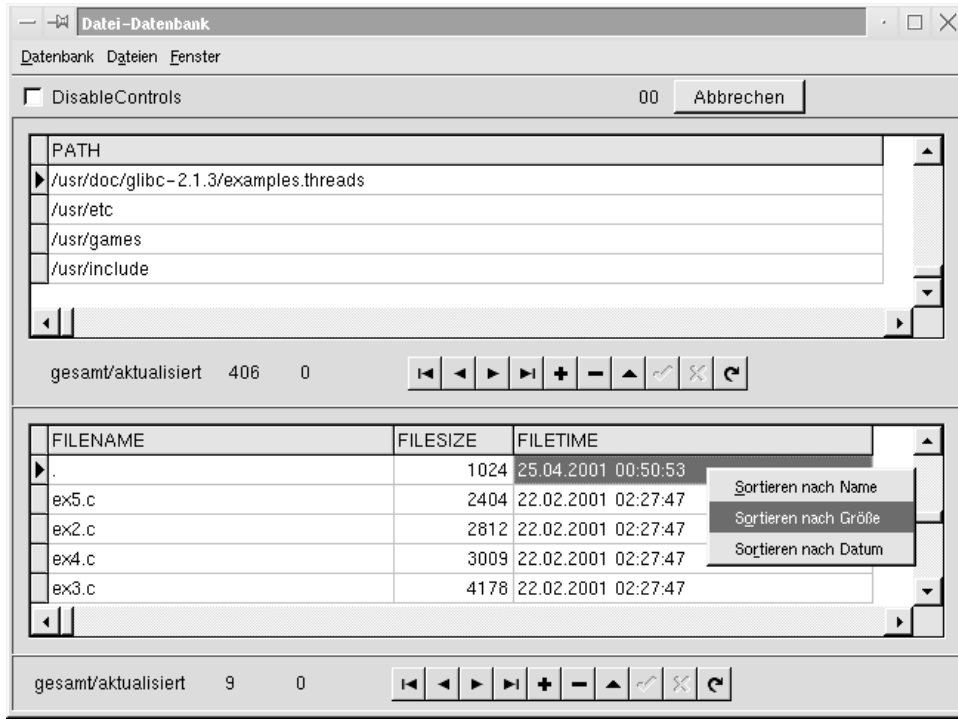


Abbildung 7.11: Das Beispielprogramm FileDB

Das Formular

Das Hauptformular von *FileDB* wird wie folgt entworfen:

- Es wird von vier Panels ausgefüllt, von denen die beiden ersten mit *alTop*, die letzte mit *alBottom* und die vorletzte mit *alClient* ausgerichtet ist. Dies hat zur Folge, dass Größenänderungen des Fensters immer zu Gunsten oder zu Lasten des zweiten *DBGrids* gehen, das sich in diesem dritten Panel befindet.
- Eine *TSQLConnection*-Komponente, in deren *Params*-Property der Treibertyp *INTERBASE* und die Datenbank *FileDB.gdb* im Verzeichnis des Beispielprogramms angegeben wird.
- Je zwei Exemplare von *TClientDataSet*, *TDataSource*, *TDBGrid* und *TDBNavigator* sorgen dafür, dass die beiden Tabellen unabhängig voneinander angesprochen werden können.
- In den Properties der *TSQLClientDataSet*-Komponenten *Files* und *Dirs* wird zunächst die *DBConnection* auf die *SQLConnection*-Komponente gesetzt. Dann erhalten die *CommandType*-Properties den Wert *ctTable* und in *CommandText* wird die jeweilige Tabelle ausgewählt (»Files« oder »Dirs«). Zur Laufzeit muss dann nur noch das *Active*-Property eingeschaltet werden, um die Tabellen zu aktivieren. Falls die Tabellen noch nicht existieren, können Sie sie zur Laufzeit mit dem Menüpunkt DATENBANK | DATENBANK UND TABELLEN PER SKRIPT ERZEUGEN erzeugen. Die beiden Tabellen sind zu einem Haupt-/Detailgespann verbunden, das jedoch erst in Kapitel 7.4.5 Bedeutung erlangt.
- Im Felder-Editor werden alle Felder der Tabellen als persistente Felder definiert, so dass wir diese im Quelltext über einen festen Namen ansprechen können.
- Die weiteren Bestandteile und Eigenschaften des Formulars erschließen sich von selbst, bis auf einige Details, die in späteren Kapiteln besprochen werden (im Zusammenhang mit der Aufteilung in Haupt- und Detail-Tabelle).

Wir beginnen mit den wichtigsten Methoden des Formulars, und zwar denen, die die eigentliche Erweiterung der Datenbank übernehmen (*StoreFiles* und *StoreDirs*), und sehen uns erst danach an, wie die Suche der Dateien abläuft und wie die beiden Methoden aufgerufen werden.

Programmgesteuertes Erzeugen von Datensätzen R132

StoreFile erhält als Parameter den Namen, die Größe und den Zeitstempel einer Datei sowie die Nummer des Verzeichnisses, in dem sie gespeichert ist. Sie fügt einen neuen Datensatz in die Tabelle *Files* ein, setzt die einzelnen Felder über die in Kapitel 7.3.5 besprochenen *As...*-Properties auf die in den Parametern erhaltenen Werte und speichert den Datensatz mit *Post*:

```
procedure TMainForm.StoreFile(Dir: LongInt; Name: string;
  Size: LongInt; Date: TDateTime);
{ gekürzte Version }
begin
  Files.Insert;
  FilesFileTime.AsDateTime := Date;
  FilesFileName.AsString := Name;
  FilesDirID.AsInteger := Dir;
  FilesFileSize.AsInteger := Size;
  Files.Post;
end;
```

StoreDir erhält einen Verzeichnispfad als Parameter, gibt diesem eine Nummer, speichert Nummer und Pfad in der Tabelle *Dirs* und liefert die Nummer als Ergebnis zurück. (Zur Vergabe der Verzeichnisnummer kommen wir in Kapitel 7.4.4, für diese gekürzte Version würde es genügen, die bisher letzte Nummer in einer globalen Variablen zu speichern und in jedem *StoreDir*-Aufruf um eins zu erhöhen.)

```
function TMainForm.StoreDir(Path: string): LongInt;
{ gekürzte Version }
begin
  Dirs.Insert;
{    DirIndex := MaxDirIndex;   siehe vollständige Version }
  DirsDirID.AsInteger := DirIndex;
  inc(DirIndex)
  DirsPath.AsString := Path;
  Dirs.Post;
end;
```

StoreFile und *StoreDir* bewirken mit ihrem Aufruf von *Post* lediglich, dass die neuen Datensätze in das Änderungsprotokoll der jeweiligen Datenmenge eingefügt werden. Die Übernahme der Änderungen in die Datenbank-Datei mit Hilfe der *ApplyUpdates*-Methode erfolgt für jeden Einlesevorgang nur ein einziges Mal, und zwar in der als Nächstes gezeigten Methode.

Einlesen der Verzeichnisse

Zum Starten des Verzeichnis-Einlesevorgangs benötigen wir zuerst einmal ein Ereignis. *FileDB* stellt dazu den Menüpunkt DATEIEN | HINZUFÜGEN bereit, der in der folgenden Methode bearbeitet wird:

```
procedure TMainForm.AddFilesClick(Sender: TObject);
{ vollständige Version }
begin
  if DirSelect.ShowModal = mrOK then begin
    AddFromDir(DirSelect.DirectoryListBox1.Directory);
  end;
```

```
  Dirs.ApplyUpdates(-1); // Alle Änderungen in die Datenbank übernehmen
  Files.ApplyUpdates(-1); // (zum Parameter "-1" siehe Kapitel 7.5.4)
end;
```

Diese Methode ruft eine hier nicht weiter interessante Dialogbox auf, aus der sie das vom Benutzer ausgewählte Verzeichnis ausliest und es an die Methode weitergibt, die die eigentliche Arbeit erledigt: *AddFromDir*. Diese führt einige Verwaltungsaufgaben durch, die in Kapitel 7.3.7 beschrieben sind, und delegiert das gesamte Einlesen an die Methode *InsertDir*, die den Pfad des Verzeichnisses ebenfalls als Parameter erhält:

```
procedure TMainForm.InsertDir(Path: string);
{ vollständige Version }
var
  SearchRec: TSearchRec;
  Error: Integer;
  DirIndex: LongInt;
  DateTime: TDateTime;
begin
  Error := FindFirst(IncludeTrailingPathDelimiter(Path)+Mask,
                     faAnyFile, SearchRec);
  SearchRec.Name := LowerCase(SearchRec.Name); { nur Kleinbuchstaben }
  DirIndex := StoreDir(Path); { * Neuer Datensatz * }
  Count.Caption := IntToStr(DirIndex);
  while (Error = 0) and not CancelRequest do begin
    UpdateLabels; { Statusanzeige über den Einlesevorgang erneuern }
       { UpdateLabels: siehe vollständige Datei auf CD-ROM }
    { nur Kleinbuchstaben: }
    SearchRec.Name := LowerCase(SearchRec.Name);
    DateTime := 0; { manche Dateien (z.B. verschiedene ".."-Einträge }
    try            { haben kein Datum. SearchRec.Time ist dann 0 }
                   { und FileDateToTime erzeugt eine Exception }
      DateTime := FileDateToDateTime(SearchRec.Time)
    except       { ... diese Exception einfach ignorieren. }
    end;
    { * Neuer Datensatz: * }
    StoreFile(DirIndex, SearchRec.Name, SearchRec.Size, DateTime);
    if (SearchRec.Name <> '.') and (SearchRec.Name <> '..') then
       { Die Verzeichnisse "." und ".." nicht beachten }
      if (SearchRec.Attr and faDirectory) <> 0 then
        InsertDir(IncludeTrailingPathDelimiter(Path)+SearchRec.Name);
    Error := FindNext(SearchRec);
  end;
  FindClose(SearchRec);
end;
```

InsertDir speichert das gewählte Verzeichnis mit der Methode *StoreDir*, die die Nummer des Verzeichnisses zurückliefert. Diese Nummer kann dann für jede der im Verzeichnis enthaltenen Dateien an die Methode *StoreFile* übergeben werden. *InsertDir* findet diese Dateien mit Hilfe der Standardfunktionen *FindFirst* und *FindNext* (siehe Kapitel 2.8.1).

Wenn es sich bei der mit *FindFirst* oder *FindNext* gefundenen Datei um ein Verzeichnis handelt (*SearchRec.Attr* enthält dann das Bit *faDirectory*), ruft *InsertDir* sich selbst rekursiv auf, damit auch dieses Unterverzeichnis vollständig in die Datenbank aufgenommen wird.

7.3.7 Zusatzfunktionen für die Anwendung

Die Anwendung *FileDB* soll, während sie einen Verzeichnisbaum einliest, auch Ereignisse des Benutzers verarbeiten können. Zwar soll der Benutzer während dieses Vorgangs keine der Tabellen verändern können, aber zwei Dinge sollen ihm erlaubt sein:

- An- und Abschalten der laufenden Aktualisierung der *DBGrid*-Komponenten über den Markierungsschalter *DisableControls*.
- Und, was noch wichtiger ist, er soll den Einlesevorgang abbrechen können.

Für beides muss der Benutzer einen (Markierungs-)Schalter drücken. Das Programm kann dieses Drücken des Schalters jedoch nicht bearbeiten, wenn es gleichzeitig bereits die Dateien einliest (dieses Einlesen der Dateien ist ja bereits die Bearbeitung eines Ereignisses – und zwar des *OnClick*-Ereignisses für den Menüpunkt DATEIEN | HINZUFÜGEN...).

ProcessMessages *R133*

Damit der Benutzer die Einlesefunktion dennoch jederzeit abbrechen kann und auch das Drücken des *DisableControls*-Schalter sofort Wirkung zeitigt, muss das Programm quasi mit sich selbst Multitasking durchführen – es wird den Einlesevorgang in kurzen Intervallen unterbrechen, indem es die Methode *Application.ProcessMessages* aufruft, während der alle inzwischen angefallenen Nachrichten bearbeitet werden.

Ausführliche Informationen zu *ProcessMessages* und zum Multitasking allgemein gibt Kapitel 4.8. Wir verwenden hier keine Threads, da sonst wieder das Problem der Synchronisierung mit der CLX auftauchen würde (die ständige Verwendung der *Synchronize*-Methode würde auf das hinauslaufen, was wir auch mit *ProcessMessages* erreichen können: dass der Haupt-Thread abwechselnd Nachrichten bearbeitet und den Verzeichniseinlesevorgang fortsetzt).

Der Aufruf von *Application.ProcessMessages* findet in der vollständigen Methode *StoreFiles* (Kapitel 7.4.4) statt, also je einmal pro Datei, die der Datenbank hinzugefügt wird. Diese Häufigkeit stellt sicher, dass die Anwendung eingehende Nachrichten quasi sofort bearbeitet kann.

Absichern des Einlesevorgangs vor störenden Ereignissen

Der Benutzer kann nun also, während die Verzeichnisse eingelesen werden, Eingaben an das Formular senden. Der Aufruf bestimmter Menüpunkte würde jedoch das Einlesen massiv stören (z.B. der Befehl DATENBANK | NEU). Alle Kontrollelemente und Menüpunkte, die beim Einlesen nicht verwendet werden dürfen, sollten daher vorher deaktiviert werden. Dies ist Aufgabe der Methode *AddFromDir*.

Eine Möglichkeit, jegliche Eingabe in das Formular zu verhindern, wäre, es vollständig zu deaktivieren (Property *TForm.Enabled*). Dann könnten Sie jedoch auch die Schalter ABBRUCH und DISABLECONTROLS nicht mehr drücken.

Abbrechen des Vorgangs

Ein langwieriger Vorgang wie das Einlesen mehrerer tausend Dateien in eine Datenbank wird zwar oft als Hintergrundoperation implementiert, so dass er das System nicht mehr blockiert, trotzdem sollte er auch abgebrochen werden können. *FileDB* stellt hierzu den Schalter ABBRECHEN zur Verfügung, dessen *OnClick*-Methode die interne Formularvariable *CancelRequest* auf *True* setzt. Die oben gezeigte Methode *InsertDir* fragt diese Variable nach jeder einzelnen eingefügten Datei ab und bricht den Einlesevorgang sofort ab, wenn sie den Wert *True* annimmt.

Die Methode AddFromDir

Die Methode *AddFromDir* sorgt also dafür, dass vor dem Einlesen der Verzeichnisse alle kritischen Komponenten deaktiviert werden. Der *try...finally*-Block stellt sicher, dass sie selbst dann wieder aktiviert werden, wenn es zu einer Exception kommt, durch die die Ausführung der Methode abgebrochen wird:

```
procedure TMainForm.AddFromDir(Dir: string);
{ vollständige Version }
var
  i: Integer;
begin
  { Datenbankverbindung einschalten }
  Files.Active := True;
  Dirs.Active := True;
  { Menüpunkte deaktivieren }
  ClearDB.Enabled := False;
  CreateDB.Enabled := False;
  AddFiles.Enabled := False;

  Count.Visible := True;   { Zähler anzeigen }
  { internes Flag und Zähler setzen }
  CancelRequest := False;
  FileUpdates := 0;   DirUpdates := 0;
  { Synchronisation von Haupt-/Detailtabelle abschalten }
```

```
    Files.MasterSource := nil; { MasterSource siehe 7.4.5 }
    try
      InsertDir(Dir);
    finally
      CancelRequest := False;
      ClearDB.Enabled := True;
      CreateDB.Enabled := True;
      AddFiles.Enabled := True;
      Count.Visible := False;
      Files.MasterSource := DirSource;
    end;
  end;
```

Ausschalten der Aktualisierung

Das Beispielprogramm führt das Erweitern der beiden Tabellen für den Benutzer sichtbar durch, das heißt, dass Sie in der Tabelle immer den gerade hinzugefügten Datensatz sehen könnten, wenn dieser nicht so schnell von immer neuen Datensätzen wieder außer Sichtweite geschoben werden würde.

Wie schon erwähnt, können Sie diese Aktualisierung im Programm über die Methoden *DisableControls* und *EnableControls* ein- und ausschalten. *FileDB* gibt diese Aufgabe an den Benutzer weiter und stellt ihm den Markierungsschalter *DisableControls* zur Verfügung, mit dem Sie die Aktualisierung beim laufenden Einlesevorgang beliebig ein- und ausschalten können:

```
procedure TMainForm.DisableControlsClick(Sender: TObject);
begin
  { Da DisableControls ein Markierungsschalter ist (und kein Menüpunkt),
    muss das Property Checked nicht von Hand gesetzt werden. }
  if DisableControls.Checked then begin
    Files.DisableControls;
    Dirs.DisableControls;
  end else begin
    Files.EnableControls;
    Dirs.EnableControls;
  end;
end;
```

Während des Einlesevorgangs zeigt das Programm die Zahl der hinzugefügten Dateien und Verzeichnisse an (Aufruf von *UpdateLabels* in der Methode *InsertDir*), daher können Sie den Geschwindigkeitsunterschied zwischen beiden Modi leicht erkennen.

7.4 Sortieren, Suchen und Filtern

Während das Kapitel 7.3 ganz im Zeichen der Felder stand, bringt dieses Kapitel als wesentliche Neuerung die Indizes, die dazu dienen, die elementaren Datenbankoperationen wie Suchen, Sortieren und Filtern zu beschleunigen. Da diese Indizes von dbExpress und Interbase weitgehend automatisch gehandhabt werden, können wir uns hier auf die Sortier-, Such- und Filterfunktionen konzentrieren und damit in Kapitel 7.4.4 die Funktionen der Beispielanwendung *FileDB* komplettieren.

ültigkeitsbereich der Methoden

Zuerst muss aber auf eine generelle Einschränkung der in diesem Kapitel erwähnten Such- und Filtermethoden hingewiesen werden: Sie beziehen sich nicht auf die Datenbank, wie sie vom Server verwaltet wird, sondern auf die Datenmenge, wie sie der Kylix-Anwendung erscheint. In dbExpress werden diese Methoden nur von den Client-Datenmengen wie *TClientDataSet*, *TSQLClientDataSet* unterstützt, die grundsätzlich die gesamte Datenmenge, auf die die genannten Methoden wirken, im Hauptspeicher halten.

Während in den Browser-Beispielprogrammen von Kapitel 7.2 und in der Dateidatenbank aus Kapitel 7.3.6 jeweils ganze Tabellen vollständig in eine Client-Datenmenge (also in den Hauptspeicher) geladen werden, enthalten Client-Datenmengen in der Praxis häufig nur einen Ausschnitt aus der Datenbanktabelle (Abbildung 7.12).

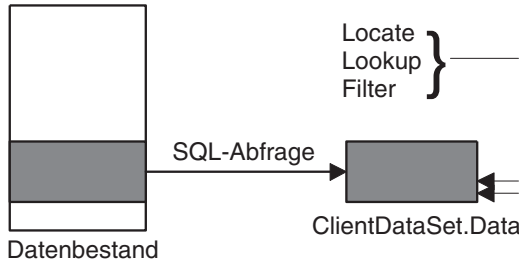


Abbildung 7.12: Wirkungsbereich der Such- und Filtermethoden in einem ClientDataSet

Dieser in einer Client-Datenmenge sichtbare Ausschnitt kann

- das Ergebnis einer expliziten Abfrage sein (in Kapitel 7.5.2 finden Sie Beispiele, bei denen zwar gleich Daten aus mehreren Tabellen kombiniert werden, im Prinzip handelt es sich aber auch dabei nur um einen Ausschnitt, und zwar um einen Ausschnitt des Gesamt-Datenbestandes),
- oder implizit beispielsweise durch die Definition einer *MasterSource* bestimmt werden (siehe Kapitel 7.4.5).

Bei allen in diesem Kapitel erwähnten Methoden zum Suchen und Filtern sollten Sie im Hinterkopf behalten, dass sich diese nur auf diesen in der Client-Datenmenge dargestellten Ausschnitt beziehen. Wenn Sie also etwa die gesamte Datenbanktabelle nach einem bestimmten Datensatz durchsuchen wollen, müssen Sie dies entweder mit einer SQL-Abfrage machen oder Sie müssen ein ClientDataSet verwenden, in dem die gesamte Tabelle geladen ist (für MyBase-Dateien: diese werden immer komplett in das ClientDataSet geladen; für SQL-Tabellen: setzen Sie *CommandType* auf *ctTable* und *CommandText* auf den Namen der Tabelle, um die gesamte Tabelle zu laden).

7.4.1 Methoden zum Suchen, Filtern und für Lesezeichen

Dieses Kapitel beschreibt einige Methoden, die bereits in der Klasse *TDataSet* definiert sind, die jedoch nicht in den neuen unidirektionalen Datenmengen von dbExpress verwendet werden dürfen. Der Aufruf dieser Methoden führt bei diesen Datenmengen entweder zu einer Exception (bei den meisten Funktionen) oder zu einem undefinierten Ergebnis (im Falle von *GetBookmark*). Die ClientDataSet-Familie jedoch unterstützt diese Methoden vollständig, wobei der Wirkungsbereich auf die im Speicher gehaltenen Daten beschränkt ist, wie oben erläutert.

Suchen mit Locate

Die Methode *Locate* ist eine sehr flexible Suchfunktion, die mit nur drei Parametern auskommt:

- Im ersten Parameter geben Sie die Namen der Felder an, in denen gesucht werden soll. Wenn Sie mehrere Felder angeben, trennen Sie diese durch Semikola.
- Im zweiten Parameter geben Sie die Werte an, nach denen gesucht werden soll. Dieser Parameter hat den Typ *Variant*. Das bedeutet, dass Sie, wenn Sie nur in einem Feld suchen, einen Wert eines beliebigen Typs angeben können, der zum Suchfeld passt (für ein numerisches Feld können Sie z. B. einen String angeben, der eine Zahl enthält). Wenn Sie in mehreren Feldern suchen, übergeben Sie diese in einem Varianten-Array (siehe Beispiel unten).
- Im dritten Parameter geben Sie eine Menge an, die zwei Optionen enthalten kann: Mit *loCaseSensitive* beachtet die Suchfunktion Groß- und Kleinschreibung in Strings, mit *loPartialKey* interpretiert sie die Suchwerte nur als Teilschlüssel, findet also alle Datensätze, die mit dem angegebenen Schlüssel beginnen.

Locate gibt *True* zurück, wenn sie einen Datensatz mit den angegebenen Kriterien finden konnte. In diesem Fall setzt sie die aktuelle Position auf diesen Datensatz, so dass Sie seine anderen Felder sofort auslesen können. Zwei Beispiele für *Locate*-Aufrufe sind:

```
Table.Locate('Nummer', 100, []);
Files.Locate('DirID;FileName', VarArrayOf([3, 'A']), [loPartialKey]);
```

Die erste Anweisung sucht im Feld *Nummer* nach dem konstanten Wert 100 (dieser Wert wird automatisch in eine Variante umgewandelt), die zweite Anweisung sucht einen Datensatz, dessen Feld *Vznr* den Wert 3 aufweist und dessen Feld *Dateiname* mit dem Buchstaben »A« beginnt (da *loPartialKey* gesetzt ist). Ein Praxisbeispiel zu *Locate* finden Sie in Kapitel 7.4.4.

Die Vorteile von *Locate* sind, dass sie sich selbst um die Aktivierung der notwendigen Indizes kümmert und dass sie auch funktioniert, wenn keine passenden Indizes verügbar sind (wenn sie dann auch nicht so schnell ist ...).

Lookup

Die Methode *Lookup* sucht ebenso flexibel in einer oder mehreren Spalten, allerdings verändert sie nicht die aktuelle Position der Tabelle, sondern liest nur Teile des gefundenen Datensatzes aus und gibt sie zurück. Welche Teile (Felder) des gefundenen Datensatzes Sie erhalten wollen, geben Sie im letzten Parameter an. *Lookup* liefert das Ergebnis in Form einer Varianten, bei mehreren Rückgabefeldern in einem Varianten-Array:

```
Pfad := Dirs.Lookup('DirID', GesuchteNummer, 'Path');
```

Dieser Aufruf sucht die *GesuchteNummer* in der Spalte *DirID* der Tabelle *Dirs* und gibt den Inhalt des Feldes *Path* aus dem gefundenen Datensatz zurück. *Lookup* führt damit prinzipiell genau die Funktion durch, die ein Lookup-Feld (siehe Kapitel 7.3.2) bei jeder Änderung des aktuellen Datensatzes durchführen muss. Allerdings haben Sie bei der Methode *Lookup* die Möglichkeit, mehrere Rückgabefelder anzugeben (siehe hierzu die Online-Hilfe).

Filtern

Beim Filtern einer Tabelle werden *mehrere* Datensätze gesucht, die bestimmten Kriterien genügen. Filterkriterien können in der Praxis erheblich komplexer sein als die oben besprochenen Suchkriterien (beispielsweise können sie »<=«- und »>=«-Bedingungen enthalten). Dieser Komplexität wäre selbst ein Variantenparameter kaum gewachsen, daher gibt es zum Filtern keinen einfach zu handhabenden Methodenaufruf wie für das Suchen. Stattdessen müssen Sie selbst eine Methode für das Ereignis *OnFilterRecord* schreiben und in dieser den aktuellen Datensatz der im Parameter erhaltenen Tabelle auf die Kriterien hin untersuchen. Die folgende Methode könnte beispielsweise alle Dateien herausfiltern, die größer als 200000 Byte sind:

```
procedure TForm1.Table1FilterRec(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['Groesse'] > 200000;
end;
```

Wenn sie eine solche Methode mit Ihrer *DataSet*-Komponente verknüpft haben, müssen Sie die Filterung noch einschalten, indem Sie das *DataSet*-Property *Filtered* auf *True* setzen. Dadurch werden so lange nur noch die Datensätze sichtbar und ansprechbar, die den Kriterien entsprechen, bis Sie *Filtered* wieder abschalten. Sie können also alle gefundenen Datensätze durchlaufen, indem Sie mit *DataSet.First* an den Anfang der Datenmenge gehen und dann mit der Methode *Next* jedes einzelne Element besuchen, bis *EOF=True*.

Lesezeichen

Ein Effekt der Methode *Lookup* ist, dass sie die im Formular angezeigte Position in der Tabelle ändert. Auch in anderen Situationen kann ein kurzzeitiger Wechsel zu einem anderen Datensatz erforderlich sein, nach dem der vorherige Datensatz wieder aktiviert werden soll.

Da Datensätze nicht über eine absolute Position angesprochen werden sollten, gibt es die Möglichkeit, die aktuelle Position mit Hilfe der Lesezeichen-Methoden wiederherzustellen. Sie können ein Lesezeichen immer nur an der aktuellen Position unterbringen, daher fällt der Aufruf der Methode *GetBookMark* recht kurz aus:

```
var
  BookMark: TBookMark;
begin
  BookMark := Tabelle.GetBookMark;
```

GetBookmark liefert eine Variable des Typs *TBookmark* zurück. Was es mit diesem auf sich hat, brauchen Sie nicht zu wissen, denn der einzige Zweck der *TBookmark*-Variable ist es, später an die Funktion *GoToBookmark* übergeben zu werden, um an die Position des Lesezeichens zurückzukehren:

```
VeraendereTabelle(Tabelle);
Tabelle.GotoBookmark(BookMark);
```

Da *GoToBookmark* das Lesezeichen nicht wieder auflöst, können Sie es beliebig oft verwenden, um später erneut an dieselbe Position zurückzukehren. Falls Sie das Lesezeichen aber nur temporär benötigen, sollten Sie den Speicher, den die CLX dafür reservieren musste, mit *FreeBookmark* wieder freigeben:

```
Tabelle.FreeBookmark;
```

7.4.2 Sortieren mit Indizes

Ein Index sortiert die Datensätze einer Tabelle nach einem oder mehreren Feldern in aufsteigender oder absteigender Reihenfolge. Der Index enthält nicht die kompletten Datensätze, sondern gibt nur an, an welcher Position der Tabelle sie zu finden sind und in welcher Reihenfolge sie innerhalb des Indexes angeordnet sind. Durch einen Index

ist es möglich, einen gewünschten Wert viel schneller zu finden als ohne Index, wobei der Vorteil des Indexes mit der Anzahl der Datensätze wächst (normalerweise überproportional stark). Während die Datenbank-Software einen gesuchten Feldwert in einem Index z. B. mit binärer Suche in wenigen Schritten finden kann, müsste sie ohne den Index jeden Datensatz einzeln vergleichen, was schon bei kleinen Tabellen um ein Vielfaches länger dauern kann.

Zusammengesetzte Indizes

Ein Index muss sich nicht nur nach einem einzelnen Feld richten, er kann sich auch aus mehreren Feldern zusammensetzen. Dabei werden die Datensätze zuerst nach dem ersten Indexfeld sortiert. Gibt es in dieser Spalte doppelte Werte, werden die Datensätze, die hier den gleichen Wert aufweisen, nach dem zweiten Indexfeld sortiert. Dieser Vorgang setzt sich bis zum letzten Indexfeld fort. Die übliche String-Schreibweise eines zusammengesetzten Indizes besteht darin, die einzelnen Felder hintereinander zu schreiben und durch Semikola zu trennen.

Wenn Sie beispielsweise den Index »Wohnort;Nachname;Name« in einer Adressdatenbank definieren, wird dadurch die gesamte Tabelle nach dem Wohnort sortiert. Bei dieser Sortierung lässt sich eine aufsteigende Reihenfolge der Nachnamen nur innerhalb desselben Orts finden. Wenn dann derselbe Nachname in einem Ort mehrmals vorkommt, werden selbst diese Datensätze noch nach Vornamen sortiert.

Ein Beispiel für die Erzeugung von einfachen und zusammengesetzten Indizes mit Interbase finden Sie im SQL-Skript von Kapitel 7.1.2.

Den Index auswählen

Die standardmäßige Sortierreihenfolge in einem SQL-ClientDataSet wird durch die Reihenfolge bestimmt, in der die Datensätze von dem Server geliefert werden. Auf dem Server kann die Reihenfolge beispielsweise durch einen voreingestellten Primärindex bestimmt werden, falls nicht explizit ein anderer Index gewählt wird, oder durch eine *Order-By*-Klausel in einer SQL-Abfrage. In einer von einem Server unabhängigen Client-Datenmenge ist es auch möglich, die Daten ohne jegliche Sortierung einzugeben und anzeigen zu lassen.

Um einen Index zu aktivieren, können Sie die *TClientDataSet*-Properties *IndexName* und *IndexFieldNames* verwenden:

- In *IndexName* nennen Sie den Index bei dem Namen, der ihm auf der Ebene des Datenbank-Servers zugewiesen wurde, oder Sie geben den Namen *DEFAULT_INDEX* an, der wieder auf die voreingestellte Sortierreihenfolge zurückschaltet (evtl. auch ohne Sortierung). Unabhängig von jeglichen auf einem Datenbank-Ser-

ver definierten Indizes können Sie über das Property *TClientDataSet.IndexDefs* auch eigene Indizes definieren, die dann alleine von der ClientDataSet-Komponente verwaltet werden.

- Alternativ dazu können Sie in *IndexFieldNames* die einzelnen Felder aufzählen, die im Index vorhanden sein sollen (wobei Sie die Felder mit Semikolons trennen). Hierbei dürfen Sie auch Felder angeben, für die auf Datenbank-Server-Ebene kein Index definiert ist, denn das *TClientDataSet* sortiert seine Daten sowieso selbst (ausgenommen die Standard-Sortierreihenfolge) und legt bei Bedarf einfach einen neuen Index an.

Hintergrund: Die Verpflichtung des ClientDataSets zur eigenständigen Sortierung folgt aus dem in der Einleitung von Kapitel 7.4 erwähnten Prinzip, dass Client-Datenmengen hauptsächlich mit den von ihnen im Speicher gehaltenen Daten arbeiten. Selbst wenn der Datenbank-Server einen passenden Index anzubieten hätte, kann das ClientDataSet nicht von ihm verlangen, sein Datenpaket zu sortieren, das sich ja im Hauptspeicher eines möglicherweise ganz anderen Rechners als dem des Servers befindet.

Wenn Sie eines der beiden Properties *IndexName* und *IndexFieldNames* setzen, wird jeweils das andere gelöscht. Um auch dann die Felder des aktuellen Indexes zu erfahren, wenn Sie *IndexName* gesetzt haben, können Sie die *DataSet*-Properties *IndexFields* und *IndexFieldCount* verwenden. Eine Übersicht über alle definierten Indizes gibt das Property *IndexDefs*. In der Handhabung sind diese drei Properties den Properties *Fields*, *FieldCount* und *FieldDefs* sehr ähnlich.

7.4.3 Modusabhängige Filter- und Suchfunktionen

Die Client-Datenmengen stellen noch einige alternative Methoden zum Suchen und Filtern der Daten zur Verfügung, die sich – im Gegensatz zu den in Kapitel 7.4.1 besprochenen – noch auf die *TTable*-Komponente der ersten Delphi-Version zurückführen lassen und bei denen die Datenmengen-Komponente in spezielle Modi geschaltet wird, die am Ende dieses Kapitels in einer Tabelle zusammengefasst werden. Dieses Kapitel beachtet von diesen Methoden hauptsächlich die zum Filtern vorgesehenen, denn diese sind zwar weniger flexibel als das *OnFilterRecord*-Ereignis, aber einfacher zu handhaben.

Die Klasse *TTable* stellt ebenfalls einige Methoden zum Suchen und Filtern der Daten zur Verfügung. Während die Such-Methoden nicht mehr bieten als die in Kapitel 7.4.1 besprochenen, von *TDataSet* geerbten Methoden (außer, dass sie kompatibel mit Delphi 1 sind), sind die *TTable*-Methoden zum Filtern zwar weniger flexibel, aber einfacher zu handhaben als die Filterungsmöglichkeiten der Klasse *TDataSet*.

Nach Bereichen filtern

Die Methoden zum Filtern von Bereichen geben Ihnen eine einfache Möglichkeit, die Datensätze einer Datenmenge abzufragen, die einfache Bereichskriterien erfüllen. Für jede Tabellenspalte können Sie einen eigenen Bereich definieren, in dem die Feldwerte der Datensätze liegen müssen. Wenn Sie für mehrere Spalten Bereiche definieren, müssen alle Einzel-Bereichskriterien erfüllt sein, damit ein Datensatz ins Filterergebnis aufgenommen wird.

Bei der Verwendung der Methode *SetRange* müssen Sie (im Gegensatz etwa zu den in Kapitel 7.4.1 genannten Methoden) selbst einen Index setzen, denn nach dessen Spalten entscheidet sich erst, wo überhaupt gefiltert werden soll. Mit *SetRange* können Sie alle Bereiche auf einmal setzen. Die Methode erwartet sowohl für den Bereichsbeginn als auch für das Ende einen Array-Parameter, innerhalb dessen Sie die einzelnen Werte in der Reihenfolge des Felder des aktuellen Indexes angeben. Wenn beispielsweise zwei numerische Felder indiziert sind, beschränken Sie die Anzeige der Datensätze mit der folgenden Anweisung auf die Datensätze, die in beiden Feldern einen Wert zwischen 0 und 2048 haben:

```
// Der Index muss vorher eingestellt werden, z.B.:
// Table.IndexFieldNames := 'NumField1;NumField2';
Table.SetRange([0, 0], [2048, 2048]);
```

Dasselbe Ergebnis wie mit diesem einzelnen Aufruf können Sie wie folgt auch in kleineren Schritten erreichen. Sie müssen sich dann nicht um die Reihenfolge der Werte im Array kümmern.

Als Erstes rufen Sie die Methode *SetRangeStart* auf, die die Tabelle in einen speziellen Modus setzt, in dem das Beschreiben der Feldwerte nicht den aktuellen Datensatz verändert, sondern den Beginn des Bereichs definiert. Sie können die Felder dann mit den Werten beschreiben, die die jeweilige Spalte mindestens haben muss, um in den Ergebnisbereich aufgenommen zu werden:

```
Files.SetRangeStart;
FilesGroesse.AsInteger := 0; { Bereichsbeginn im Feld FilesGroesse: 0 }
```

(Dieses Beispiel nimmt an, dass Sie mit dem Felder-Editor eine persistente *TField*-Komponente namens *FilesGroesse* für eine numerische Spalte der Tabelle angelegt haben.)

Danach rufen Sie *SetRangeEnd* auf und beschreiben die Feldwerte mit dem Ende des Bereichs. Zum Abschluss aktivieren Sie den Bereich mit einem Aufruf von *ApplyRange*:

```
Files.SetRangeEnd;
FilesGroesse.AsInteger := 2048;{ Bereichsende im Feld FilesGroesse: 2048 }
ApplyRange;
```

Um die Beschränkung irgendwann wieder aufzuheben, wenden Sie sich an die Methode *CancelRange*. Sie können außerdem mit dem Property *KeyExclusive* steuern,

ob die Bereichsgrenzen zum Filterergebnis dazugehören, was per Voreinstellung der Fall ist (*KeyExclusive=False*). Außerdem können Sie das Bereichsende auch offen lassen, beispielsweise wenn Sie alle Dateien finden wollen, deren Name mit einer bestimmten Zeichenkette beginnt:

```
Files.SetRangeStart;
FilesDateiname.AsString := 'VERY';
Files.SetRangeEnd; { Alle Strings finden, die mit VERY beginnen }
Files.ApplyRange;
```

Suchen

Da 7.4.1 bereits die leistungsfähige *Locate*-Funktion besprochen hat, sollen die alternativen Suchfunktionen von *TClientDataSet* hier nur am Rande erwähnt werden. Sie unterscheiden sich von *Locate* dadurch, dass die Angabe dessen, was gesucht werden soll, nicht in einem, sondern in mehreren Schritten erfolgt. Je nach Situation kann eine solche ausführliche Schreibweise übersichtlicher, umständlicher oder einfach eine Geschmacksfrage sein.

Zuerst geben Sie die Spalten an, in denen gesucht wird, indem Sie eine passende Indizierung in *IndexFieldNames* oder *IndexName* wählen, dann folgt eine Sequenz, die dem Filtern mit *SetRangeStart/End* und *ApplyRange* aus dem vorherigen Abschnitt stark ähnelt: Statt der Bereichsgrenzen schreiben Sie die zu suchenden Daten in die Felder, zur Einleitung rufen Sie die Methode *SetKey*, zum Abschluss *GotoKey* auf. Wenn Sie nach Teilschlüsseln suchen wollen, verwenden Sie *GotoNearest* statt *GotoKey*.

Hinweis: Die Verwendung von *SetKey/GotoKey* kann sogar effizienter sein als der Aufruf von *Locate*, nämlich wenn mehrere geringfügig unterschiedliche Suchen aufeinander folgen. Während Sie bei *Locate* immer die gesamte Parameterliste angeben müssen – von den Suchspalten bis zu den Werten für jede einzelne Spalte – brauchen Sie vor einem erneuten Aufruf von *GotoKey* nur die Spalten zu ändern, deren Suchwerte sich verändert haben.

TDataSet-Zustände

Die Zustände einer *TDataSet*-Komponente dienen unter anderem dazu, bei den Zugriffen auf die aktuellen Feldwerte zu unterscheiden, ob dieser Zugriff die Daten des Datensatzes verändert oder ob er zur Angabe einer Bereichsgrenze oder von Suchparametern dient. Nachdem wir die Tabelle in den letzten Abschnitten in den *dsSetKey*-Modus geschaltet haben, fasst die folgende Tabelle die wichtigsten Zustände einer *DataSet*-Komponente zusammen.

Zustand	Beschreibung	herbeigeführt durch	beendet durch
dsInactive	Die Tabelle ist noch geschlossen.	Close	Open
dsBrowse	Anfangsmodus nach der Aktivierung, kein Editieren möglich.	Beenden jedes anderen Zustandes	Wechsel in einen beliebigen anderen Zustand
dsEdit	Normaler Editiermodus: Eine vorhandene Zeile wird editiert. Änderungen der Daten der Feldkomponenten werden bei *Post*-Aufruf in die Datenbank übernommen.	Edit	Cancel, Delete, Post
dsInsert	Eine neue, noch nicht gespeicherte Zeile wird editiert, sonst wie *dsEdit*.	Insert, Append	Post, Delete
dsSetKey	Suchparameter werden spezifiziert, nur bei *TTable* gültig.	SetKey, EditKey, SetRangeStart, SetRangeEnd	Cancel, GotoKey, FindKey, ApplyRange
dsCalcFields	Nur berechnete Felder können verändert werden (während der Bearbeitung des *OnCalcField*-Events).	automatisches Ereignis	Ende der *OnCalcField*-Methode
dsFilter	Eine mit dem *OnFilterRecord*-Ereignis verknüpfte Methode prüft gerade das Filterkriterium.	automatisches Ereignis, falls *Filtered=True*	Ende der *OnFilterRecord*-Methode

Sie können den aktuellen Zustand jederzeit über das Property *TDataSet.State* abfragen, allerdings nicht über dieses Property ändern. Hierzu müssen Sie die in der Tabelle gezeigten Methoden aufrufen.

7.4.4 Aktualisieren der Dateidatenbank

In diesem Kapitel geht es darum, die Beschränkungen der Dateidatenbank aus Kapitel 7.3.6 zu beseitigen: Da diese die Datensätze immer nur hinzufügt, aber niemals bestehende ersetzt, kann es schnell zu doppelten Einträgen, folglich zu doppelten Schlüsseln und somit zu einer Indexfehler-Exception kommen, sobald die Anwendung versucht, die geschriebenen Datensätze mit *Post* in das Änderungsprotokoll zu übernehmen.

Die Indizes der beiden Tabellen

Im SQL-Skript zur Erzeugung der Dateidatenbank (siehe Kapitel 7.1.2) werden neben den Tabellen auch einige Indizes definiert, zwei davon mit dem Schlüsselwort *Unique*, welches besagt, dass es sich um einen eindeutigen Index handelt, dass also alle Schlüssel verschieden sind:

```
CREATE UNIQUE INDEX DIRECTORIESO ON Directories(DirID);
CREATE UNIQUE INDEX FILESO ON Files(DirID, FileName);
```

Der erste Index, *Directories0*, indiziert das Feld *DirID*, für das das Beispielprogramm sicherstellt, dass jedes Verzeichnis eine eindeutige Nummer erhält. In der Tabelle *Files* würde es der Index jedoch nicht eindeutig sein, wenn er nur das Feld *DirID* berücksichtigen würde, da die gleiche Verzeichnisnummer in der Dateitabelle mehrmals vorkommt (falls der Benutzer es wagt, das Programm auf Verzeichnisse anzuwenden, die mehr als eine Datei enthalten).

Anmerkung: Die Indexnamen *Directories0* und *Files0* kamen bei der Konvertierung alter Paradox-Tabellen mit dem Data Pump-Utility zustande. In den Paradox-Tabellen waren die beiden Indizes als Primärindex definiert, und da diese Primärindizes unter Paradox unbenannt sind, hat sich Data Pump die oben erwähnten Namen ausgedacht. Da die Konvertierung nicht erfolgreich abgeschlossen werden konnte, sollen die vergeblichen Bemühungen des Data Pump-Utilities heute wenigstens dadurch geehrt werden, dass sein Benennungsvorschlag für die »Primärindizes« akzeptiert wird.

Diese beiden eindeutigen Indizes sind jedoch für unser Beispielprogramm noch nicht ausreichend. Betrachten wir die Frage, ob ein Verzeichnis schon vorhanden ist. Die Verzeichnistabelle ist im Index *Dirs0* nach der Verzeichnisnummer sortiert, muss aber in diesem Fall nach dem Pfadnamen durchsucht werden. Hierfür kann die Tabelle einen eigenen Index gut gebrauchen, er wird im SQL-Skript wie folgt definiert:

```
CREATE INDEX Path ON Directories(Path);
```

Hinweis: Im ClientDataSet des Beispielprogramms wird dieser Index auf jeden Fall benötigt, aber in der Interbase-Datenbank hätte er auch weggelassen werden können, denn da das Beispielprogramm immer die gesamte Verzeichnistabelle in das ClientDataSet *Dirs* lädt, findet die Suche nach einem Verzeichnis auch immer ausschließlich im ClientDataSet-Datenpaket statt – ganz ohne Beteiligung von Interbase.

Die folgende Tabelle zeigt alle in *FileDB* verwendeten Indizes, von denen einige erst im nächsten Kapitel eingesetzt werden:

Tabelle	Indexname	Indizierte Felder
Dirs	Directories0 (»Primärindex«)	DirID
Files	Files0 (»Primärindex«)	DirID;FileName
Dirs	Path	Path
Files	ByDate	DirID;FileDate
Files	BySize	DirID;FileSize

Der letzte Hinweis gilt analog für alle Indizes der Beispieltabelle: Da das Programm seine Such- und Sortierfunktionen ohne SQL, sondern alleine in den ClientDataSets durchführt, hätte die Definition der Indizes auf Interbase-Ebene (im SQL-Skript von Kapitel 7.1.2) auch weggelassen werden können.

Suchen, um zu aktualisieren

Die neue Version von *StoreFile* sucht nun, bevor sie die in den Parametern erhaltenen Dateidaten in einen Datensatz schreibt, in den Spalten *DirID* und *FileName*, ob bereits eine Datei des gleichen Namens und mit gleicher Verzeichnisnummer in der Datenbank enthalten ist. Das Ergebnis der Suchfunktion *Locate* entscheidet dann darüber, ob mit *Insert* ein neuer Eintrag eingefügt wird oder der gefundene Eintrag mit *Edit* angepasst wird (schließlich könnten sich die Größe und das Datum einer schon früher eingelesenen Datei geändert haben):

```
procedure TMainForm.StoreFile(Dir: LongInt; Name: string;
                              Size: LongInt; Date: TDateTime);
begin
  Application.ProcessMessages; { "Multitasking", siehe Kapitel 7.3.7 }
  if not Files.Locate('DirID;FileName',
                      VarArrayOf([Dir, Name]), [])
  then
    Files.Insert
  else begin
    inc(FileUpdates); { für die Statusanzeige }
    Files.Edit;
  end;
  { Das Eintragen der Daten ist für Insert und Edit gleich: }
  FilesFileTime.AsDateTime := Date;
  FilesFileName.AsString := Name;
  FilesDirID.AsInteger := Dir;
  FilesFileSize.AsInteger := Size;
  Files.Post; { Änderungen schreiben }
end;
```

Eindeutige Verzeichnisnummerierung

FileDB verwendet eine sehr einfache Methode, jedes Verzeichnis mit einer eindeutigen Nummer zu versehen: Jedes neue Verzeichnis erhält eine Nummer, die um eins größer ist als die bisher maximale Nummer. Falls einmal Verzeichnisse gelöscht werden (eine in diesem Beispielprogramm nicht implementierte Funktion), füllt *FileDB* die Lücken in der Nummerierung nicht auf.

Nach dem Öffnen der Tabelle stellt *FormCreate* die bisher größte Verzeichnisnummer fest, indem sie mit der Methode *TTable.Last* an das Ende der Tabelle springt (die nach ihrem Primärindex nach der Verzeichnisnummer sortiert ist) und die Nummer des dort befindlichen Eintrags ausliest:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Files.Active := True;
  Dirs.Active := True;
  Dirs.Last;
  MaxDirIndex := DirsDirID.AsInteger;
  UpdateLabels;
end;
```

Die Methode *StoreDir* ist geringfügig anders aufgebaut, da sie keine Daten aktualisieren muss, wenn das Verzeichnis schon vorhanden ist:

```
function TMainForm.StoreDir(Path: string): LongInt;
begin
  if not Dirs.Locate('Path', Path, []) then begin
    Dirs.Insert;
    inc(MaxDirIndex);
    Result := MaxDirIndex;
    DirsDirID.AsInteger := MaxDirIndex;
    DirsPath.AsString := Path;
    Dirs.Post;
  end else begin
    inc(DirUpdates);
    Result := DirsDirID.AsInteger;
  end;
end;
```

Neben dem Speichern der Datensätze wird hier die zweite Aufgabe von *StoreDir* besonders wichtig: die Rückgabe der Verzeichnisnummer an die Methode *InsertDir*, die diese Nummer an *StoreFile* weitergibt (zu *InsertDir* siehe Kapitel 7.3.6).

7.4.5 Haupt-/Detailformulare und Sortieren

Das Formular des Beispielprogramms *FileDB* (Abbildung 7.11) ist ein typisches Beispiel für ein Haupt-/Detail- (bzw. Master-/Detail-)Formular: Eine Haupttabelle gibt eine grobe Übersicht über alle zur Verfügung stehenden Daten – in diesem Fall über alle eingelesenen Verzeichnisse, während die Detailtabelle genauere Angaben zu dem Element gibt, das gerade in der Haupttabelle ausgewählt ist – in diesem Fall also alle in diesem Verzeichnis gefundenen Dateien.

Properties zur Verknüpfung *R126*

Sie können eine Haupt-/Detailkombination von Hand erstellen, indem Sie zwei Datenmengen, zwei *DataSource*-Komponenten und zwei *DBGrids* wie gewohnt in das Formular aufnehmen und dann zwei Properties der Detailtabelle verändern:

- In *MasterSource* wählen Sie die *TDataSource*-Komponente der Haupttabelle, nach der sich die Detailtabelle richten soll.

- In *MasterFields* geben Sie die Felder der Haupttabelle an, die die Auswahl der Details der Detailtabelle steuern sollen. Der Inhalt dieser Felder wird an den Index der Detailtabelle geleitet, und diese zeigt dann nur die Datensätze, die in den Indexfeldern mit den *MasterFields* der Haupttabelle übereinstimmen.

ür *FileDB* bedeutet das: Die *MasterSource* von *Files* ist *DirSource*, die mit der Verzeichnistabelle verbundene *TDataSource*-Komponente. Die *MasterFields* enthalten nur ein Feld, und zwar das Feld *DirID*.

Die Verknüpfung der beiden Tabellen wirkt sich wie folgt aus: Jedes Mal, wenn Sie in der Verzeichnisliste einen anderen Datensatz auswählen, wird die Nummer dieses Verzeichnisses im Index der Dateitabelle gesucht und nur die Dateien mit derselben Nummer werden aufgelistet.

Hinweis: Für das Einfügen neuer Datensätze beim Einlesen eines Verzeichnisses muss die Master-/Detail-Verknüpfung von FileDB wieder gelöst werden, damit per *Locate* alle Dateien der Dateitabelle gefunden werden können (sonst würden nur die Dateien des Verzeichnisses gesucht, das gerade in der Master-Tabelle gewählt ist). Zum Lösen der Verknüpfung muss *MasterSource* auf *nil* geschaltet werden. Dies wurde bereits in Kapitel 7.3.7 in der Methode *AddFromDir* gezeigt. Jedoch scheint die CLX in der ersten Version auch hier mit einem Fehler behaftet zu sein, denn um die *Files*-Tabelle vollständig aus ihrer Verbindung mit der Master-Tabelle »zu befreien«, sind einige weitere sehr umständliche, teilweise abstruse Aktionen notwendig:

```
OldCommandText := Files.CommandText;
Files.Close;
Files.MasterSource := nil; // Minimalanforderung zum Lösen der Verbindung
Files.MasterFields := '';
Files.CommandText := ''; // Erneuerung der Datenmenge erzwingen
Files.PacketRecords := -1; // sollte autom. von der CLX gesetzt werden
Files.CommandText := OldCommandText;
Files.Open;
```

Der Autor konnte nicht alle möglichen Reihenfolgen der obigen Anweisungen ausprobieren, jedoch unter Voraussetzung der gezeigten Reihenfolge feststellen, dass *alle* obigen Anweisungen erforderlich sind, damit *Files* nach dem abschließenden *Open* wie gewünscht funktioniert. Wie angedeutet kann sich diese Situation allerdings in zukünftigen Versionen der CLX schnell bessern.

Sortieren der Details *R130*

Zum Abschluss erhält das *DBGrid* der Dateiliste noch ein kleines Popup-Menü, in dem Sie wählen können, ob die Dateien nach Größe, Datum oder nach dem Namen sortiert werden sollen. Die Sortierung kann prinzipiell einfach dadurch erreicht werden, dass die Spalten, nach denen sortiert werden soll, in das Property *IndexFieldNames* des

ClientDataSets geschrieben werden. In diesem Fall muss aber zusätzlich bedacht werden, dass *Files* eine Detail-Tabelle in einem Master-/Detailgespann ist und über die Verzeichnisnummer mit der Verzeichnistabelle verknüpft ist. Und die *MasterFields* einer Detail-Datenmenge müssen immer die ersten Bestandteile des aktuellen Indexes sein, damit diese Verknüpfung funktioniert.

Um sowohl der Steuerung durch die Master-Tabelle als auch dem Sortierungswunsch des Benutzers gerecht werden zu können, benötigt *FileDB* also die beiden Indizes *DirID;FileSize* und *DirID;FileTime* (eine Tabelle in Kapitel 7.4.4 zeigt alle von *FileDB* verwendeten Indizes). Mit diesen Indizes können die drei lokalen Menüpunkte schnell implementiert werden:

```
procedure TMainForm.NameSortClick(Sender: TObject);
begin
  Files.IndexFieldNames := 'DirID;FileName';
end;

procedure TMainForm.SizeSortClick(Sender: TObject);
begin
  Files.IndexFieldNames := 'DirID;FileSize';
end;

procedure TMainForm.DateSortClick(Sender: TObject);
begin
  Files.IndexFieldNames := 'DirID;FileTime';
end;
```

7.5 Eine Beispielanwendung

Nachdem die Beispiele der vorangegangenen Kapitel ihre speziellen Demonstrationsaufgaben fernab der Alltagspraxis ausgeführt haben, zeigt dieses Kapitel zum Abschluss ein praxisnähere Anwendung: Eine Terminverwaltung, die auf einer Datenbank mit drei Tabellen basiert, in denen Termine sowie damit verknüpfte Adressen und Aktionen abgelegt werden, soll vom Benutzer auf verschiedene Weise bearbeitet und angesehen werden können (siehe Abb. 7.15-7.17 in Kapitel 7.5.3).

Die eigentliche Funktion der Anwendung soll dabei offen bleiben – so enthält die Aktionstabelle etwa ein Feld, in dem sich ähnlich wie beim Beispielprogramm aus Kapitel 1.8.5 Programme eintragen lassen, die aber von dieser Anwendung nicht gestartet werden. Die Termintabelle enthält Felder, die eine Wiederholung des Termins im Tages-, Monats- oder Jahresrhythmus zulassen, doch die Anwendung erinnert nicht an diese Wiederholung. Stattdessen sollen in diesem Kapitel einige Funktionen implementiert werden, die uns Entwicklern Experimente mit der Funktionsweise von dbExpress

erlauben: einen SQL-Monitor, eine Simulation von Mehrbenutzerzugriff auf die Datenbank und ein Formular zur Ansicht des Änderungsprotokolls (in einer modifizierten Form könnte ein solches Formular auch für normale Benutzer interessant sein).

7.5.1 Definition der Datenbank

Da die Anwendung auch die Vorzüge von SQL-Abfragen nutzen soll, kommt eine Definition der Datenbank über MyBase-Dateien nicht in Betracht. Auf der CD finden Sie sowohl eine fertige Datenbankdatei für Interbase als auch das SQL-Skript zur Erzeugung der Datei. Mit diesem Skript sollte es möglich sein, die Datenbank auch mit anderen SQL-Servern zu verwenden, wie etwa MySQL, für das Kylix ja auch bereits einen Treiber mitliefert. Das Beispielprogramm ist auf der CD unter dem Namen *TerminDB* zu finden.

Metadaten-Extraktion

Das in diesem Kapitel gezeigte Skript wurde nicht per Hand angefertigt, da die ursprüngliche Interbase-Datei von dem in Kapitel 7.1.3 beschriebenen Data Pump-Utility von Delphi 5 aus drei Paradox-Tabellen erzeugt wurde, sondern mit dem *isql*-Utility von Interbase (nach der Interbase-Installation beispielsweise zu finden in `/opt/interbase/bin`). Auf eine bestehende Interbase-Datei angewendet, kann *isql* die Metadaten dieser Datei extrahieren und in Form von DDL-Anweisungen ausgeben. Aus diesen Anweisungen lässt sich die Struktur der Datenbank genau reproduzieren, so dass die gesamte Ausgabe von *isql* sich wieder als Skript zum Erzeugen einer neuen Datenbank verwenden lässt. Der Aufruf der Extraktionsfunktion von *isql* geschieht wie folgt:

```
isql -extract TerminDB.gdb
```

Dadurch werden die DDL-Anweisungen auf der Konsole ausgegeben. Eine Umleitung in die Datei ist leicht mit der entsprechenden Shell-Syntax oder mit der *isql*-Option »-output Dateiname« möglich. Zu beachten ist dabei, dass diese Option *zwischen* die *extract*-Option und den Namen der Datenbank gehört:

```
isql -extract -output MyScript.sql TerminDB.gdb
```

Hinweis: Standardmäßig gibt *isql* alle DDL-Anweisungen und Namen von Tabellen, Feldern, Generatoren etc. in Großbuchstaben aus. SQL-Anweisungen und -Bezeichner sind wie Object-Pascal-Code jedoch nicht abhängig von Groß- und Kleinschreibung. Daher werden die Listings von *isql* hier zwar originalgetreu wiedergegeben, im Folgenden aber für die selbst definierten Bezeichner auch Kleinbuchstaben verwendet.

Die Tabellen der Termindatenbank

Die Erzeugung einer Datenbank mit *isql* wurde bereits in Kapitel 7.1.2 erläutert, so dass im Folgenden nur noch die DDL-Anweisungen gezeigt werden, mit denen die im Beispielprogramm verwendeten Tabellen erstellt werden können:

```
CREATE DATABASE "TERMINE.GDB" PAGE_SIZE 1024;

CREATE TABLE TERMINE (TERMINNR INTEGER,
        DATUM DATE,
        ZEIT DATE,
        BESCHREIBUNG VARCHAR(80),
        AKTIONNR INTEGER,
        ADRESSNR INTEGER,
        WDHOLENT INTEGER,
        WDHOLENM INTEGER,
        WDHOLENJ INTEGER);
```

Diese Tabelle versieht jeden Termin mit einer Nummer. Die »Kerndaten« eines Termins bestehen aus der *Beschreibung* und einer Zeitangabe, die in *Datum* und *Zeit* aufgeteilt wurde.

Hinweis: Effizienter wäre es, Datum und Zeit in einem gemeinsamen Feld zu speichern, denn der *Date*-Datentyp von Interbase enthält auch Zeitinformationen. Vor der Konvertierung hatten die beiden Felder in der Paradox-Tabelle noch spezielle Typen für Datum und Zeit. Für dieses Beispielprogramm wurde es bei den beiden Feldern belassen, weil dadurch eine einfache Möglichkeit für den Benutzer entsteht, entweder Datum oder Zeit offen zu lassen (das Feld bei der SQL-Konstanten *null* zu belassen).

Die drei letzten Felder *WdHolenT/M/J* sind für sich wiederholende Termine gedacht und werden in der Programmversion auf der CD nur dadurch beachtet, dass alle wiederholten Termine in einer Tabelle des Hauptfensters aufgelistet werden (jedoch rechnet das Programm nicht aus, an welchen Tagen nun die Wiederholungen eines bestimmten Termins stattfinden).

Wichtig sind die beiden Felder *AktionNr* und *AdressNr*, die den Termin jeweils mit einer Aktion und einer Adresse aus den anderen beiden Tabellen verknüpfen:

```
CREATE TABLE ADRESSEN (ADRESSNR INTEGER,
        VORNAME VARCHAR(40),
        NACHNAME VARCHAR(40),
        POSTANSCHRIFT VARCHAR(100),
        EMAIL VARCHAR(100),
        NOTIZEN VARCHAR(100));

CREATE TABLE AKTIONEN (AKTIONNR INTEGER,
```

```
    AUSGABEOPTIONEN INTEGER,
    SIGNALZAHL INTEGER,
    KOMMANDO VARCHAR(180));
```

ür das Beispielprogramm sind die wichtigsten Felder dieser beiden Tabellen jeweils die erstgenannten, die die Adress- und Aktionsnummer angeben. Alle anderen Felder gehören lediglich zur »Kulisse« des Programms.

Hinweis: Was es bedeutet, dass ein Termin mit einer Adresse bzw. einer Aktion verknüpft ist, bleibt in diesem Beispielprogramm alleine dem Benutzer überlassen. Wie schon in der Einleitung erwähnt, führt das Programm selbst keine Interpretationen aus und beinhaltet auch keine automatische Ausführung irgendwelcher Aktionen.

Automatische Nummerierung von Datensätzen *R128*

Jede der drei Beispieltabellen verfügt über eine Spalte, durch die jeder Datensatz eine Nummer zugeordnet bekommt. Dass diese Nummer eindeutig sein sollte, versteht sich von selbst, da die Nummer in anderen Tabellen als Referenz auf den jeweiligen Datensatz verwendet werden soll. Eine Möglichkeit, diese Eindeutigkeit von der Datenbanksoftware sicherstellen zu lassen, besteht darin, auf jede Nummernspalte einen eindeutigen Index zu definieren, also einen Index, in dem jeder Wert nur ein einziges Mal vorkommen darf. Einen solchen erhält man in SQL mit dem Schlüsselwort *UNIQUE*:

```
/*  Index definitions for all user tables */
CREATE UNIQUE INDEX ADRESSENO ON ADRESSEN(ADRESSNR);
CREATE UNIQUE INDEX AKTIONENO ON AKTIONEN(AKTIONNR);
CREATE UNIQUE INDEX TERMINEO ON TERMINE(TERMINNR);
```

Jeder Versuch, einen neuen Datensatz mit einer schon bestehenden Nummer zu versehen, wird von Interbase mit einem SQL-Fehler geahndet, der seine Laufbahn in einer Kylix-Anwendung als Exception fortsetzt. Die drei Indizes dienen in diesem Beispielprogramm hauptsächlich dieser Absicherung der Eindeutigkeit und werden ansonsten nicht explizit verwendet, da das Programm auf Sortieroptionen verzichtet.

Interbase verfügt über ein sehr robustes Konzept für die automatische Zuweisung eindeutiger Schlüssel zu neuen Datensätzen, mit dessen Hilfe es der Anwendung leicht gemacht wird, die Eindeutigkeitsregel für die Nummern zu beachten. Dieses Konzept setzt sich zusammen aus

- Generatoren – das sind Variablen, deren Werte von Interbase automatisch bei jedem Abruf in festgelegten Schritten erhöht werden,
- Triggern – das sind Anweisungen, die als Reaktion auf bestimmte Ereignisse auf dem Server selbst ausgeführt werden.

Das Ereignis, das bei der automatischen Nummerierung entscheidend ist, ist das der Erzeugung eines neuen Datensatzes; in SQL heißt dieses Ereignis *INSERT* und Sie können eigenen Code sowohl direkt vor diesem Ereignis als auch danach ausführen lassen (hierfür geben Sie vor *INSERT* noch den Zusatz *BEFORE* oder *AFTER* an).

Um beispielsweise das Feld *TerminNr* in der Termintabelle bei jeder Einfügung eines neuen Datensatzes mit einem neuen Wert des Generators zu belegen, wird folgender Trigger definiert:

```
SET TERM ^ ;
CREATE TRIGGER CREATE_TERMINNR FOR TERMINE
ACTIVE BEFORE INSERT POSITION 0
AS BEGIN
    NEW.TERMINNR = GEN_ID(TerminNr_Gen, 1);
END ^
SET TERM ; ^
```

Sie können auf diese Weise auch *mehrere* Ereignis-Handler mit demselben Ereignis verknüpfen. Die Angabe von *POSITION 0* besagt, an welcher Stelle dieser Handler in einer Kette von mehreren Handlern ausgeführt werden soll.

Wer die Arbeit mit einem »intelligenten« Compiler wie dem Object-Pascal-Compiler von Kylix gewohnt ist, wird sich bei der ersten Begegnung mit einem solchen Skript über eine weitere Besonderheit wundern: Die gesamte Trigger-Definition soll von SQL als eine Anweisung verstanden werden und muss damit mit einem »Terminator«-Zeichen beendet werden, welches standardmäßig das Semikolon ist. Dieses Semikolon wird jedoch auch zum Abschluss des geschachtelten Codes, der beim *INSERT*-Ereignis ausgeführt werden soll, benötigt. Damit der ahnungslose SQL-Interpreter dieses geschachtelte Semikolon nicht als Ende der *CREATE-TRIGGER*-Anweisung missversteht, muss vor dieser SQL-Anweisung mit *SET TERM* ein anderes Terminatorzeichen eingestellt werden. Diese Einstellung wird am Schluss wieder rückgängig gemacht.

Der im obigen Skript genannte Generator muss schon vorher definiert worden sein, in der Beispieldatenbank geschah dies wie folgt:

```
CREATE GENERATOR TERMINNR_GEN;
```

Ausführliche Informationen zu *CREATE TRIGGER* und *CREATE GENERATOR* erhalten Sie beispielsweise in der mit Interbase mitgelieferten Dokumentation *langref.pdf*.

Verwendung des Triggers in der Praxis

Die Verwendung des Triggers sieht nun so aus, dass bei der Eingabe eines neuen Datensatzes die Nummernspalte einfach offen bleiben kann. Wenn der Datensatz aus dem Änderungsprotokoll mit *ApplyUpdates* an den Server geschickt wird, versieht dieser den Datensatz mit einer neuen Nummer. Damit der Benutzer nicht versehentlich trotzdem eine Nummer eingibt, die dann vom Server überschrieben wird, sollte das

Nummernfeld sich in der Anwendung nicht editieren lassen. Damit die vom Server erzeugte Nummer später auch im Programm angezeigt wird, ist ein *Refresh* der Datenmenge notwendig (siehe *TdmTermine.cdsGeneralAfterPost* Kapitel 7.5.4).

7.5.2 Das Datenmodul

Im Gegensatz zu den bisherigen Beispielprogrammen drängt es sich für die Terminverwaltung geradezu auf, alle Datenzugriffskomponenten in einem eigenen Datenmodul unterzubringen, denn die Komponenten sollen von vielen verschiedenen Formularen aus angesprochen werden.

Abbildung 7.13 zeigt das Datenmodul, wie es von der Kylix-IDE präsentiert wird.

- Zunächst wird eine Verbindungskomponente (*TSQLConnection*) verwendet, über die die Verbindung zu Interbase hergestellt wird (siehe Kapitel 7.1.4). Das Property *LoginPrompt* wurde auf *False* eingestellt, weil die Datenbank sowieso nur das allgemein bekannte Passwort *masterkey* verwendet. Alle im Folgenden genannten Datenmengen sind über ihr *DBConnection*-Property mit dieser Verbindungskomponente verknüpft.
- Für jede der drei Tabellen wurde nun jeweils ein Exemplar von *TSQLClientDataSet* und eines von *TDataSource* in das Modul eingefügt. Der *CommandType* der Datenmengen wurde auf *ctTable* gesetzt und in *CommandText* wurde die passende Tabelle ausgewählt. Damit entsprechen diese drei Client-Datenmengen übrigens einer direkten Portierung einer BDE-Anwendung, in der die drei Tabellen über die Komponente *TTable* angesprochen wurden. Für größere Datenmengen kann diese simple Übertragung problematisch werden, weil das ClientDataSet die gesamte Tabelle in den Hauptspeicher lädt.
- Anders verhält es sich bei den Datenmengen *cdsTagesTermine*, *cdsTermineZuAdresse* und *cdsWhTermine*. Diese enthalten keine Original-Tabellen der Datenbank, sondern das Ergebnis von SQL-Abfragen, die weiter unten gezeigt sind. Auch für diese drei Datenmengen steht im Datenmodul je eine *TDataSource*-Komponente zur Verfügung.
- Um die Beschriftung und die Spaltenbreite von Feldern in einem DBGrid anzupassen, wurden außerdem in den meisten ClientDataSets persistente Feldkomponenten erzeugt und deren Properties wie etwa *DisplayLabel*, *DisplayWidth* gesetzt. Für die Datenmenge *cdsTermine* wurden außerdem zwei Lookup-Felder erzeugt, die dem Benutzer später im Termineingabe-Dialog (siehe Kapitel 7.5.3) bei der Auswahl von verknüpfter Adresse und Aktion behilflich sein können.
- Außerdem finden Sie in der Abbildung noch die folgenden Komponenten, deren Zweck in den späteren Kapiteln noch genauer erläutert wird: *SQLMonitor1* (Kapitel

7.5.7), das Duo *cdsAdressen2/AdressenSrc2* (Kapitel 7.5.5) und die Datenmenge *qryModifyTagesTermine*, die nicht an eine *DataSource* angeschlossen zu werden braucht (Kapitel 7.5.6).

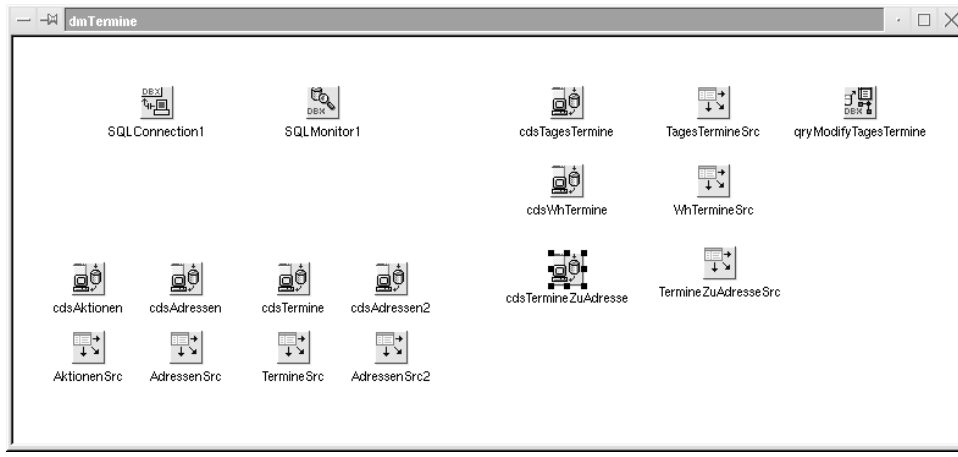


Abbildung 7.13: Das Datenmodul der Terminverwaltung

SQL-Abfragen

Mit SQL-Abfragen können Sie Teilmengen aus einer einzelnen Tabelle oder aus einer Verknüpfung mehrerer Tabellen vom Server abrufen. Die Datensätze, die das Ergebnis der Abfrage bilden, können als neue Datenmenge aufgefasst werden und werden daher in dbExpress auch als *TDataSet* angesehen. Eine generelle Beschreibung von SQL-Abfragen muss spezieller Datenbankliteratur überlassen bleiben; eine ausführliche Referenz finden Sie in der schon erwähnten *Language Reference*, die mit Interbase als PDF-Dokument mitgeliefert wird.

In dbExpress gibt es zwei Möglichkeiten, eine Datenmenge mit dem Ergebnis einer SQL-Abfrage zu verknüpfen:

- Wenn Sie *TSQLQuery* verwenden, können Sie in Delphi-5-kompatibler Weise eine SQL-Abfrage im Property *SQL* angeben (die Kompatibilität bezieht sich auf die *TTable*-Komponente). Allerdings handelt es sich bei *TSQLQuery* um eine unidirektionale Datenmenge mit allen an anderer Stelle schon aufgezählten Einschränkungen.
- In *TSQLClientDataSet* geben Sie den Abfrage-Text im Property *CommandText* ein, wenn Sie unter *CommandType* den Wert *ctQuery* eingestellt haben. Sobald Sie die so vorbereitete Komponente aktivieren (*Active*-Property oder *Open*-Methode), leitet

diese die Abfrage an den SQL-Server und nimmt die von diesem zurückgelieferten Datensätze in ihr Datenpaket *Data* auf (die Datensätze befinden sich dann also alle im Speicher der Kylix-Anwendung).

ür beide Properties stellt Ihnen Kylix einen *Anweisungstext-Editor* zur Verfügung, der Ihnen die Zusammenstellung der SQL-Anweisung dadurch erleichtert, dass er zwei Listen enthält, aus denen Sie die Namen von Tabellen und deren Feldern per Klick in den Abfragetext einfügen können (siehe Abbildung 7.14).

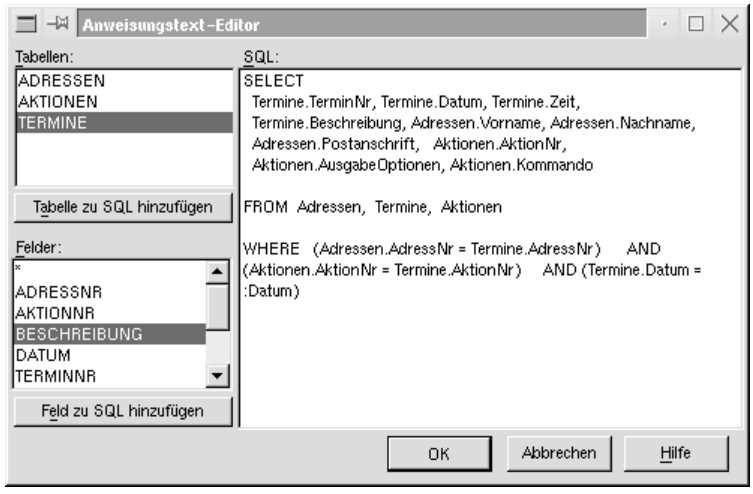


Abbildung 7.14: Editieren von SQL-Anweisungen mit den Bordmitteln von Kylix

Das Beispielprogramm gibt drei Beispiele für SQL-Abfragen. Zum Einstieg soll jedoch eine minimale Abfrage den grundsätzlichen Aufbau einer solchen Abfrage verdeutlichen:

```
select * from Adressen
```

Diese Abfrage wählt alle Spalten (*) der Adress-Tabelle und enthält keine weiteren Bedingungen zur Auswahl der Datensätze. Folge ist, dass das Ergebnis der Abfrage einfach aus dem gesamten Inhalt der Tabelle *Adressen* besteht. Bei sehr großen Tabellen sollte also genau diese Abfrage mit einem ClientDataSet vorsichtig verwendet werden, denn sie zwingt das ClientDataSet zur Verwaltung eines riesigen Datenpakets.

Die Datenmenge *cdsWhTermine* des Beispielprogramms soll nur die Termine enthalten, die mit einer Aktion verknüpft sind und die in bestimmten Abständen wiederholt werden – bei denen also eines der Felder *WdholenT/M/J* größer als Null ist (die Null bedeutet damit also »keine Wiederholung des Termins«, jedoch ist das eine reine Definitionsfrage). Für die Termine, auf die diese Bedingung zutrifft, sollen neben den

Spalten der Termintabelle auch die Felder *Kommando* und *AusgabeOptionen* der zugehörigen Aktion aus der Aktionstabelle ausgegeben werden (und zwar ohne die Verwendung eines Lookup-Feldes wie in der Datenmenge *cdsTermine*). Die dafür erforderliche SQL-Abfrage sieht wie folgt aus:

```
SELECT Termine.Beschreibung, Termine.WdholenT, Termine.WdholenM,
    Termine.WdholenJ, Termine.AktionNr, Termine.AdressNr,
    Aktionen.Kommando, Aktionen.AusgabeOptionen
FROM Termine, Aktionen
WHERE (Termine.AktionNr = Aktionen.AktionNr)
         AND  (    (Termine.WdholenT > 0)
                OR (Termine.WdholenM > 0)
                OR (Termine.WdholenJ > 0) )
```

Hier werden nun hinter *SELECT* alle Spalten aufgelistet, die in der Ergebnismenge enthalten sein sollen. Da in diesem Fall Spalten sowohl aus der Tabelle *Adressen* als auch aus *Aktionen* im Ergebnis vorkommen, ist die Ergebnismenge nicht mehr einer bestimmten Tabelle der Datenbank zuzuordnen, sondern sie stellt eine ganz neue Tabelle dar, in der Inhalte aller drei Tabellen kombiniert werden. Diese neue Tabelle wird nicht in der Datenbank gespeichert, sondern kann in diesem Beispielprogramm nur über eine Abfrage gewonnen werden.

Wenn Sie im ClientDataSet den Felder-Editor zu einer solchen Datenmenge aufrufen, werden Sie sehen, dass Sie für jede dieser Spalten eine eigene persistente Feldkomponente erzeugen können. Alles weitere in Kapitel 7.3 über persistente Felder Gesagte lässt sich somit auf die Ergebnisspalten einer SQL-Abfrage übertragen.

Joins

Im *From*-Teil der obigen Abfrage müssen die beiden Tabellen genannt werden, aus denen im vorherigen *Select*-Teil bereits Spalten genannt wurden. Wenn auf diese Weisen mehrere Tabellen in einer Ergebnismenge zusammengefasst werden, spricht man von einem *Join*. Ein solcher Join ist in der Praxis nur sinnvoll, wenn er durch eine Bedingung eingeschränkt wird, die im *Where*-Teil der Abfrage angegeben wird. Im obigen Listing ist `(Termine.AktionNr = Aktionen.AktionNr)` die Join-Bedingung. Ohne eine Join-Bedingung würde die *Select*-Abfrage alle Kombinationen aus Datensätzen der beiden Tabellen enthalten. Jeder einzelne Termin würde also mit allen in der Aktionstabelle definierten Aktionen kombiniert werden und für eine Datenbank mit 20 Terminen und fünf Aktionen würde eine Ergebnismenge mit 100 Datensätzen generiert werden. Wenn Sie eine formalere Beschreibung von Joins oder weitere Informationen über verschiedene Arten von Joins suchen, finden Sie diese sehr ausführlich in Büchern, die sich allgemein mit der Datenbanktheorie befassen (oben wird übrigens ein »inner join« durchgeführt, was zur Folge hat, dass Termine, die *nicht* mit einer Aktion verknüpft sind, gar nicht in das Ergebnis aufgenommen werden).

Filtern und Sortieren

Schließlich enthält die oben gezeigte Abfrage noch drei weitere Bedingungen, die mit *AND* mit der Join-Bedingung verknüpft werden. Hier werden einfach die *WdholenX*-Felder abgefragt, denn es sollen ja nur die wiederholten Termine abgefragt werden. Solche Bedingungen, die nur auf die Felder einer Tabelle Bezug nehmen, führen zu einer *Filterung*, die von der späteren Filterung in einem ClientDataSet – etwa mit dem *OnFilterRecord*-Ereignis (Kapitel 7.4.1) – zu unterscheiden ist: Während von der SQL-Anweisung ausgefilterte Records gar nicht an die dbExpress-Komponenten gesendet werden, werden die von Client-Datenmengen ausgefilterten Datensätze nur verborgen, bleiben aber weiterhin im Datenpaket der Datenmenge enthalten.

Sie können außerdem noch eine Sortierung für SQL-Ergebnismengen festlegen, indem Sie eine *Order-By*-Klausel an den *Where*-Teil anfügen. Im Beispielprogramm wurde hiervon kein Gebrauch gemacht und auch unsortierte Ergebnismengen können Sie ja durch ein ClientDataSet beliebig sortieren, indem Sie *IndexFieldNames* auf die zu sortierenden Spalten setzen.

SQL-Abfragen mit Parametern

Die zweite Abfrage-Datenmenge des Beispielprogramms, *cdsTagesTermine*, soll alle Termine eines bestimmten Tages angeben. Dabei sollen statt der Nummer der mit dem Termin verknüpften Adresse die Adressdaten und statt der Aktionsnummer die Aktionsdaten im Ergebnis aufgeführt werden. Die folgende Abfrage verbindet also alle drei Tabellen in einem Join:

```
SELECT Termine.TerminNr, Termine.Datum, Termine.Zeit,
   Termine.Beschreibung, Adressen.Vorname, Adressen.Nachname,
   Aktionen.AktionNr, Aktionen.AusgabeOptionen, Aktionen.Kommando
FROM Adressen, Termine, Aktionen
WHERE
       (Adressen.AdressNr = Termine.AdressNr)
   AND (Aktionen.AktionNr = Termine.AktionNr)
   AND (Termine.Datum = :Datum)
```

Die letzte Bedingung des *Where*-Teils sorgt für die Filterung der Ergebnisliste auf die Termine eines gewählten Tages. Dieser Tag wird in der Abfrage nicht als festes Datum angegeben, da der Benutzer zur Laufzeit wählen soll, für welchen Tag die Termine angezeigt werden sollen. In der Abfrage wird der Tag daher als Parameter (»:Datum«) angegeben, der zur Laufzeit mit dem tatsächlichen Wert ersetzt wird.

Die ClientDataSet-Komponente sucht ihr *CommandText*-Property automatisch nach Parametern ab (die sie ja an dem vorangestellten Doppelpunkt erkennen kann). Alle gefundenen Parameter werden dann der Reihe nach im *Params*-Property aufgeführt, das Sie auch schon zur Entwurfszeit einsehen können.

Folgende wichtige Vorbereitungen für die Verwendung der Parameter können Sie schon zur Entwurfszeit treffen: Öffnen Sie den Property-Editor für das *Params*-Property und stellen Sie für jeden Parameter die folgenden Properties ein: *DataType* für den Datentyp des Parameters und *ParamType* auf *ptInput*, wenn es sich wie hier um die Parameter einer *Select*-Abfrage handelt.

Der Wert des Parameters, den Sie ebenfalls schon zur Entwurfszeit setzen können, wird im Beispielprogramm erst zur Laufzeit gesetzt, und zwar wenn der Benutzer den Schalter *Update* neben dem Eingabefeld für das Datum drückt (siehe Abbildung 7.15):

```
with dmTermine do begin // Datenmodul
  cdsTagesTermine.Close;
  cdsTagesTermine.Params[0].AsString := DateEdit.Text;
  cdsTagesTermine.Open;
end;
```

Die Änderung der Parameter ist nur bei geschlossener Datenmenge möglich, daher wird vorher die *Close*-Methode aufgerufen. Das abschließende *Open* führt dazu, dass die Abfrage mit dem geänderten Parameterwert neu gestartet wird und im Falle des Beispielprogramms die Termine eines anderen Tages angezeigt werden.

Hinweis: Ein solcher Parameter wird auch benötigt, wenn Sie die Detailtabelle aus einem Haupt-/Detailformular per SQL generieren. So könnte beispielsweise eine Detailtabelle, die alle Termine anzeigt, die eine bestimmte Aktion referenzieren, wie folgt aufgebaut werden:

```
SELECT ...
FROM Termine
WHERE Termine.AktionNr = :AktionNr
```

Der Parameter, hier *AktionNr*, ist vom aktuellen Datensatz in der Aktionstabelle abhängig und muss auch bei jeder Änderung der Position in dieser Tabelle geändert werden. Wenn Sie *MasterSource* und *MasterFields* wie in Kapitel 7.4.5 einstellen, sorgt ein ClientDataSet automatisch für das Setzen dieses Parameters und für die Aktualisierung der Detail-Abfrage.

Aufgaben eines Datenmoduls *R129*

Außer den Komponenten kann ein Datenmodul natürlich auch noch Code enthalten und die Datenzugriffskomponenten bieten ja auch zahlreiche Ereignisse an, die im Datenmodul bearbeitet werden können. Anhand der Beispielanwendung seien hier nun ein paar dieser Ereignisse vorgestellt:

- Zunächst bietet sich das *OnCreate*-Ereignis des Datenmoduls dazu an, die Verbindung zu aktivieren (*TSQLConnection.Connect*) und die Datenmengen zu öffnen (*TDataSet.Open*), die während der gesamten Programmlaufzeit benötigt werden. Das Beispielprogramm öffnet bei diesem Ereignis alle seine Client-Datenmengen.

- Das Ereignis *OnNewRecord* einer Client-Datenmenge eignet sich dazu, bestimmte Felder mit Standardeinstellungen oder automatisch errechneten Werten vorzubelegen (z.B. mit dem aktuellen Datum). Das Beispielprogramm verknüpft dieses Ereignis bei der Client-Datenmenge für die Termine mit der folgenden Methode, die mit Hilfe der persistenten Feldkomponenten die Werte einiger Spalten initialisiert:

```
procedure TdmTermine.cdsTermineNewRecord(DataSet: TDataSet);
begin
  cdsTermineWDHOLENT.Value := 0;
  cdsTermineWDHOLENM.Value := 0;
  cdsTermineWDHOLENJ.Value := 0;
  cdsTermineZEIT.AsDateTime := 0;
end;
```

  Da in der Definition der Termindatenbank nicht vorgeschrieben wurde, dass diese Felder einen Wert haben müssen, würde das Programm auch ohne diese Methode gut funktionieren. Der Benutzer könnte dann die Felder einfach offen lassen und die datensensitiven Komponenten würden dann für diese Felder keine Null, sondern einfach ein leeres Feld anzeigen (im Programm lässt sich dieser Fall übrigens mit der Methode *IsNull* von *TField* abfragen).

- Das Ereignis *OnGetText* der Feldkomponenten kann dazu verwendet werden, den in den datensensitiven Komponenten angezeigten Text zu modifizieren. Die folgende Methode des Beispielprogramms sorgt beispielsweise dafür, dass für das Feld *Zeit* nur der Zeitanteil des Feldwertes angezeigt und der Datumsanteil ignoriert wird:

```
procedure TdmTermine.cdsTagesTermineZEITGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if not Sender.IsNull then
    Text := TimeToStr(Sender.Value);
end;
```

  Der Inhalt des Feldes selbst wird dadurch zunächst nicht beeinflusst. Erst wenn der Benutzer den angezeigten Wert editiert und die Änderungen speichert, wirkt sich unter Umständen aus, welcher Text von *GetText* angezeigt wurde. (Entsprechend zu *OnGetText* gibt es natürlich auch ein Ereignis *OnSetText*, in dem Sie den vom Benutzer eingegebenen Wert interpretieren und eventuell vor dem Setzen des Feldes verändern können).

- Das Ereignis *AfterPost* einer Client-Datenmenge eignet sich sehr gut dazu, die ApplyUpdates-Methode aufzurufen, um die Änderungen des Benutzers auch an den Datenbank-Server zu übertragen (siehe Kapitel 7.5.4).
- Die Bearbeitung des Ereignisses *TClientDataSet.OnReconcileError* ist in Kapitel 7.5.5 beschrieben.
- Für die Bearbeitung des Events *BeforeUpdateRecord* (auch ein Ereignis der Client-Datenmengen) im Beispielprogramm siehe Kapitel 7.5.6.

Die Zusammenfassung solcher Ereignisbearbeitungen in einem Datenmodul kann dazu führen, dass der datenbankabhängige Code Ihrer Anwendung leicht zu warten ist, da alle relevanten Methoden sich an einem gemeinsamen Ort befinden. Außerdem werden auf diese Weise die Formular-Units entlastet. Sie müssen sich nicht um die Initialisierung neuer Records und auch kaum noch um die Anzeige der Daten oder die Aktualisierung kümmern. Manche Formulare des Beispielprogramms kommen daher fast ohne eigenen Code aus.

7.5.3 Die Formulare

Das Hauptfenster der Beispielanwendung (Abbildung 7.15) enthält zwei spezielle Ansichten der Termindatenbank, die über die beiden im letzten Kapitel beschriebenen SQL-Abfragen *cdsTagesTermine* und *cdsWhTermine* gewonnen werden. Ohne dass dafür im Hauptformular weitere Programmierung notwendig wäre, werden beide Datenmengen einfach durch je eine *DBGrid*-Komponente angezeigt. Die Anzeige der Daten in diesem Grid richtet sich nach den Properties der persistenten Feldkomponenten und nach dem Wert, den die im letzten Kapitel beschriebene *OnGetText*-Methode zurückliefert. Da sich sowohl die Feldkomponenten als auch die *OnGetText*-Bearbeitung im Datenmodul befinden, muss sich das Hauptformular gar nicht um diese Angelegenheit kümmern.

Außer dem Ansehen dieser Datenmengen hat der Benutzer noch die Möglichkeit, das erste DBGrid mit gewissen Einschränkungen zu editieren. Dies stellt für das Programm eine besondere Herausforderung dar, die in Kapitel 7.5.6 genauer beschrieben wird.

Ansonsten dient das Hauptformular hauptsächlich zum Öffnen der anderen Fenster. Einige Methoden des Hauptformulars werden in den folgenden Kapiteln noch besprochen (die Reaktion auf den Schalter *Update* neben dem Datumsfeld wurde bereits in Kapitel 7.5.2 erklärt).

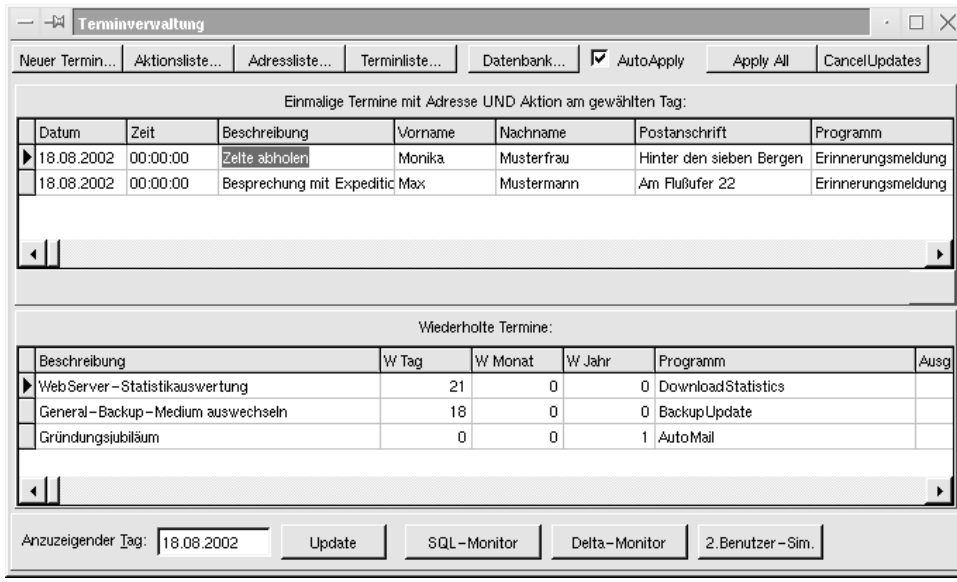


Abbildung 7.15: Das Hauptformular zeigt das Ergebnis der SQL-Abfragen und dient als Startpunkt für die rmulare zum Editieren der einzelnen Tabellen.

Modale Dateneingabe-Fenster

Die Schalter AKTIONSLISTE... und TERMINLISTE... des Hauptformulars öffnen jeweils ein einfaches modales Fenster, in dem die jeweilige Tabelle editiert werden kann. Die Formulare für diese beiden Fenster bringen keine Neuerungen, was die in ihnen enthaltenen Komponenten betrifft, so dass hier nicht weiter auf diese eingegangen werden soll.

Eine grundsätzliche Überlegung wert ist jedoch, wie die modale Anzeige eines solchen Fensters mit dem Aktualisierungsprozess einer Datenmenge harmoniert. An dieser Stelle soll es nur um das Änderungsprotokoll gehen, also um die Frage, wie die Daten mit *Post* in das Änderungsprotokoll geschrieben werden. Kapitel 7.5.4 erläutert später, wie das Eintragen der Änderungen für alle Datenmengen des Programms an einer zentralen Stelle vorgenommen wird, so dass sich die einzelnen Formulare nicht mehr darum zu kümmern brauchen.

Wenn der Benutzer nun in einem modalen Fenster eine Dateneingabe-Komponente editiert, kann er das Fenster normalerweise über einen OK- oder einen ABBRUCH-Schalter schließen. Diese Schalter sind nicht »datensensitiv«, haben also keine Ahnung von der Datenbank und rufen auch nicht etwa *Post* oder *Cancel* auf, wie es zum Abschluss eines Dateneingabedialogs empfehlenswert wäre. Daher werden beide Schalter im Beispielprogramm wie folgt bearbeitet:

```
procedure TfrmAdressListe.OkButtonClick(Sender: TObject);
begin
  if dmTermine.cdsAdressen.State = dsEdit then
    dmTermine.cdsAdressen.Post;
end;

procedure TfrmAktionsListe.CancelButtonClick(Sender: TObject);
begin
  if dmTermine.cdsAdressen.State = dsEdit then
    dmTermine.cdsAdressen.Cancel;
end;
```

Post bzw. *Cancel* brauchen nur aufgerufen zu werden, wenn sich die Datenmenge im Editiermodus befindet.

Die Adressliste

Die Adresstabelle wird über ein drittes Formular zur Eingabe bereitgestellt, das Sie in Abbildung 7.16 sehen können. Links oben befindet sich zunächst ein DBGrid, das nur die Spalten *Vorname* und *Nachname* des Adressen-ClientDataSets anzeigt (die anderen Spalten wurden aus dem Property *TDBGrid.Columns* gelöscht). Es dient als Namensliste, damit der Benutzer schneller einen bestimmten Namen auswählen kann, dessen Adressdaten er dann in den rechts daneben befindlichen Einzel-Steuerelementen ansehen und editieren kann.

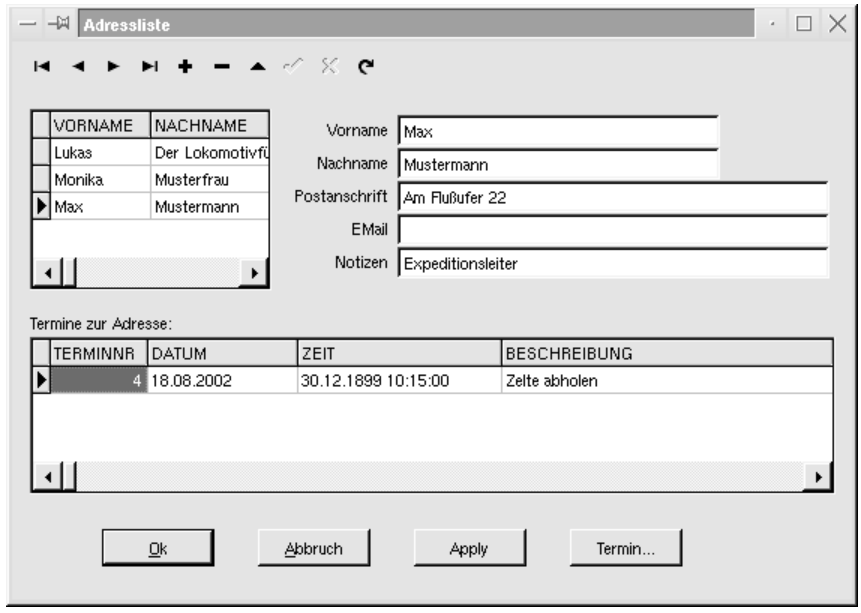


Abbildung 7.16: Das Formular zum Editieren der Adressliste

Das untere DBGrid gibt eine Ansicht auf die Datenmenge *cdsTermineZuAdresse*, die alle mit der aktuellen Adresse verknüpften Termine enthält. Das Adressformular wird dadurch zu einem Haupt-/Detailformular, wobei die Details nur zur Information angezeigt werden, also nicht editierbar sind.

Da auch das Adressformular seine Funktion hauptsächlich auf die selbsttätige Arbeitsweise der datensensitiven Komponenten stützt, bleibt hier nur eine erwähnenswerte Aufgabe des Formulars übrig: Der Aufruf des Termin-Dialogs über den Schalter ERMIN.

Im Gegensatz zum Aufruf des Termindialogs aus dem Hauptformular soll über den ERMIN-Schalter im Adressformular ein Termin speziell für die gerade bearbeitete Adresse eingegeben werden können. Die Verknüpfung mit der Adresse soll automatisch erfolgen, so dass dem Benutzer die Eingabe des Adressfeldes des neuen Termins erspart bleibt. Die folgende Methode setzt diesen Plan in die Tat um:

```
procedure TfrmAdressListe.Button3Click(Sender: TObject);
begin
  dmTermine.cdsTermine.Insert;
  dmTermine.cdsTermineADRESSNR.Value :=
    dmTermine.cdsAdressen.FieldByName('ADRESSNR').Value;
  if frmNeuerTermin.ShowModal = mrOk then
    dmTermine.cdsTermine.Post
  else dmTermine.cdsTermine.Cancel;
end;
```

Sie sorgt mit der schon aus Kapitel 7.2.3 bekannten *Insert*-Methode dafür, dass ein neuer Datensatz in das Termin-ClientDataSet eingefügt wird, und setzt dessen Feld *AdressNr* auf die Adressnummer des aktuellen Datensatzes im Adress-ClientDataSet. Nach dieser Vorbereitung wird der Termindialog ganz normal mit *ShowModal* aufgerufen, wie es auch beim Aufruf des Dialogs aus dem Hauptformular geschieht.

Der Termindialog

Beim Termindialog (Abbildung 7.1.2) lassen sich zwei Besonderheiten feststellen: Zum einen enthält er das für dieses Buch einzige Anwendungsbeispiel für *TDBLookupComboBox*-Komponenten, die die im Felder-Editor definierten Lookup-Felder der Komponente *cdsTermine* anzeigen. Sie wurden wie jede andere datensensitive Komponente einfach über ihr *DataField*-Property mit dem entsprechenden Feld verknüpft und ersparen dem Benutzer, die mit dem Termin verknüpfte Aktion und Adresse über eine Nummer eingeben zu müssen.

Zum anderen braucht für diesen Dialog kein einziges Ereignis bearbeitet zu werden, da alles automatisch abläuft oder (im Falle des Aufrufs von *Post* und *Cancel*) an den Aufrufer des Dialogs delegiert wird (siehe Listing oben).

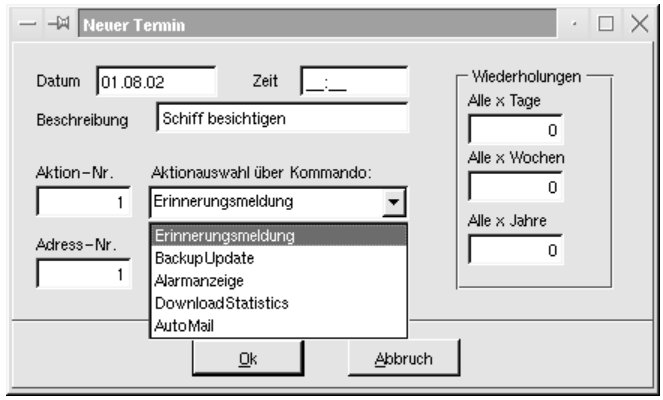


Abbildung 7.17: Ein Dialog zur Eingabe eines neuen Termins

Die weiteren Formulare der Anwendung werden in den folgenden Kapiteln noch zur Sprache kommen.

7.5.4 Updates und das Änderungsprotokoll

In Kapitel 7.2.6 wurde bereits erläutert, wie die Änderungen an dem Datensatz einer SQLClientDataSet-Komponente zuerst mit *Post* in das Änderungsprotokoll geschrieben werden und dort verweilen, bis sie entweder mit *CancelUpdates* verworfen oder mit *ApplyUpdates* in die Datenbank geschrieben werden. Während die Browser-Beispielprogramme aus Kapitel 7.2 sich noch gar nicht um die Änderungen gekümmert haben und die Dateidatenbank aus Kapitel 7.3.6 das Einfügen aller Datensätze selbst vornimmt, hat die aktuelle Beispielanwendung erstmals die Aufgabe zu erfüllen, Änderungen des Benutzers an den Datenbank-Server weiterzuleiten.

Einfache Updates *R127*

Eine besonders einfache Möglichkeit, sicherzustellen, dass alle Änderungen des Benutzers gespeichert werden, besteht darin, *ApplyUpdates* immer dann aufzurufen, wenn der Benutzer die Änderung eines Datensatzes abschließt (wenn er also eine *Post*-Operation verursacht – entweder implizit, indem er zu einem anderen Datensatz wechselt, oder explizit, indem er etwa den Post-Schalter einer *DBNavigator*-Komponente betätigt).

Eine solche Verknüpfung ist über das Ereignis *TDataSet.AfterPost* möglich. Im Minimalfall würde es bereits genügen, hier einfach nur *ApplyUpdates* aufzurufen. Im Datenmodul der Beispielanwendung sind die *AfterPost*-Ereignisse *aller* editierbaren ClientDatenmengen mit der folgenden Methode verknüpft, die quasi eine Luxus-Version des *ApplyUpdates*-Aufrufs darstellt:

```
procedure TdmTermine.cdsGeneralAfterPost(DataSet: TDataSet);
var
  ErrorCount: Integer;
begin
  with DataSet as TSQLClientDataSet do begin
    if AutoApply then begin
      ErrorCount := ApplyUpdates(-1);
      If ErrorCount <> 0 then
        ShowMessage(Format('%d Datensätze wurden nicht eingetragen.
            Refresh wird nicht durchgeführt.', [ErrorCount]))
      else
        Refresh;
    end;
  end;
end;
```

Zunächst wird *ApplyUpdates* nicht für ein bestimmtes ClientDataSet aufgerufen, sondern für die im Parameter *DataSet* übergebene Komponente (der *DataSet*-Parameter ist mit dem *Sender*-Parameter von Steuerelement-Ereignissen vergleichbar). Diese Unabängigkeit von einer bestimmten Datenmenge ermöglicht erst die Verknüpfung dieser Methode mit den *AfterPost*-Ereignissen mehrerer Datenmengen (diese Datenmengen müssen hier aber vom Typ *TSQLClientDataSet* sein, da die obige Methode den *DataSet*-Parameter mit *as* in diese Klasse umzuwandeln versucht).

Dann berücksichtigt die Methode eine interne Variable namens *AutoApply*, über die das automatische Annehmen der Änderungen abgeschaltet werden kann. Diese Möglichkeit dient hier zum Experimentieren und zum Vergleichen und wird gleich bei der Untersuchung des Änderungsprotokolls eine wichtige Rolle spielen. Im Hauptformular des Programms finden Sie eine CheckBox, über die Sie *AutoApply* zur Laufzeit verndern können.

Die ApplyUpdates-Methode

Als Nächstes findet der entscheidende Aufruf von *ApplyUpdates* statt. Als Parameter erwartet diese Methode die maximale Zahl von Fehlern, die auftreten darf, bevor der Aktualisierungsvorgang abgebrochen wird. Sollte es zu einem solchen Abbruch kommen, werden alle vorher bereits erfolgreich im Server eingetragenen Änderungen zurückgenommen (hierfür wird intern von dem Transaktionsmechanismus der SQL-Server Gebrauch gemacht). Im Beispielprogramm ist ein solcher Abbruch des Aktualisierungsvorgangs auszuschließen, da als Parameter »-1« übergeben wird, also beliebig viele Fehler auftreten dürfen.

Obwohl Sie als Parameter statt »-1« beliebige andere Zahlen angeben können, stehen sich hierfür hauptsächlich zwei gegensätzliche Prinzipien gegenüber:

- Ein Parameter von »-1« bewirkt, dass so viele geänderte Datensätze wie möglich in die Datenbank übernommen werden. Wenn es bei einem bestimmten Datensatz zu einem Fehler kommt, wird dieser eine Datensatz nicht eingetragen, jedoch wird das Eintragen anderer Datensätze nicht verhindert.
- Ein Parameter von »0« lässt die Methode nach dem Alles-oder-nichts-Prinzip vorgehen: Entweder werden alle Datensätze in die Datenbank geschrieben oder keiner (hier kommt der erwähnte Transaktionsmechanismus zum Tragen, indem beim ersten Update-Fehler die Eintragung der bisher schon erfolgreich eingetragenen Datensätze rückgängig gemacht wird).

Die Zahl der aufgetretenen Fehler wird in jedem Fall als Ergebnis von *ApplyUpdates* zurückgeliefert und von der obigen Methode in einem Meldungsfenster angezeigt (falls von Null verschieden).

Wie es überhaupt zu Aktualisierungsfehlern kommen kann und wie sie behandelt werden können, ist Thema von Kapitel 7.5.5.

Hinweis: Da in Kapitel 7.5.3 die *Ok*-Schalter der modalen Formulare mit einem *Post*-Aufruf verknüpft wurden, wird bei jedem normalen Schließen dieser Formulare auch die obige Methode aufgerufen. Sofern Sie also zur Laufzeit die *AutoApply*-Option nicht abschalten, ist die Termindatenbank nach dem Schließen eines modalen Eingabefensters immer auf dem neuesten Stand.

Erneuerung der Daten nach ApplyUpdates

An dieser Stelle interessiert noch der abschließende Aufruf von *Refresh* im obigen Listing. Dieser sorgt dafür, dass alle Ansichten der aktualisierten Datenmenge ebenfalls aktualisiert werden. Dabei werden sowohl Änderungen am Datensatz, die auf der Seite des Servers durchgeführt werden, sichtbar (wie etwa die in Kapitel 7.5.1 beschriebene automatische Nummerierung durch Trigger), als auch Änderungen, die vom Benutzer zur Behandlung von Update-Fehlern im Reconcile-Dialog veranlasst wurden (siehe Kapitel 7.5.5).

Das Änderungsprotokoll einsehen

Eines der »Experimental-Formulare« des Beispielprogramms befasst sich mit dem Änderungsprotokoll der Client-Datenmengen. Es erklärt auch den Daseinszweck der im letzten Listing abgefragten Variable *AutoApply*. Wenn Sie *AutoApply* abschalten, werden die Änderungen nicht mehr automatisch an den Server geschickt, sondern häufen sich so lange im Änderungsprotokoll an, bis Sie einen der *Apply*-Schalter drücken, die das Beispielprogramm extra für diesen Zweck vorsieht.

Hinweis: Nach der Eingabe eines *neuen* Datensatzes muss in diesem Beispielprogramm auf jeden Fall ein *ApplyUpdates* durchgeführt werden, bevor ein weiterer neuer Datensatz eingegeben wird, denn erst beim Senden des Datensatzes an den Server erhält dieser dort durch die Trigger-Funktion eine eindeutige Nummer zugeordnet. Wenn Sie zwei neue Datensätze hintereinander eingeben, ohne dass der erste eine Nummer erhalten hat, resultieren zwei Datensätze mit gleichem Inhalt in der Nummernspalte (in beiden Fällen »leer« bzw. *Null*). Da jedoch ein eindeutiger Index auf die Nummernspalte gelegt wurde, wird bei einem doppelten Wert eine Indexfehler-Exception ausgelöst. Wenn Sie also *AutoApply* abgeschaltet haben, sollten Sie nach der Eingabe eines neuen Datensatzes manuell den *Apply*-Schalter des jeweiligen Formulars drücken.

Das in Abbildung 7.18 gezeigte Formular ermöglicht zur Laufzeit einen Einblick in das nderungsprotokoll; es ist nur dann sinnvoll zu gebrauchen, wenn das automatische *Apply* abgeschaltet ist, denn sonst werden die Änderungsprotokolle ja immer automatisch geleert, wenn einer der modalen Dialoge der Anwendung geschlossen wird.

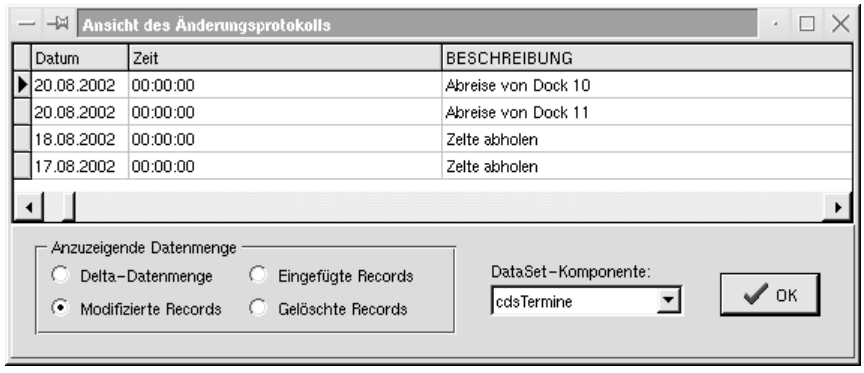


Abbildung 7.18: Ansicht der geänderten Datensätze über die ClientDataSet-Option »StatusFilter = usModified«

Das Formular ist so allgemein gehalten, dass Sie es auch leicht in eigene Anwendungen einbauen können. Der Aufruf des Formulars im Beispielprogramm (über den Schalter DELTA-MONITOR) läuft wie folgt ab:

```
procedure TfrmTermine.Button3Click(Sender: TObject);
begin
  DeltaView.CDSContainer := dmTermine;
  DeltaView.ShowModal;
end;
```

or der Anzeige des Formulars wird dessen Variable *CDSContainer* auf ein Datenmodul gesetzt, das ClientDataSets enthält. Diese ClientDataSets können dann bei der folgenden Verwendung des Formulars aus einer KomboBox ausgewählt werden. Diese

spezielle Funktion soll aber hier nicht untersucht werden, sondern wir nehmen nun an, dass bereits eine der Datenmengen ausgewählt und in der Variablen *CurrentDataSet* gespeichert wurde.

Über die Radioschaltergruppe des Formulars können Sie nun zwischen vier Ansichten wählen, wobei die letzten drei zusammengehören. In der ersten Ansicht soll der Inhalt des *Delta*-Properties der gewählten Datenmenge dargestellt werden. Auf direktem Weg ist das nicht möglich, da die datensensitiven Komponenten immer nur den Inhalt des *Data*-Properties anzeigen (dem vom ClientDataSet verwalteten Datenpaket). Da jedoch *Data* und *Delta* zuweisungskompatibel sind, können wir das *Delta*-Property unserer Datenmenge an das *Data*-Property einer anderen Datenmenge zuweisen. Bei dieser Zuweisung erhält diese zweite Datenmenge eine *Kopie* des *Delta*-Inhalts, allerdings in ihrem *Data*-Property, so dass der Inhalt des Datenpakets nun in datensensitiven Komponenten sichtbar gemacht werden kann.

Das DeltaView-Formular enthält zu diesem Zweck eine eigene Komponente *ClientDataSet1*, die mit dem Inhalt des in Frage stehenden *Delta*-Properties gefüllt werden kann. *Delta* ist allerdings nur gültig, wenn die Datenmenge aktiv ist (*Active = True*) und Änderungen vorgenommen wurden (*ChangeCount > 0*). Dies wird im Folgenden vorher abgefragt:

```
// Typ von CurrentDataSet: TSQLClientDataSet
procedure TDeltaView.UpdateView;
begin
  if Assigned(CurrentDataSet) then
    if rgViewType.ItemIndex = 0 then // Index der RadioGroup
    begin
{$B-} if (CurrentDataSet.Active = True) and
         (CurrentDataSet.ChangeCount > 0) then
      begin
        ClientDataSet1.Data := CurrentDataSet.Delta;
        // DataSource1 leitet die Daten an das DBGrid weiter:
        DataSource1.DataSet := ClientDataSet1;
      end;
    end else begin
```

Die drei anderen Ansichten von *DeltaView* kommen ohne das *Delta*-Property aus, sie beziehen sich aber dennoch auf die Änderungen, die in der Client-Datenmenge durchgeführt wurden. *TClientDataSet* erlaubt nämlich über das Property *StatusFilter*, die Anzeige auf die geänderten Datensätze zu beschränken. *StatusFilter* kann ein beliebige Kombination von *usModified* (geänderte Datensätze), *usInserted* (neu eingefügte Datensätze) und *usDeleted* (gelöschte Datensätze) enthalten. Im Programm wird jeweils nur eine dieser Konstanten gesetzt:

```
        // (Fortsetzung der Methode UpdateView)
        DataSource1.DataSet := CurrentDataSet;
        if CurrentDataSet.Active = True then
          case rgViewType.ItemIndex of
            1: CurrentDataSet.StatusFilter := [usModified];
            2: CurrentDataSet.StatusFilter := [usInserted];
            3: CurrentDataSet.StatusFilter := [usDeleted];
          end;
      end;
  end;
```

In der Praxis können Sie die *StatusFilter*-Option dazu verwenden, dem Benutzer eine ückmeldung darüber zu geben, welche Änderungen er zuletzt durchgeführt hat bzw. welche Änderungen bei der nächsten Aktualisierung an den Server geschickt werden würden. Dies ist natürlich besonders interessant bei Anwendungen nach dem Aktenkoffermodell, bei denen sich über einen längeren Zeitraum Änderungen im Protokoll anhäufen.

7.5.5 Mehrbenutzersimulation

Das Eintragen von Änderungen auf dem Datenbank-Server ist ein kritisches Unterfangen, weil in der Praxis häufig mehrere Anwender mit derselben Datenbank arbeiten und man es daher nicht ausschließen kann, dass zwei oder mehr Anwender zur selben Zeit den gleichen Datensatz ansehen und sich entschließen, ihn zu ändern.

Programmtechnisch sieht das dann beispielsweise so aus, dass auf dem Rechner jedes Anwenders eine eigene ClientDataSet-Komponente arbeitet, die eine Kopie des Datensatzes im Speicher aufbewahrt. Wenn nun der erste Anwender den Datensatz ändert und zum Server zurückschickt, aktualisiert dieser erwartungsgemäß die Datenbank entsprechend. Wenn aber nun der zweite Anwender kurze Zeit später seine eigenen nderungen abschickt, so könnten dadurch Änderungen des ersten Anwenders überschrieben werden, die dem zweiten Anwender noch gar nicht angezeigt wurden.

Außerdem könnte es sein, dass beide Anwender nur ein einzelnes Feld geändert haben, und zwar jeweils ein anderes. Soll der Datenbank-Server dann die beiden geänderten Felder behalten oder den Datensatz so akzeptieren, wie ihn der zweite Anwender abgeschickt hat (ohne die Änderung des ersten Anwenders)?

Ein zweites Adresslisten-Formular

Im Beispielprogramm können Sie diese Situation simulieren, indem Sie den Schalter 2-ENUTZER-SIM. drücken. Sie erhalten dann neben dem schon in Kapitel 7.5.3 beschriebenen Adresseingabe-Formular noch ein zweites Fenster, das ebenfalls eine Adressliste enthält. Wichtig ist, dass beide Fenster nicht über dieselbe ClientDataSet-Komponente

auf die Tabelle zugreifen. Hierfür wird nun die Datenmenge *cdsAdressen2* benötigt, die in Kapitel 7.5.2 schon als Teil des Datenmoduls gezeigt wurde.

Diese Datenmenge lädt wie *cdsAdressen* die gesamte Adresstabelle und verfügt über ein eigenes Änderungsprotokoll. Sie können nun, wenn Sie die beiden Fenster aufgerufen haben, in beiden Änderungen am gleichen Datensatz vornehmen, ohne eines der Fenster zu schließen oder einen *Apply*-Schalter zu drücken. Erst nachdem Sie den gleichen Datensatz zweifach geändert haben, drücken Sie in beiden Fenstern den *Apply*-Schalter. Beim zweiten *Apply*-Versuch sollte es nun zu einem Aktualisierungs-Konfikt kommen.

Der Reconcile-Dialog

Wie Update-Fehler am besten behoben werden, hängt vom konkreten Anwendungsfall ab. Am flexibelsten ist es, dem Benutzer die Wahl zu lassen, was im Konfliktfall geschehen soll. In der Objektablage von Kylix finden Sie bereits einen vorgefertigten Dialog, der dafür verwendet werden kann (Seite *Dialoge*, Eintrag *Dialogfeld Fehler beheben*); in Abbildung 7.1.4 ist dieser Dialog in Aktion zu sehen.

Dieser Dialog wurde mit der voreingestellten KOPIEREN-Option[1] der Objektablage in das Beispielprogramm eingefügt und dann im Verzeichnis des Beispielprogramms gespeichert. Dieses verfügt also über eine eigene Kopie des Dialogs, die unabhängig von der in der Objektablage befindlichen Kopie verändert werden kann (im Beispielprogramm wurde allerdings keine Veränderung vorgenommen).

Die Verwendung des Dialogs ist sehr einfach, denn die Dialog-Unit (im Beispielprogramm unter dem Namen *RecError* gespeichert) enthält bereits eine vordefinierte Funktion zum Behandeln eines Update-Konflikts. Der Update-Konflikt wird zunächst in Form eines *OnReconcileError*-Ereignisses der verantwortlichen Datenmenge angezeigt. Dieses Ereignis wird nun wie folgt bearbeitet:

```
// uses RecError;
procedure TdmTermine.cdsAdressenReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Wie in Abbildung 7.1.4 gezeigt, kann der Benutzer so zur Laufzeit zwischen verschiedenen Möglichkeiten wählen, wie der Konflikt zu beheben ist. Wenn Sie *OnReconcileError* nicht bearbeiten, bekommt der Benutzer im Fehlerfall nur eine Exception-Meldung und seine Änderungen werden nicht auf dem Server eingetragen.

1. Hier ist der KOPIEREN-Radioschalter im DATEI | NEU-Dialog gemeint (siehe Kapitel 1.6.4).

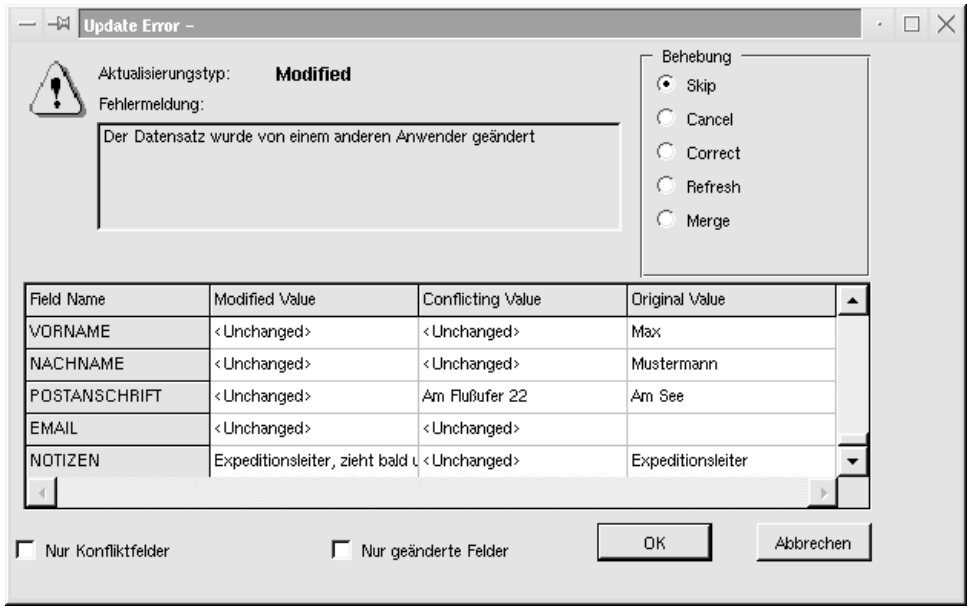


Abbildung 7.19: Der von Borland mitgelieferte Dialog zur Behandlung von Update-Konflikten

Im Beispielprogramm wurde die oben gezeigte Methode sowohl mit *cdsAdressen* als auch mit *cdsAdressen2* verknüpft, da der Update-Konflikt in beiden Datenmengen auftreten kann, je nachdem, in welcher Reihenfolge die *ApplyUpdates* ausgeführt werden. Die anderen Datenmengen des Programms verfügen, unter der Annahme, dass sie jeweils nur von einem Benutzer gleichzeitig gesehen werden, nicht über eine *OnReconcileError*-Behandlung.

Hinweis: Während Sie einen Update-Konflikt als Ereignis bearbeiten, wird ein solcher Konflikt innerhalb der DataCLX als Exception weitergeleitet. Wenn Sie die Option TOOLS | DEBUGGER-OPTIONEN... | SPRACH-EXCEPTIONS | BEI DELPHI-EXCEPTIONS STOPPEN nicht deaktivieren, hält Kylix Ihr Programm daher bei einem Update-Konflikt in jedem Fall an, auch wenn das *OnReconcileError*-Ereignis vorbildlich behandelt wird. Dies liegt daran, dass der Debugger beim Auftreten der Exception noch nicht wissen kann, ob sie gleich behandelt wird oder nicht.

7.5.6 Updates per SQL

Dieser Abschnitt soll Ihnen einen kurzen Ausblick auf ein Thema geben, das an sich eine ausführliche Behandlung verdienen würde, die jedoch hier der Spezialliteratur berlassen werden muss. Es geht um Datenmengen, deren Änderungen nicht automatisch von *ApplyUpdates* im Server eingetragen werden können.

Ein Beispiel für eine solche Datenmenge ist die obere Terminliste im Hauptformular des Beispielprogramms (Abbildung 7.15). In ihr sind Felder aus allen drei Tabellen der Anwendung kombiniert. Wird nun ein Datensatz dieser Tabelle geändert, müssten im Extremfall Änderungen in allen drei Tabellen eingetragen werden. Das Standardverhalten der DataCLX besteht jedoch darin, die Änderungen nur an die erste Tabelle zu senden, die im *From*-Teil der *Select*-Anweisung aufgeführt ist.

Eigene Updates durchführen

Die radikalste Lösung dieses Problems besteht darin, das gesamte Update selbst durchzuführen, also auch die SQL-Update-Anweisung selbst zu generieren. *TClientDataSet* ermöglicht dies über das Ereignis *BeforeUpdateRecord*, das für jeden zu aktualisierenden Datensatz vor der Aktualisierung ausgelöst wird. In der Methode für dieses Ereignis können Sie nun die Aktualisierung selbst durchführen und den Parameter *Applied* auf *True* setzen, um dem ClientDataSet mitzuteilen, dass die Arbeit erledigt ist. Bei der Aktualisierung muss der Parameter *UpdateKind* berücksichtigt werden, der angibt, ob es sich um eine Änderungs-, eine Lösch- oder eine Einfügeoperation handelt:

```
procedure TdmTermine.cdsTagesTermineBeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind = ukModify then
    ... Aktualisierung der Datensatzänderung auf dem Server
  else if UpdateKind = ukDelete then... // Löschen auf dem Server
  else if UpdateKind = ukInsert then... // Einfügen auf dem Server
  Applied := True;
end;
```

Bevor wir uns ansehen, wie dies im Beispielprogramm konkret im Fall von *ukModify* funktioniert, sei noch einmal der entscheidende Teil der *Select*-Anweisung gezeigt, durch den die Datenmenge *cdsTagesTermine* aufgebaut wird:

```
SELECT Termine.TerminNr, Termine.Datum, Termine.Zeit,
   Termine.Beschreibung, Adressen.Vorname, Adressen.Nachname,
   Adressen.Postanschrift, Aktionen.AktionNr,
   Aktionen.AusgabeOptionen, Aktionen.Kommando
FROM Adressen, Termine, Aktionen
WHERE ...
```

Um die Änderungen in der Datenbank einzutragen, müssen die neuen Werte von *TerminNr*, *Datum*, *Zeit* und *Beschreibung* in die Tabelle *Termine* eingetragen werden. *Vorname*, *Nachname* und *Postanschrift* gehören in die Tabelle Adressen; die restlichen Felder in die Tabelle Aktionen. Für jede Tabelle muss eine eigene SQL-Update-Anweisung ausgeführt werden.

Als Beispiel soll hier die Aktualisierung der Termintabelle dienen, die mit der folgenden SQL-Anweisung durchgeführt wird:

```
update Termine
  set
    Datum = :Datum,
    Zeit = :Zeit,
    Beschreibung = :Beschreibung
  where
    TerminNr = :Old_TerminNr;
```

Die vier Parameter dieser Anweisung sind bei jeder Aktualisierung mit den aktuellen Feldwerten bzw. mit dem alten Feldwert der Terminnummer zu füllen (auch wenn die erminnummer durch den Benutzer nicht editiert werden kann, ist es nahe liegend, die alte Nummer zu verwenden, denn es soll ja der alte Datensatz in der Datenbank gefunden werden).

Hinweis: In der gezeigten *Update*-Anweisung wird der zu aktualisierende Datensatz über die Terminnummer identifiziert. Hierfür ist es wichtig, dass das Feld *TerminNr* überhaupt in der Datenmenge enthalten ist (dafür muss es in der *Select*-Anweisung genannt werden). Im Beispielprogramm wird die Terminnummer nicht im DBGrid angezeigt, da sie für den Benutzer vermutlich keine Rolle spielt, jedoch wird sie in der *Select*-Anweisung genannt, ebenso wie natürlich auch die Aktionsnummer, die für das Update der Aktionstabelle benötigt wird.

SQL-Anweisungen ausführen

Um eine solche eigene SQL-Anweisung durchzuführen, bieten sich mit den dbExpress-Komponenten zwei Wege an:

- Mit der Methode *TSQLConnection.Execute* können Sie Ihrer Verbindungskomponente eine SQL-Anweisung als String übergeben, die dann direkt ausgeführt wird.
- Wenn Sie die SQL-Anweisung lieber als Komponente zur Entwurfszeit definieren wollen, können Sie eine *TSQLDataSet*-Komponente in das Datenmodul einfügen und die SQL-Anweisung in deren *CommandText*-Property eingeben. *CommandType* kann dann auf *ctQuery* gesetzt werden, jedoch wird die DataSet-Komponente nicht aktiv geschaltet oder mit *Open* geöffnet, sondern die SQL-Anweisung wird mit der Methode *ExecSQL* ausgeführt. Diese Möglichkeit wurde im Beispielprogramm gewählt.

Alte und neue Feldwerte

Nun ist die Frage, wie das Programm an die Feldwerte des zu aktualisierenden Datensatzes gelangen soll, denn diese müssen ja noch als Parameter in die SQL-Anweisung eingesetzt werden. Hier kommt nun ein weiterer Parameter des *BeforeUpdateRecord-*

Ereignisses ins Spiel: die Datenmenge *DeltaDS*. Diese enthält alle Felder der geänderten Datenmenge, und Sie können mit den gewohnten Methoden *FieldByName* auf diese zugreifen. Die bisher noch nicht besprochenen *TField*-Properties *NewValue* und *OldValue* geben Ihnen nun Zugriff auf den alten bzw. neuen Wert des Feldes. *NewValue* ist nur bei den Feldern gesetzt, die gegenüber dem alten Wert geändert wurden.

Um beispielsweise den Parameter *:Beschreibung* aus der obigen SQL-Anweisung ermitteln zu können, kann der folgende Code ausgeführt werden:

```
Beschreibung := DeltaDS.FieldByName('Beschreibung').NewValue;
if Beschreibung = '' then
  Beschreibung := DeltaDS.FieldByName('Beschreibung').OldValue;
```

Dieser kann dann in eine *TSQLDataSet*-Komponente eingesetzt werden – wie auch die in Kapitel 7.5.2 verwendete *TClientDataSet* verfügt diese über ein *Params*-Property, in dem die Parameter der Abfrage aufgezählt werden und über das die Werte zugewiesen werden können:

```
qryModifyTagesTermine.Params[2].Value := Beschreibung;
```

Dieser Vorgang wird in der Beispielanwendung auch für die anderen drei Parameter der oben gezeigten SQL-Anweisung wiederholt – eine Wiederholung, die wir uns hier ersparen werden.

Handarbeit für die Erzeugung der SQL-Anweisung

Leider führte die oben gezeigte Verwendung des *Params*-Properties im Test auch nur dazu, dass die Komponente die Parameter derart falsch in die SQL-Anweisung einsetzte, dass die Syntax der Anweisung zerstört wurde und Interbase eine Fehlermeldung zurücklieferte. Das Beispielprogramm geht den Weg daher ganz zu Fuß und erzeugt die SQL-Anweisung mitsamt Parametern selbst. Als Schablone dient nun eine Stringkonstante, in der die Parameter jeweils durch ein »%s« stellvertreten werden:

```
const
  sqlstrModifyTagesTermineTermine =
    'update Termine'+
    ' set'+
    '   Datum = "%s",'+
    '   Zeit = "%s",'+
    '   Beschreibung = "%s"'+
    ' where'+
    '   TerminNr = "%s"';
```

Zum Einsetzen von Werten in einen solchen String sieht die Laufzeitbibliothek von Kylix die *Format*-Funktion vor. Mit ihrer Hilfe erhält die *TSQLDataSet*-Komponente des Beispielprogramms eine SQL-Anweisung ohne syntaktische Mängel:

```
qryModifyTagesTermine.CommandText :=
    Format(sqlstrModifyTagesTermine, [t1, t2, t3, t4]);
```

Dabei sind die Variablen *t?* die Werte der Felder, die wie oben gezeigt vorher mit *NewValue* und *OldValue* ermittelt wurden. Die fertige SQL-Anweisung wird schließlich mit

```
qryModifyTagesTermine.ExecSQL(False);
```

ausgeführt, wobei der Parameter *False* besagt, dass die Komponente nicht selbst versuchen soll, Parameter in den *CommandText* einzuarbeiten.

Grenzen und Erweiterungsmöglichkeiten

Um das Beispielprogramm übersichtlich zu halten, werden übrigens in der Terminliste des Hauptformulars nur die Änderungen berücksichtigt, die sich auf die Tabellen *Aktionen* und *Termine* beziehen. Die aus der Adress-Tabelle stammenden Felder wurden auf *ReadOnly* geschaltet, damit hier Änderungen erst gar nicht möglich sind (dieses *ReadOnly*-Property ist über die persistenten Feldkomponenten der Datenmenge *cdsTagesTermine* im Datenmodul zu finden; rufen Sie für diese Komponente den Felder-Editor auf und wählen Sie eines der Felder).

Außerdem wurden Löschoperationen in der kombinierten Terminliste unterbunden, indem der entsprechende Schalter der *DBNavigator*-Komponente entfernt wurde (Property *VisibleButtons*), ebenso wie der Schalter zum Einfügen eines neuen Records. Für beide Operationen wären nämlich weitere SQL-Anweisungen für alle betroffenen abellen notwendig. Schließlich wurde in der zweiten Terminliste des Hauptformulars jegliches Editieren unmöglich gemacht, in dem die zugrunde liegende ClientDataSet-Komponente *cdsWhTermine* auf *ReadOnly* gesetzt wurde.

Updates für eine einzelne Tabelle

Mit dem in Kapitel 7.5.7 vorgestellten SQL-Monitor können Sie übrigens erforschen, welche Update-Anweisung die Komponenten von Kylix verwenden, wenn Sie *ApplyUpdates* aufrufen, ohne das Ereignis *BeforeUpdateRecord* zu bearbeiten. Wenn Sie den Code der oben gezeigten *BeforeUpdateRecord*-Methode ausklammern, das Programm starten und eine Änderung in die Hauptformular-Tabelle einzutragen versuchen, erhalten Sie erst einmal eine Exception, können dann aber im weiteren Programmverlauf im SQL-Protokoll beispielsweise den folgenden Text lesen:

```
Update Adressen
  set
    Beschreibung = ?
  where
    TerminNr = ? and
    Datum = ? and
```

```
Zeit = ? and
Vorname = ? and
...
```

Wenn wir annehmen, dass bei diesem Update immer nur eine der Tabellen aktualisiert werden soll (und nicht alle drei Tabellen wie im Beispielprogramm), so lassen sich Möglichkeiten finden, die automatisch erzeugte SQL-Anweisung so zu korrigieren, dass sie funktioniert. Dazu könnten wir hinter *Update* die Tabelle angeben, die aktualisiert werden soll. Nehmen wir einmal an, die Nennung der Tabelle *Adressen* im obigen Listing wäre schon korrekt, dann müsste als Weiteres beachtet werden, dass im *where*-Teil nur Felder angegeben werden, die auch in dieser Tabelle enthalten sind. Wir müssten also einfach alle »falschen« Felder aus der Anweisung entfernen.

Tatsächlich enthalten die Komponenten von Kylix einen Mechanismus, der genau dieser Vorgehensweise entspricht, er soll hier als Ausblick kurz skizziert werden (Details dazu gibt die Online-Hilfe):

- Über die Bearbeitung des Ereignisses *OnGetTableName* von *TClientDataSet* können Sie der Komponente mitteilen, welche Tabelle aktualisiert werden soll.
- Über das Property *ProviderFlags* der persistenten Feldkomponenten für die Felder der Tabelle können Sie einstellen, ob und wann dieses Feld in der Update-Anweisung berücksichtigt werden soll. Im obigen Beispiel würden wir beispielsweise die Flags *ProviderFlags.pfInUpdate* und *ProviderFlags.pfInWhere* auf *False* setzen, um ein Feld komplett aus der automatisch erzeugten Update-Anweisung herauszuhalten.

7.5.7 Ein SQL-Monitor

Zum Abschluss soll noch ein Fenster in die Anwendung eingebaut werden, durch das man einen Blick hinter die Kulissen der Datenzugriffskomponenten werfen und die SQL-Anweisungen, mit denen die dbExpress-Komponenten im Hintergrund mit dem Datenbank-Server kommunizieren, beobachten kann.

In der dbExpress-Komponentenpalette finden Sie zu diesem Zweck eine Komponente namens *TSQLMonitor*, die gewissermaßen als Ersatz für den eigenständigen SQL-Monitor von Delphi fungiert. *TSQLMonitor* ist eine nicht-visuelle Komponente, die über das Property *SQLConnection* mit der Verbindungskomponente verknüpft, deren Datenverkehr überwacht werden soll. Am sichersten ist es, die Monitor-Komponente in dasselbe Datenmodul zu setzen wie die Verbindungskomponente. Zwar können Sie die Komponente auch in einem Formular unterbringen, jedoch funktioniert die Verbindung zwischen *SQLMonitor* und *SQLConnection* nur dann, wenn das Formular der Monitor-Komponente *vor* dem Datenmodul erzeugt wird (dies bezieht sich auf die Reihenfolge der Formularerzeugung in den Projektoptionen).

In der Beispielanwendung wird das Datenmodul jedoch als Erstes initialisiert, da es die Grundlage für die gesamte Anwendung liefert. Die Monitor-Komponente wird daher im Datenmodul platziert, und ein eigenes Formular dient dazu, die Meldungen dieser Komponente anzuzeigen (Abbildung 7.1.3).

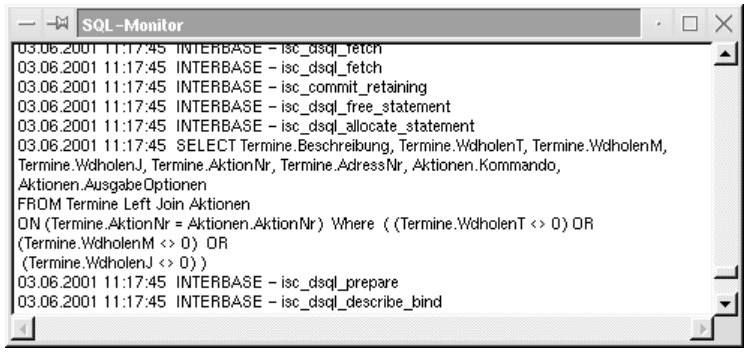


Abbildung 7.20: Dieses Formular zeigt die TraceList der nicht-visuellen TSQLMonitor-Komponente an.

Ein SQL-Monitorfenster mit Zeitstempeln *R134*

Die Monitorkomponente muss zuerst aktiviert werden, indem das *Active*-Property gesetzt wird. Dies kann schon zur Entwurfszeit geschehen. Die Meldungen stehen im Folgenden über das Property *TSQLMonitor.TraceList* zum Abruf bereit. Über zwei Ereignisse – *OnTrace* und *OnLogTrace* – können Sie sich von neuen Meldungen informieren lassen und diese im Falle von *OnTrace* sogar bearbeiten, bevor sie Eingang in das Protokoll finden.

Das Monitor-Formular des Beispielprogramms enthält den folgenden Code:

```
procedure TFrmSQLMonitor.FormCreate(Sender: TObject);
begin
  Memo1.Lines := dmTermine.SQLMonitor1.TraceList;
  dmTermine.SQLMonitor1.OnTrace := SQLMonitor1Trace;
end;

procedure TFrmSQLMonitor.FormDestroy(Sender: TObject);
begin
  dmTermine.SQLMonitor1.OnTrace := nil;
end;

procedure TFrmSQLMonitor.SQLMonitor1Trace(Sender: TObject;
  CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
var
  TimeStamp: String;
begin
  TimeStamp := DateTimeToStr(Now)+'  ';
```

```
  if Strlen(CBInfo.pszTrace) + length(TimeStamp) + 1
       < sizeof(CBinfo.pszTrace) then
  begin
    Move(CBInfo.pszTrace[0], CBInfo.pszTrace[length(TimeStamp)],
                             strlen(CBInfo.pszTrace));
    Move(TimeStamp[1], CBInfo.pszTrace[0], length(TimeStamp));
  end;
  Memo1.Lines.Add(CBInfo.pszTrace);
end;
```

Zunächst wird beim *OnCreate*-Ereignis das *OnTrace*-Event des SQL-Monitors mit einer Formularmethode verknüpft, da dies im Objektinspektor nicht möglich ist, weil die Komponente nicht dem Formular gehört. Passend dazu wird die Ereignisverknüpfung in *OnDestroy* wieder gelöst, denn es könnte ja sein, dass das Formular einmal vor dem Datenmodul freigegeben wird. Ein Versuch des SQL-Monitors, die Formularmethode aufzurufen, würde dann im Daten-Nirvana enden. In der *OnCreate*-Methode werden außerdem die bisherigen Meldungen des SQL-Monitors im Memo-Feld angezeigt (das Datenmodul des SQL-Monitors wird vor dem Anzeige-Formular erzeugt, daher können schon vorher erste Meldungen protokolliert worden sein).

Die Bearbeitung der *OnTrace*-Methode fügt dem Meldungstext *CBInfo.pszText* noch einen Zeitstempel mit der aktuellen Zeit hinzu. Da *pszText* nicht als normaler String, sondern als festes Zeichen-Array mit 1024 Zeichen Länge definiert ist, findet dieses Hinzufügen oben statt, indem Speicherbereiche verschoben und kopiert werden (siehe auch Online-Hilfe zu *Move*). Die Länge eines nullterminierten Strings wie des erwähnten Zeichenarrays muss außerdem mit *strlen* (statt mit *length*) ermittelt werden.

Der erweiterte *pszText* wird dann vom Beispielformular an den Text des Memos angehängt und von *SQLMonitor1* automatisch in eine Datei geschrieben (sein *AutoSave*-Property wurde auf *True* gesetzt).

Hinweis: Sie können das SQL-Protokoll auch einfach in eine Datei schreiben lassen, indem Sie die Properties *FileName* und *AutoSave* von *TSQLMonitor* verwenden. Die neuesten Meldungen dieser Datei können Sie dann beispielsweise in einem Terminalfenster mit `tail -f Dateiname` auf einer Konsole anzeigen und fortlaufend aktualisieren lassen.

8 Komponentenentwicklung

Der Begriff der »Komponente« kann unter Kylix von verschiedenen Ebenen aus verstanden werden. Er kann sich speziell auf die Bestandteile der Komponentenpalette beziehen oder ganz allgemein auf sprachunabhängige Software-Komponenten.

Dieses Kapitel konzentriert sich auf die speziellere Bedeutung des Komponentenbegriffs, also auf die »CLX-Komponenten«, womit sowohl die mit Kylix mitgelieferten Komponenten als auch die selbst geschriebenen gemeint sein sollen. Es beschreibt das Komponentenkonzept der CLX und zeigt die internen Abläufe, auf denen CLX-Komponenten basieren, sowie die Techniken, mit denen sie entwickelt werden. Schließlich beschreibt es die Entwicklung von einigen Beispiel-Komponenten, die Sie sinnvoll in Ihren Anwendungen einsetzen können und die teilweise auch im TreeDesigner Verwendung finden.

Die Komponenten dieses Buchs

Nachdem Sie den Inhalt der CD zu diesem Buch auf der Festplatte installiert haben, haben Sie zwei verschiedene Möglichkeiten, die in diesem Kapitel entwickelten Komponenten zu installieren:

- Wählen Sie KOMPONENTE | PACKAGES INSTALLIEREN..., drücken Sie den Schalter HINZUFÜGEN... und fügen Sie das Package der Buch-Komponenten `awdlew.bpl` aus dem `Buch-Verzeichnis/complib` hinzu. Daraufhin fügt Kylix das Package in die Liste des Package-Installationsdialogs ein, selektiert und lädt es automatisch. Schließen Sie den Dialog daraufhin mit *Ok*. Die Komponenten werden auf der Seite *EW-Buch* in der Komponentenpalette installiert, die Sie natürlich im Eigenschaftsdialog der Palette beliebig umbenennen können.
- Alternativ können Sie auch DATEI | ÖFFNEN wählen und den Package-Quelltext `awdlew.dpk` im Verzeichnis `[Buch-Verzeichnis]/complib` öffnen. Diese Vorgehensweise erlaubt Ihnen, sich das Package vor der Installation im Package-Editor genauer anzusehen. Drücken Sie in diesem Editor den Aktionsschalter INSTALL., um das Package in die Komponentenpalette aufzunehmen.

8.1 Das Komponentenkonzept von Kylix

Das Komponentenkonzept erfordert zunächst einmal eine neue Begriffsunterscheidung: Die Bezeichnung *Benutzer* meint in diesem Kapitel normalerweise den Entwickler, der die fertigen Komponenten bei seiner Anwendungsentwicklung benutzt, und nicht den *Anwender*, der mit der fertigen Anwendung arbeitet. Über dem *Komponentenentwickler* befinden sich also zwei Ebenen von Benutzern, die in diesem speziellen Kapitel durch die etwas willkürliche Terminologie (*Komponenten-*)*Benutzer/(End-)Anwender* unterschieden werden sollen (wobei der Anwender in diesem Kapitel kaum vorkommt).

Komponenten bedeuten für den Benutzer und den Entwickler der Komponenten unterschiedliche Dinge:

- Für den Benutzer betont Kylix besonders stark die ereignisorientierte Programmierung, während die objektorientierte Programmierung etwas in den Hintergrund tritt. Der Benutzer braucht nicht sehr viel von OOP zu verstehen, da er nur innerhalb einer Klasse, der Formularklasse, arbeitet und von dieser aus die einzelnen Komponenten anspricht (im einfachsten Fall braucht der Benutzer noch nicht einmal das Klassenkonzept zu kennen, sondern kann die Klasse mit dem zur Laufzeit erscheinenden Fenster gleichsetzen). Außerdem stellen die Komponenten alle Ereignisse als Events im Objektinspektor zur Verfügung. Das Überschreiben von virtuellen Methoden ist nicht erforderlich, und interne Abläufe innerhalb der Komponenten bleiben dem Benutzer weitgehend verborgen.
- Für den Entwickler von Komponenten wird die objektorientierte Sichtweise des Programms noch weiter verstärkt, denn Komponenten können als Objekte mit besonders scharfen Umrissen angesehen werden. Natürlich sollten Objekte im OOP immer »deutliche Umrisse« haben, das Komponentenkonzept von Kylix fördert es jedoch besonders, dass Komponenten unabhängig von anderen Komponenten als eigenständige Software-Bausteine verwendbar sind. Der Komponentenentwickler sollte sich gut mit der Sprache Object Pascal und mit deren OOP-Konzepten auskennen. (Zu den schon in Kapitel 2 beschriebenen Spracheigenschaften kommen in diesem Kapitel nur noch wenige hinzu.)

8.1.1 Das Wesen einer Komponente

Komponenten sind spezielle Objekte, die sich von normalen Objekten dadurch unterscheiden, dass Sie sie zur Entwurfszeit in das Formular einfügen und im Objektinspektor bearbeiten können. Dazu bedarf es keiner neuen Eigenschaften der Programmiersprache: Auch Komponenten nutzen lediglich die normalen Object-Pascal-Sprachelemente, insbesondere natürlich die Klassen und die Properties (allerdings kann kaum ein Zweifel daran bestehen, dass einige Merkmale von Object Pascal speziell zur Unterstützung der Komponenten eingeführt wurden).

Um den Weg in die Kylix-IDE zu finden, müssen Komponentenklassen mindestens zwei Voraussetzungen erfüllen:

- Sie müssen direkt oder indirekt von der Klasse *TComponent*, die in der CLX definiert ist (siehe Kapitel 3.1.2), abstammen. Die Vererbung sorgt dafür, dass Ihre eigenen von *TComponent* abgeleiteten Klassen von der Kylix-IDE genauso behandelt werden wie die von Kylix definierten Komponenten, die natürlich auch alle von *TComponent* abgeleitet sind.
- Nach diesem ersten Schritt verfügen Sie bereits über eine Komponente, die Sie zur Laufzeit wie jede andere Klasse verwenden können. Um die Verbindung zur Kylix-IDE herzustellen, damit die Komponente auch schon zur Formular-Entwurfszeit einsatzfähig ist, müssen Sie Ihre Komponentenklasse bei Kylix registrieren.

Obwohl Sie die fertigen Komponenten zur Entwurfszeit eines Formulars auf vielfältige Weise manipulieren können und der Objektinspektor genau über die Properties Bescheid wissen muss, genügt es, für die Registrierung jeder Komponente eine einzige Zeile Code zu schreiben, alles andere erledigt die Vererbung.

CLX-Klassen und die Komponenten

Alle Klassen auf dem Vererbungspfad der CLX von *TObject* bis *TComponent* spielen eine wichtige Rolle:

- Die Laufzeit-Typinformationen sind der Grund dafür, dass der Objektinspektor den Typ der Properties erkennt und diese in einer angemessenen Form darstellen kann. Diese Typinformationen sind schon in der Klasse *TObject* vorhanden, allerdings gehört diese Klasse weniger zur CLX, als vielmehr zur Sprache Object Pascal.
- *TPersistent* spielt eine wichtige Rolle beim Erzeugen der `xfm`-Formulardateien und ist somit einer der Schlüssel zur Rekonstruktion der Formulare zur Laufzeit.
- *TComponent* definiert die Verbindungen unter den Komponenten, sorgt also quasi für den Zusammenhalt zwischen Formular und Komponenten.
- *TWidgetControl* ist zwar nicht erforderlich, um das Komponentenkonzept durchzusetzen, ist aber die Schlüsselklasse, ab der eine Komponente richtig visuell wird, also als Fenster oder Steuerelement sichtbar wird.

8.1.2 Überblick über die Komponentenentwicklung

Der Code von Komponenten nutzt normalerweise mehr Sprachelemente von Object Pascal, als die Teile der Anwendung, die Komponenten lediglich benutzen. Der grobe Ablauf der Komponentenentwicklung ist jedoch nicht komplizierter als der Ablauf zum Entwickeln einer normalen Kylix-Anwendung:

- Am Anfang steht hier nicht der visuelle Entwurf (es sei denn, die Komponente kapselt ein Formular, siehe Kapitel 8.7), sondern das Implementieren der Komponente in einer Unit. Hier können Sie genauso verfahren, als wollten Sie eine neue Objektklasse entwerfen, die Sie direkt im Programm nutzen wollen. (Die Klasse muss allerdings wie schon erwähnt direkt oder indirekt von *TComponent* abgeleitet sein.)
- Sobald diese Unit sich wie jede andere Unit übersetzen lässt, muss sie um wenige Zeilen Registrierungs-Code erweitert werden, durch den die Kylix-IDE sich Zugriff auf die Komponente verschaffen kann. Dadurch wird die Unit bereits zu einer fertigen Komponenten-Unit, die Sie beispielsweise mit KOMPONENTE | KOMPONENTE INSTALLIEREN... in die Palette aufnehmen können.

Der zweite Schritt kann auch etwas umfangreicher als wenige Zeilen ausfallen, wenn Sie weitere Schnittstellen der IDE ansprechen oder Editoren für Ihre Komponente oder ür Properties registrieren wollen.

Die Schnittstelle zu Kylix

Die minimale Schnittstelle zu Kylix besteht aus einer Prozedur wie der folgenden:

```
procedure Register;
begin
  RegisterComponents('Zum Buch', [THotkeyManager]);
end;
```

Sie registriert eine Komponente für die Komponentenpalette der IDE. Zusätzlich stehen Ihnen die folgenden weiteren Optionen zur Verfügung:

- Für die Registrierung von Editoren für Properties oder andere, nicht im Objektinspektor sichtbare Eigenschaften der Komponente gibt es weitere Registrierungsmethoden.
- Um Ihre Komponente in der Komponentenpalette durch eine eigene Bitmap darzustellen, legen Sie eine Ressourcen-Datei im DCR-Format an.
- Um Online-Hilfstexte zur Verfügung zu stellen, wie sie für die vorgefertigten Komponenten bestehen, schreiben Sie unter Berücksichtigung zusätzlicher Konventionen eine Hilfedatei.
- Bei dem Vorhaben, in Ihrem Programm mit der Kylix-IDE Verbindung aufzunehmen, stehen Ihnen Schnittstellen des *OpenTools API* zur Verfügung.

Die genannten Themen werden ausführlicher in den Kapiteln 8.4 und 8.6.5 besprochen. Lediglich die Hilfedateien können hier nicht besprochen werden, denn wie diese erzeugt werden, hängt davon ab, welches der unzähligen auf dem Markt angebotenen ools Sie verwenden (und die Verwendung eines solchen Tools ist normalerweise sehr zu empfehlen).

8.1.3 Das Package-Konzept

Das Package-Konzept ist für die Komponentenentwicklung von großer Bedeutung. Da Packages im Normalfall die meiste Zeit im Hintergrund wirken und Sie nur bei der Entwicklung und der Installation von Komponenten zwingend mit Packages und den damit verbundenen Erweiterungen der Kylix-IDE in Berührung kommen, wurde es im bisherigen Verlauf des Buchs noch nicht erläutert. Trotzdem können Sie aus den Packages auch für normale Anwendungen gewichtige Vorteile ziehen, siehe hierzu den abschließenden Abschnitt zu den *Laufzeit-Packages*.

Unter Delphi 1 und 2 war die Komponentenbibliothek sehr übersichtlich organisiert: Alle Komponenten, die in der IDE installiert wurden, fügte Delphi zu einer einzigen großen Datei hinzu, die schon in der Basisausstattung fast 1,5 MByte groß war. Die einzelnen, zur Bibliothek gehörenden Units konnten nur statisch zu den ausführbaren Dateien von Anwendungen hinzugebunden werden.

Seit Delphi 3 besteht die Komponentenbibliothek aus mehreren getrennten Bibliotheken, den *Packages*. Eine Übersicht über alle zurzeit in der IDE installierten Packages erhalten Sie mit KOMPONENTE | PACKAGES INSTALLIEREN; wählen Sie aus der Liste ein Package aus, können Sie sich mit dem Schalter *Komponenten* auch noch eine Übersicht über die in diesem Package enthaltenen Komponenten anzeigen lassen.

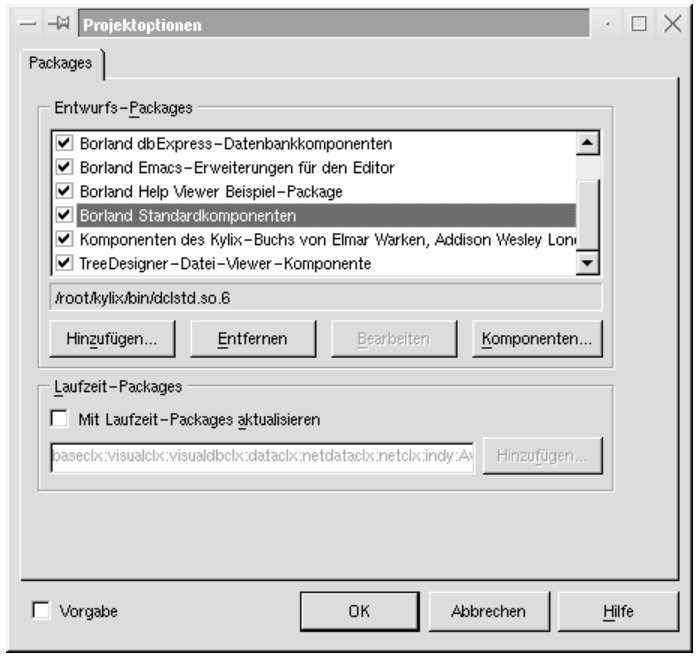


Abbildung 8.1: Der Dialog zur Wahl der aktiven und zum Installieren neuer Packages

Laufzeit-Packages

So wie Kylix selbst beim Entwurf der Formulare dynamisch geladene Packages verwendet, können auch Sie die Komponenten dynamisch zu Ihren Anwendungen binden lassen. Per Voreinstellung erzeugt Kylix ausführbare Dateien, in die alle benötigten Komponenten direkt eingebunden werden. Wenn Sie aber in den Projektoptionen die Option MIT LAUFZEIT-PACKAGES COMPILIEREN einschalten, erzeugt Kylix eine ausführbare Datei, die Laufzeit-Packages verwendet. Für die Weitergabe einer Anwendung bedeutet das einen etwas größeren Aufwand, da neben der ausführbaren Datei auch noch die gerade benötigten Laufzeit-Packages weitergegeben werden müssen; dem stehen aber auch Vorteile gegenüber:

- Bei sehr kleinen Programmen mit z.B. nur einem Formular und hundert Zeilen Code wird die Größe der ausführbaren Datei drastisch reduziert: Statt mit 300-400 KByte können Sie nun eher mit 20-60 KByte rechnen, die nur die Formulare Ihrer Anwendung und den von Ihnen geschriebenen Code enthalten.
- Haben Sie viele kleine Anwendungen, ergibt sich auf der Festplatte eine deutliche Platzersparnis, da die Laufzeit-Packages nur einmal für alle Anwendungen gespeichert werden müssen.

In Kapitel 8.2.3 kommen wir wieder auf die Laufzeit-Packages zurück; dort werden wir uns auch den Aufbau eines solchen Packages genauer ansehen.

Entwurfszeit-Packages

Entwurfszeit-Packages sind spezielle Packages zur Einbindung der Komponenten in die Komponentenpalette der Kylix-IDE. Im einfachsten Fall kann die IDE natürlich dasselbe Laufzeit-Package verwenden wie auch Ihre Anwendung. Wenn aber in der Kylix-IDE zusätzliche Funktionalität installiert werden soll, die im normalen Laufzeit-Package unnötig ist, bietet es sich an, eine spezielle Package-Version als Entwurfszeit-Package zu erstellen. So bestehen die zum Lieferumfang von Kylix gehörenden Entwurfszeit-Packages (deren Namen zumeist mit dem Präfix *dcl* beginnen) unter anderem aus den Komponenten- und Property-Editoren, die Sie in der IDE zu den verschiedenen Komponenten aufrufen können. Die Entwurfszeit-Packages enthalten aber normalerweise *nur* diese zusätzliche Funktionalität, darum muss die IDE neben den Entwurfszeit-Packages auch noch die normalen Laufzeit-Packages laden. Intern ist dies dadurch automatisiert, dass die Laufzeit-Packages in der *requires*-Klausel (siehe Kapitel 8.2.3) der Entwurfszeit-Packages angegeben sind.

Ein Vorteil der Verwendung von Packages auch für die Kylix-IDE liegt in der erheblich schnelleren Installation und Deinstallation von Komponenten und Komponentengruppen in die Komponentenpalette der IDE. Statt jedes Mal die komplette Komponentenbibliothek neu zu kompilieren, genügt es nun, einen kleinen Bestandteil davon als

Package in den Speicher zu laden oder aus ihm zu entfernen. Im schon erwähnten Package-Installationsdialog brauchen Sie nur das Markierungsfeld neben einem Package anzuklicken, um das Package zu aktivieren oder zu deaktivieren.

Hinweis: Durch das Deaktivieren nicht benötigter Packages kann die Ladezeit der Kylix-IDE erheblich beschleunigt werden.

8.2 Beispiele und Installation

Bevor wir in die Details der Komponentenentwicklung einsteigen, soll dieses Kapitel ein sehr einfaches Beispiel geben, das bereits aus einer sinnvoll einsetzbaren Komponente besteht. Das erste Beispiel soll jedoch eine Minimalkomponente sein.

8.2.1 Starthilfe durch den Komponenten-Experten

Den in Kapitel 8.1.2 beschriebenen Grundlagen folgend benötigen wir zuerst eine neue Unit, die die Komponente beherbergen soll. Um das Grundgerüst dieser Unit nicht selbst schreiben zu müssen, geben wir ein solches beim Komponenten-Experten in Auftrag, der sich in der Objektablage auf der Seite *Neu* unter dem Namen *Komponente* bzw. hinter dem Menüpunkt KOMPONENTE | NEUE KOMPONENTE verbirgt.

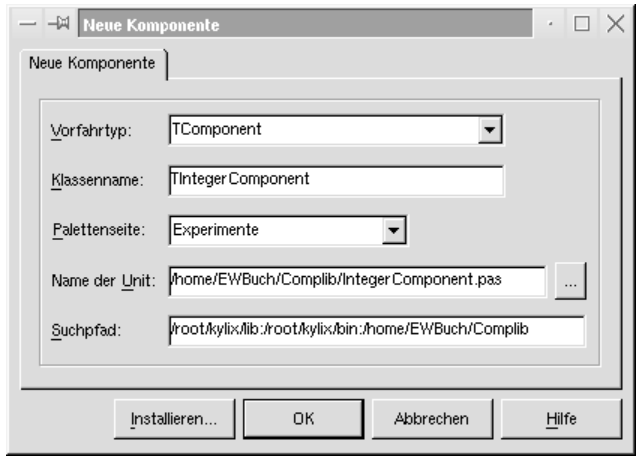


Abbildung 8.2: Die Expertendaten für die erste Beispielkomponente

Die Schnittstelle des Experten besteht aus der in Abbildung 8.2 gezeigten Dialogbox, die Sie zu fünf Angaben auffordert:

- Im ersten Eingabefeld geben Sie die Basisklasse der neuen Komponente an,
- im darunter befindlichen Feld den Namen der neuen Komponentenklasse,
- im dritten Feld eine Seite der Komponentenpalette, in die die neue Komponente bei der später erfolgenden Installation eingetragen werden soll.
- Unter NAME DER UNIT können Sie der neuen Komponente bereits eine Speicherposition zuweisen, was den Vorteil hat, dass der Bibliothekssuchpfad (unterstes Dialogfeld) automatisch erweitert wird, wenn Sie die Komponente in einem Verzeichnis speichern, das noch nicht in diesem Pfad enthalten ist.
- Das Feld SUCHPFAD ist quasi ein Spiegel des in den Umgebungsoptionen eingestellten Suchpfades. Änderungen dieses Feldes wirken sich nicht auf die erzeugte Komponenten-Unit, sondern nur auf die Umgebungsoptionen aus.

Durch Drücken von OK starten Sie die Expertentätigkeit, die sich jedoch sehr schnell in einem kleinen Gerüst einer noch unbenannten Unit erschöpft, das einen Klassenrumpf und eine Registrierungsroutine enthält. Ein solches, um zwei Zeilen erweitertes Unit-Gerüst finden Sie im nächsten Abschnitt.

Die Angaben, die Sie in der Dialogbox gemacht haben, sind nur ein- bzw. zweimal in der neuen Unit aufgeführt. Da Kylix die Eingaben in die Dialogbox schnell wieder vergisst, macht es nichts aus, wenn Sie Klassenname, Basisklasse und Palettenseite nachäglich ändern.

8.2.2 Eine Minimalkomponente

Eine Minimalkomponente eignet sich sehr gut dazu, in der Praxis zu erfahren, welche Entwurfszeit-Fähigkeiten selbst die einfachste von *TComponent* abgeleitete Klasse von ihrer Basisklasse erbt. Die wirklich minimale Komponente besteht zwar aus dem leeren Gerüst, wie es vom Komponenten-Experten erstellt wird, unsere erste Komponente soll jedoch wenigstens ein eigenes Property besitzen.

Der folgende Code der Unit *IntComp* wurde geschrieben, indem das vom Komponenten-Experten erstellte Gerüst um die beiden mit »(**)« markierten Zeilen erweitert wurde. :

```
unit Intcomp;

interface

uses
  SysUtils, Classes, QGraphics, QControls, QForms,

type
  TIntegerComponent = class(TComponent)
  private
```

```
    { Private-Deklarationen }
  protected
(**)    FValue: LongInt;
    { Protected-Deklarationen }
  public
    { Public-Deklarationen }
  published
    { Published-Deklarationen }
(**)    property Value: LongInt read FValue write FValue;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Zum Buch', [TIntegerComponent]);
end;

end.
```

TIntegerComponent soll dazu dienen, im Property *Value* eine Zahl zu speichern. Sie benötigt keine Methoden, um das Property zu lesen oder zu schreiben, sondern gibt dem Compiler mit den *read*- und *write*-Direktiven an, dass er Lese- und Schreibzugriffe auf das Property direkt an die Variable *FValue* weiterleiten soll.

An sich besteht also kein Unterschied zwischen der Variablen und dem Property. Um jedoch Werte einer Komponente für den Objektinspektor zugänglich zu machen, ist es immer notwendig, ein Property dafür zu deklarieren, selbst wenn sich dahinter nur eine Variable verbirgt.

TIntegerComponent in der Praxis

TIntegerComponent wird automatisch mit allen anderen Komponenten dieses Buchs über installiert, wie in Kapitel 8.2.2 beschrieben. Wie jede andere Komponente finden Sie sie jedoch nicht nur in der Komponentenpalette, sondern auch in der *Komponentenliste* (Ansichtsmenü).

Sie können mehrere *TIntegerComponent*-Exemplare in ein Formular einfügen und deren *Value*-Property zur Entwurfszeit ändern. Durch den Standard-Konstruktor von *TObject* wird dieses Property (bzw. die zugrunde liegende Variable *FValue*) vorher immer mit 0 initialisiert. Jedes Exemplar von *TIntegerComponent* wird zur Entwurfszeit bereits als Icon dargestellt und verfügt über die geerbten Properties *Name* und *Tag*. Auch das Speichern des neuen Properties in der Formulardatei läuft automatisch ab.

Zur Laufzeit lässt sich die (dann unsichtbare) Komponente kaum zu mehr verwenden als eine normale Integer-Variable.

8.2.3 Installation von Packages und Komponenten

Zur Installation von Komponenten haben Sie in Kylix zwei Punkte im Menü KOMPONENTE zur Auswahl:

- Mit PACKAGES INSTALLIEREN gelangen Sie in den Dialog, in dem Sie die schon installierten Packages aktivieren und deaktivieren können (siehe Abbildung 8.1). Dort können Sie auch neue Komponenten installieren, sofern diese schon als Packages vorliegen.
- Außerdem gibt es den Menüpunkt KOMPONENTE INSTALLIEREN.... Hier geht es darum, eine Unit auszuwählen, die in der Komponentenpalette aufgenommen werden soll. Da Sie jedoch nur Packages installieren können, müssen Sie neben dem Namen der zu installierenden Unit auch den Namen eines Packages angeben, in das die Unit aufgenommen werden soll (Abbildung 8.3). Sie haben hier die Wahl, ein neues Package zu erzeugen oder ein bestehendes Package zu erweitern, wobei Kylix Ihnen hierzu das zunächst einmal leere Package *user.dpk* aus seinem LIB-Verzeichnis vorschlägt. Auf diese Weise gelangen Sie in den Package-Editor, in dem es jedoch wahrscheinlich nichts zu editieren gibt, da Kylix Ihnen normalerweise bereits anbietet, alle erforderlichen Schritte wie Neukompilieren und Installieren selbst auszuführen.

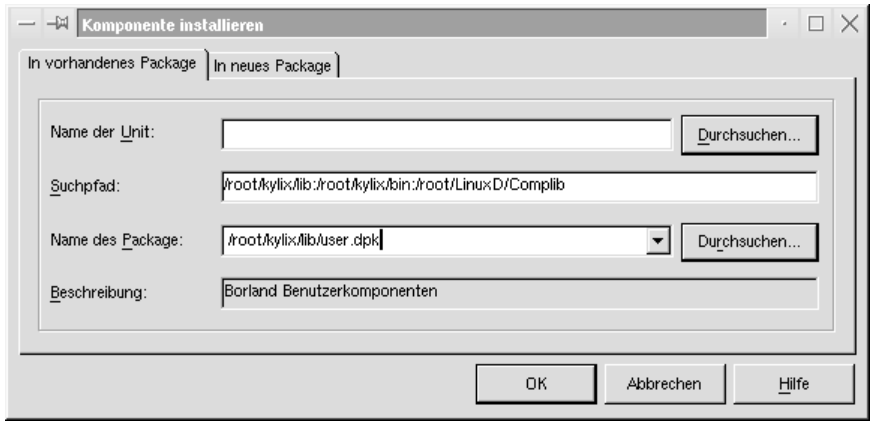


Abbildung 8.3: Installieren neuer Komponenten in ein Package

Der Package-Editor

Kylix verfügt über einen Package-Editor, in dem Sie alle zu einem Package gehörenden Daten verändern können, sofern Sie das Package selbst erzeugt haben oder zumindest eine DPK-Datei für das Package besitzen (Abbildung 8.4). Zu einem bestehenden Package können Sie diesen Editor entweder über den normalen *Datei-öffnen*-Dialog

oder im schon in Abbildung 8.1 gezeigten Package-Installationsdialog über den Schalter *Bearbeiten...* öffnen. Für ein neues Package finden Sie in DATEI | NEU auf der ersten Seite einen passenden Eintrag.

Das in Abbildung 8.4 gezeigte Package enthält die Komponenten zu diesem Buch. Jede Komponente wurde in einer eigenen Unit implementiert, die Sie in der Liste unter dem *Contains*-Knoten finden und die mit dem Schalter HINZUFÜGEN in das Package aufgenommen wurde. Einige Komponenten verwenden noch weitere Units, die ebenfalls in das Package übernommen wurden (z.B. *HList* für das interne *THistoryList*-Objekt der Komponente *THistoryCombo* und *ColorPalEdit* für den Property-Editor der Farbpalette). Auch wenn Sie eine vom Package verwendete Unit nicht explizit in die *Contains*-Liste eintragen, wird sie vom Compiler automatisch zum Package hinzugefügt; allerdings erhalten Sie dann eine Warnung, dass die Unit nicht explizit in das Package aufgenommen wurde.

Die *Contains*-Liste in Abbildung 8.4 enthält neben Units auch noch eine DCR-Datei. Diese enthält Bitmaps für die einzelnen Komponenten und wird in Kapitel 8.4.4 erläutert werden.

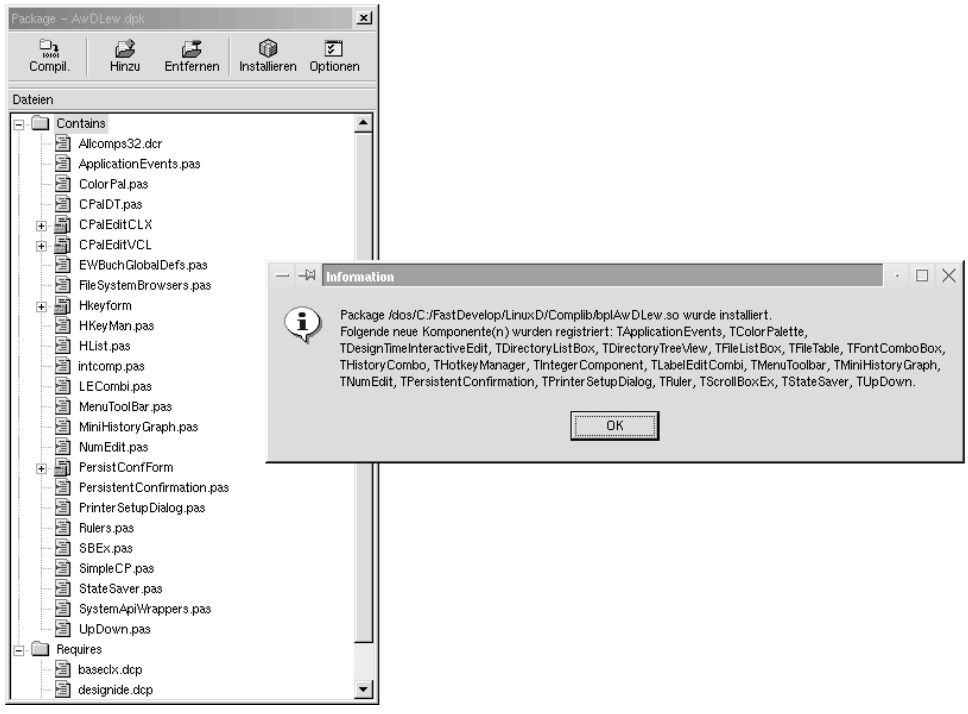


Abbildung 8.4: Der Package-Editor von Kylix beim Installieren der Komponenten dieses Buchs

Der Inhalt eines Packages

Bei einer DPK-Datei handelt es sich um den menschenlesbaren Quelltext eines Packages, der von Kylix in eine binäre BPL-Datei (Borland Package Library) übersetzt wird, die dann in die Kylix-IDE bzw. in Ihre Anwendung eingebunden wird. Für die Anwendung des Package-Editors kann es hilfreich sein, einen solchen Package-Quelltext zu kennen. Für das Package der Komponenten dieses Buchs sieht der Quelltext wie folgt aus (gekürzt):

```
package AwDLew;

{$R *.res}
{$R 'Allcomps.dcr'}
{$ALIGN ON}
{$ASSERTIONS ON}
...weitere Compiler-Anweisungen...
{$DESCRIPTION 'Komponenten des Kylix-Buchs von Elmar Warken'}
{$IMPLICITBUILD ON}

requires
  baseclx, visualclx;

contains
  Intcomp in 'Intcomp.pas',
  NumEdit in 'NumEdit.pas',
  Rulers in 'Rulers.pas',
  ColorPal in 'ColorPal.pas',
  ColorPalEdit in 'ColorPalEdit.pas' {PalEditForm},
  ColorPalDesignTime in 'ColorPalDesignTime.pas',
  ... Liste weiterer Units ...
  MenuToolBar in 'MenuToolBar.pas';
```

Jeder Bestandteil dieses Quelltextes lässt sich durch den Package-Editor festlegen:

- Die bereits erwähnten eingebundenen Units unter dem Knoten *Contains* bzw. im *contains*-Abschnitt des Package-Quelltexts.
- Auch zum Knoten *Requires* im Package-Editor gibt es im Package-Quelltext eine direkte Entsprechung. Dieser Abschnitt nennt alle Packages, die zum Betrieb des aktuellen Packages vorausgesetzt werden. Anwendungen, die dieses Package benutzen, müssen auch alle in der *requires*-Klausel aufgeführten Packages laden. Normalerweise benötigt jedes Package zumindest die Standard-Packages, die den Kern der CLX und den Grundvorrat an Komponenten bereitstellen.
- Die Compiler- und Linkeroptionen, die bei der Übersetzung der Units verwendet werden sollen, können Sie in einem Dialog einstellen, wenn Sie den Schalter *Optionen* im Package-Editor drücken. Die Seiten dieses Dialogs stimmen zum größten Teil mit dem Dialog der Projektoptionen überein.

- Über die üblichen Projektoptionen hinaus gehen die Einstellungen auf der ersten Seite dieses Dialogs. Hier genügt es meistens schon, wenn Sie eine Beschreibung für Ihr Package eintragen, für das Package der Buchkomponenten wurde jedenfalls nur diese Beschreibung geändert.

Die Aktionsschalter des Package-Editors

Eine sehr angenehme Einrichtung sind die Schalter zum Kompilieren und Installieren eines Packages:

- Mit dem Installieren-Schalter fügen Sie ein Package in die Komponentenpalette ein.
- Mit dem Kompilieren-Schalter übersetzen Sie ein Package nicht nur neu – falls es sich um ein Package handelt, das bereits in der Komponentenpalette installiert ist, installiert Kylix das neu kompilierte Package automatisch in der Komponentenpalette. Auf diese Weise können Sie Verbesserungen und Fehlerkorrekturen in Ihren Komponenten sehr schnell in die Palette übernehmen.

Sie können ein im Package-Editor angezeigtes Package wie ein »Nebenprojekt« ansehen, das Sie gleichzeitig mit einem »Hauptprojekt« (dem Projekt, das in der Titelzeile der IDE angezeigt wird) bearbeiten können. Mit START | START können Sie so z.B. jederzeit die Hauptprojekt-Anwendung starten und ohne das aktuelle Projekt zu schließen, können Sie auch das Package neu übersetzen, indem Sie im Package-Editor den Kompilieren-Schalter drücken.

Hinweis: Achten Sie beim Neu-Kompilieren eines Packages auf eventuelle Compiler-Fehlermeldungen im Editorfenster. Unter besonderen Umständen (wenn die IDE z.B. zweimal geöffnet ist und das neu kompilierte Package installiert ist) kann Kylix eine Package-Datei nicht neu erzeugen und außer einer versteckten Meldung im Editorfenster deutet nichts darauf hin, dass die Komponentenpalette noch nicht aktualisiert wurde.

Probleme mit fehlenden Komponenten

Sollte die Installation eines Packages aus irgendeinem Grund fehlschlagen, vergessen werden oder sollten Sie ein Package absichtlich deinstallieren, können Sie Formulare, die Komponenten aus diesem Package enthalten, nicht mehr vollständig laden.

Kylix zeigt Ihnen in diesen Fällen eine Meldung und lässt Sie wählen, ob das Laden des Formulars abgebrochen oder ob der Fehler ignoriert werden soll. Beim Ignorieren werden die nicht gefundenen Komponenten aus dem Formular entfernt, so dass Sie diese Option nur wählen sollten, wenn Sie diese Komponenten nicht mehr brauchen. Wenn Sie die Komponenten auf diese Weise löschen, bietet Kylix Ihnen beim nächsten Speichern der Formulardatei übrigens an, auch die überzähligen Komponentendeklarationen aus der Klassendeklaration des Formulars zu entfernen.

8.2.4 Eine sinnvolle Beispiel-Komponente

Unsere zweite Beispielkomponente wird eine visuelle Komponente sein, die eine sehr leichte, aber sinnvolle Aufgabe zu erfüllen hat. *TNumEdit* soll eine spezialisierte Version des Eingabefelds *TEdit* sein, die nur für die Eingabe von Integer-Zahlen gedacht ist. Wenn Sie eine Zahleneingabe aus einem *TEdit* oder *TMaskEdit* auslesen und im Programm auch als Zahl verwenden wollen, müssen Sie den String *T(Mask)Edit.Text* zuerst in einen Zahlenwert umwandeln, beispielsweise mit der Standardprozedur *StrToInt*. Diese Arbeit soll *TNumEdit* automatisch erledigen, und zwar unter Verwendung eines *LongInt*-Properties mit dem Namen *Number.*

Benutzung von TNumEdit

Wenn Sie *Number* einen Wert zuweisen, änden Sie auch den Textinhalt des Editierfelds. Umgekehrt erhalten Sie die Zahl, die dem Textinhalt entspricht, indem Sie *Number* auslesen:

```
{ Auslesen der Zahl }
  EingegebeneZahl := NumEdit1.Number;
{ Setzen von NumEdit.Text auf '10' }
  NumEdit1.Number := 10;
```

Wichtig ist dabei, dass *Number* immer dem geerbten Property *Text* entspricht. Wenn Sie also *Number* ändern, passt *TNumEdit* das Property *Text* automatisch an. Ändern Sie dagegen *Text*, so merkt das Property *Number* dies zwar nicht sofort, wenn Sie es aber abfragen, verwendet die Lesemethode für *Number* in jedem Fall den neuesten *Text*-Wert als Grundlage für die String-Konvertierung. Sie erhalten also mit *Number* immer den aktuellen Textwert. Die Properties *Number* und *Text* sind damit nach außen hin vollständig synchron.

Implementierung

TNumEdit ist von der Klasse *TEdit* abgeleitet und erweitert diese um das schon erläuterte Property sowie um die beiden Zugriffsmethoden, die die Konvertierung übernehmen:

```
unit Numedit;

interface

uses
  SysUtils, LibC, Classes, QGraphics, QControls,
  QForms, QStdCtrls,

type
  TNumEdit = class(TEdit)
  protected
```

```
    function AsInt: LongInt;
    procedure SetInt(i: LongInt);
  published
    property Number: LongInt read AsInt write SetInt;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Zum Buch', [TNumEdit]);
end;

function TNumEdit.AsInt: LongInt;
begin
  try
    Result := StrToInt(Text);
  except
    on EConvertError do
      Result := 0;
  end;
end;

procedure TNumEdit.SetInt(i: LongInt);
begin
  Text := IntToStr(i);
end;

end.
```

Während *AsInt* den String mit Hilfe der oben schon erwähnten Prozedur *StrToInt* zu einem *LongInt* macht, nutzt *SetInt* eine zweite Standardprozedur. Wenn sich der Text im Eingabefeld nicht in eine Zahl umwandeln lässt, erzeugt *StrToInt* eine Exception, die von *TNumEdit* jedoch wieder unter den Teppich gekehrt wird – *TNumEdit* zeigt dem Benutzer also keine Meldung über die Exception an, in der Annahme, dass dieser selbst sieht, dass es sich bei seiner Eingabe nicht um eine Zahl handelt. Stattdessen gibt *TNumEdit* bei einer falschen Eingabe den Wert 0 zurück (ohne diesen aber in den Text des Editierfelds zu schreiben).

Über eine kleine Erweiterung einer bestehenden Komponentenklasse können Sie also sehr schnell zu einer neuen Komponente gelangen, die voll funktionsfähig in die Kylix-IDE eingebunden ist. Von dieser Basis ausgehend können Sie immer weitere Änderungen vornehmen wie Änderung des Paletten-Bitmaps, Hinzufügen von Properties, Registrieren von Property-Editoren usw. Kapitel 8.5 wird sich mit etwas größeren Erweiterungen bestehender Komponenten befassen.

8.3 Die Schnittstelle zum Benutzer

Die Schnittstelle einer Komponente zum Benutzer (mit »Benutzer« ist auch hier der Entwickler gemeint, der die Komponente verwendet) besteht zur Entwurfszeit aus den Properties und den Events, die im Objektinspektor editiert werden können. Im Programmcode gehören auch die Methoden zu dieser Schnittstelle, sie brauchen hier jedoch nicht mehr beschrieben zu werden, da dies für alle Klassen inklusive Komponenten bereits in Kapitel 2.5 geschehen ist.

8.3.1 Properties

Die Benutzung der Properties und die allgemeine Funktionsweise mit *read* und *write*-Methoden ist in Kapitel 2.2.5 beschrieben, für diesen Abschnitt bleibt eine Zusammenfassung der eher bei Komponenten anzutreffenden Property-Themen. Zunächst haben Sie durch Variationen des Zugriffsschutzes und der *read*- und *write*-Methoden die Wahl zwischen den folgenden grundlegenden Property-Typen:

- *Read/Write-Properties* sind die »normalen« Properties, die sowohl gelesen als auch geschrieben werden können.
- *Read only-Properties*: Properties, die nur zum Lesen geeignet sind, verfügen zwar über eine *read*-Methode, enthalten aber keine *write*-Direktive in ihrer Deklaration. Viele Nur-Lesen-Properties geben feste Tatsachen an, die nicht ohne weiteres geändert werden können, z.B. die Liste der am Bildschirm verfügbaren Schriftarten (*TScreen.Fonts*). Das Lesen solcher Properties kann auch mit der Ausführung komplizierterer Funktionen verknüpft sein.
- *Write only-Properties*: Properties, die nicht gelesen werden können, sind relativ selten, sie dienen eher dazu, Aktionen auszulösen, als Werte zu speichern. In Kapitel 6.3.2 finden Sie mit den Properties von *TGraphicDocument* ein Beispiel dafür.
- *Runtime-Properties*: Dies sind Properties, die Sie nicht im Objektinspektor editieren können, sondern nur zur Laufzeit. Ein solches Property erhalten Sie, wenn Sie es nicht als *published*, sondern als *public* deklarieren. Dieser Property-Typ kann natürlich mit dem Nur-Lesen- oder Nur-Schreiben-Typ kombiniert werden.

berschreiben von Properties

Eine Eigenschaft der Properties, die sie von den Methoden und Daten eines Objekts unterscheidet, ist, dass Sie die Zugriffsbeschränkung in abgeleiteten Klassen ändern önnen. Die CLX-Klassen machen dies vielfach vor: So enthält beispielsweise die Klasse *TControl* eine Reihe von Properties, die als *protected* deklariert sind. Es steht den abgeleiteten Klassen frei, diese geschützten Properties zu veröffentlichen.

Dazu müssen Sie diese erneut in der Klassendeklaration erwähnen, allerdings nur mit Nennung ihres Namens. So veröffentlicht beispielsweise die Klasse *TColorPalette* das Property *DragMode* mit einer Zeile in ihrem *published*-Abschnitt:

```
published
  ...
  property DragMode;
```

Dieser Vorgang wird als *Überschreiben* des Properties bezeichnet. Sie können einem Property beim Überschreiben auch neue *read*- und *write*-Methoden zuweisen, indem Sie diese wie gewohnt hinter dem Property-Namen angeben. Auch die in Kapitel 8.4.3 beschriebenen weiteren Property-Direktiven *stored*, *default* und *nodefault* sind überschreibbar.

Hinweis: Das Zuweisen einer neuen *read*- oder *write*-Methode hat nicht dieselbe Wirkung wie das Überschreiben einer virtuellen Methode der Basisklasse, denn die Basisklasse ruft grundsätzlich immer die in ihrer eigenen Deklaration definierten Zugriffsmethoden auf. Diese können jedoch normale virtuelle Methoden sein, die Sie in den abgeleiteten Klassen überschreiben können (ohne das Property neu deklarieren zu müssen).

Property-Kategorien *R142*

Sofern Sie sich nicht darum kümmern, ordnet Kylix die von Ihnen definierten Properties automatisch in die im Objektinspektor genannte Kategorie »Verschiedene«. Properties, die eine neue Komponente von einer CLX-Komponente erbt, bleiben in der Kategorie, in der sie von der CLX eingeordnet wurden.

Davon abweichend können Sie in der seit Kapitel 8.2.2 bekannten *Register*-Prozedur weitere Kategorie-Zuweisungen vornehmen. Dabei können Sie auch selbst definierte Kategorien verwenden. Wie beides geht, zeigt der folgende Code-Auszug für die *TColorPalette*-Komponente, die in Kapitel 8.6 entwickelt wird.

Er dient dazu, die Properties der Farbpalette, welche die gewählten Farben für Stift, Pinsel und Schriftart angeben, in der Kategorie »Gewählte Palettenfarben« und sechs weitere Properties unter der Kategorie »TColorPalette-spezifisch« zusammenzufassen. Im Aufruf von *RegisterPropertiesInCategory* werden der Kategorie-Name, die Komponentenklasse und ein Array mit den Namen aller in die angegebene Kategorie fallenden Properties übergeben:

```
procedure Register;
begin
  RegisterPropertiesInCategory('Gewählte Farbpalettenfarben',
    TColorPalette, ['PenColor', 'BrushColor', 'FontColor']);
  RegisterPropertiesInCategory('TColorPalette-spezifisch',
    TColorPalette, ['Columns', 'Lines', 'SwapAlign', 'NoSelection',
    'RGBSteps', 'ColorDummy']);
```

Dies ist jedoch nur das Grundprinzip, von dem Sie abweichen können, indem Sie:

- statt einer selbst definierten Kategorie eine vordefinierte verwenden (in der Unit *DesignIntf* finden Sie unter der Überschrift *resourcestring* Namenskonstanten für mehr als zehn vordefinierte Kategorien, z. B. *sActionCategoryName* und *sLayoutCategoryName*);
- weitere Registrierungsmethoden verwenden, mit denen Sie Properties auch einzeln oder mit Platzhalterzeichen (*color** für alle Properties, deren Name mit »Color« beginnt) oder durch Angabe ihrer Typen (statt ihrer Namen) oder durch Kombinationen aus Komponentenklasse, Property-Typ und Property-Name registrieren können. Hierzu sei auf die Online-Hilfe zu den Funktionen *RegisterPropertyInCategory* und *RegisterPropertiesInCategory* verwiesen.

Wichtig ist vor allem noch, dass die Registrierung der Property-Kategorien eine reine Entwurfszeitfunktion ist und somit in einer Unit definiert werden sollte, die nur zur Entwurfszeit geladen wird. Die Kategorien der Farbpalette beispielsweise werden zusammen mit den Komponenten- und Property-Editoren (siehe Kapitel 8.6.5) in der Unit *ColorPalDesignTime* definiert.

Hinweis für Delphi-Kenner: Borland hat sich von der erst in Delphi 5 eingeführten komplizierten Art, die Kategorien zu registrieren, schnell wieder verabschiedet. Wo Sie bisher eine neue Klasse von *TPropertyCategory* ableiten mussten, nur um den Kategorie-Namen in einer Methode *Name* zurückzugeben, geben Sie den Namen nun einfach direkt an *RegisterPropertiesInCategory*.

8.3.2 Events

Die Verknüpfung von Komponentenereignissen mit Methoden des Formulars kommt ohne dynamische oder virtuelle Methoden aus und verwendet stattdessen Methodenzeiger und Properties, die zusammen eine Automatisierung von Routineaufgaben im Objektinspektor erlauben.

Definition des Events

Die Begriffe Event/Ereignis und Message/Nachricht/Botschaft sind im Zusammenhang mit ereignisorientierten Programmen meistens austauschbar und nicht auf einen speziellen Weg der Nachrichtenübermittlung festgelegt. In Kylix hat der Begriff Event« jedoch eine feste Bedeutung. So meint Event immer ein Ereignis, das von einer Komponente ausgelöst wird und bei dem die Komponente eine Methode aufruft, die in einem Methodenzeiger gespeichert ist. Meistens weist der Methodenzeiger auf die Methode eines Formulars. Sie können jedoch an der Automatik des Objektinspektors

vorbei zur Laufzeit auch Methoden anderer Klassen oder anderer Formulare mit den Event-Properties verknüpfen (wie beispielsweise bei der Bearbeitung eines Dokumentfenster-Menüpunkts durch das MDI-Hauptfenster, siehe Kapitel 6.7.4).

Deklaration eines Events

Die Deklaration eines Events besteht aus mehreren Teilen an verschiedenen Stellen der Komponenten-Unit:

- einem Methodenzeiger-Typ, der die Parameter der Ereignisbearbeitungsmethode festlegt (dieser Typ befindet sich außerhalb der Komponentendeklaration im normalen *type*-Block),
- einem Methodenzeiger innerhalb der Komponente, der den eben beschriebenen Typ hat
- und dem Property, das dem Objektinspektor Zugriff auf diesen Methodenzeiger verschafft

Für den Methodentyp machen wir einen kleinen Rückblick auf die Ereignisbearbeitungs-Methoden, die Sie aus dem Objektinspektor automatisch erstellen können, z. B.:

```
procedure TForm1.DateiOeffnenClick(Sender: TObject);
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

Die dazugehörigen Methodenzeiger-Typen sind in den Units der CLX wie folgt deklariert:

```
{ Unit Classes }
  TNotifyEvent = procedure(Sender: TObject) of object;
{ Unit Controls: }
  TMouseEvent = procedure(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer) of object;
```

Die Variablen dieser Typen befinden sich in den Komponenten, die Events dieser Art implementieren, so z. B. in den Menüpunktkomponenten für die *OnClick*-Ereignisse:

```
type
  TMenuItem = class(TComponent)
    ...
    FOnClick: TNotifyEvent;
    ...
```

Das zugehörige Property ist nur dazu da, die Variable *FOnClick* im Objektinspektor zugänglich zu machen. Es benötigt daher keine *read*- und *write*-Methoden, sondern leitet Anfragen direkt an *FOnClick* weiter:

```
property OnClick: TNotifyEvent read FOnClick write FOnClick;
```

Die anderen Regeln, die für Properties gelten, treffen auch bei Event-Properties zu. So önnen die Event-Properties als *protected* deklariert sein und erst später in abgeleiteten Klassen veröffentlicht werden. Weniger sinnvoll erscheinen allerdings nur lesbare oder nur beschreibbare Events.

Ein Beispiel-Event

Zweck eines neuen Events könnte es beispielsweise sein, das Formular über die Wahl einer Farbe aus einer Farbpalette zu benachrichtigen. Die in Kapitel 8.6 besprochene Komponente *TColorPalette* definiert für die Farben von Stift, Pinsel und Schrift jeweils ein Event des Typs *TColorChangeEvent*. Alle dafür notwendigen Deklarationen sind im folgenden Quelltextausschnitt zusammengefasst:

```
type
  TColorChangeEvent =
    procedure(Sender: TObject; Color: TColor) of object;
  TColorPalette = class(TCustomControl)
  private
    FOnPenColorChange,
    FOnBrushColorChange,
    FOnFontColorChange: TColorChangeEvent;
  published
    property OnPenColorChange: TColorChangeEvent
       read FOnPenColorChange write FOnPenColorChange;
    ...
```

Dabei gibt der erste Parameter wie üblich die *TColorPalette*-Komponente an, bei der das Ereignis entstanden ist, im zweiten Parameter befindet sich die neu gewählte Farbe. Der Parameter *Sender* könnte natürlich auch wie folgt deklariert sein, da das Event hier ausschließlich von der Klasse *TColorPalette* generiert wird:

```
TColorPalette = class; { Vorwärtsdeklaration }
TColorChangeEvent =
  procedure(Sender: TColorPalette; Color: TColor) of object;
```

onventionen für Events

Wenn Sie eigene Events für Komponenten schreiben, sollten Sie sich in der Namensgebung an die Konventionen der CLX halten. Demnach beginnen alle Events mit dem Präfix *On*, die zugehörigen Methodenzeigervariablen sind als *private* deklariert und stellen dem Namen des Properties ein *F* voran, z. B. *FOnClick*.

ür den Komponentenbenutzer ist noch ein zweiter Bezeichner sichtbar: der Name der Methoden, die mit dem Event verknüpft werden. Dieser Name wird jedoch von Kylix (bzw. dem Property-Editor für die Event-Properties) automatisch erzeugt und setzt sich aus dem Namen der Komponente, deren Methodenzeiger mit dieser Methode ver-

knüpft wird, und aus dem Namen des Events, allerdings ohne das *On* zusammen (z.B. *Button1Click*). Für Events des Formulars setzt Kylix statt des Formularnamens einfach *Form* ein, z.B. *FormMouseDown*.

Eine weitere Konvention für Events ist, dass sie immer als Prozedur deklariert sind. Wenn die verknüpfte Methode einen Wert zurückgeben soll, so sollte das Event dafür einen Variablenparameter bereitstellen. Ein solcher Variablenparameter kann vor dem Aufruf der Bearbeitungsmethode bereits auf einen sinnvollen Wert voreingestellt werden – anders als das Ergebnis einer Funktion. Wenn die Bearbeitungsmethode vergisst, den Ergebniswert zu setzen, bleibt der Variablenparameter bei seinem voreingestellten Wert, während ein nicht zugewiesenes Funktionsergebnis ein zufälliger Wert ist.

8.3.3 Events auslösen

Um das Event auszulösen, müssen Sie die mit dem Methodenzeiger verknüpfte Ereignisbearbeitungs-Methode aufrufen (die aus der Sicht der Komponente auch als *Event-Handler* bezeichnet wird). Dieser Aufruf unterscheidet sich ein wenig von einem normalen Prozeduraufruf. Da wahrscheinlich die meisten Events im Objektinspektor gar nicht mit Methoden verknüpft werden, bleiben die meisten Methodenzeiger bei ihrem voreingestellten Wert *nil*. Gegenüber dem normalen Prozeduraufruf kommt beim Aufruf von Events also immer eine Abfrage hinzu, ob das Event überhaupt verknüpft wurde.

Angenommen, in einer Komponente der Klasse *TColorPalette* aus dem obigen Beispiel ändert sich die Stiftfarbe. Die Komponente ruft dann das zugehörige Event wie folgt auf und gibt sich selbst im *Sender*-Parameter an:

```
if Assigned(FOnPenColorChange)
  then FOnPenColorChange(self, NewPenColor);
```

Leere Event-Handler

Die Tatsache, dass die mit dem Ereignis verknüpfte Methode leer sein kann, führt nicht nur dazu, dass Sie die Methode als Prozedur statt als Funktion deklarieren sollten, sondern auch dazu, dass Sie die obige *if*-Anweisung nicht wie folgt erweitern sollten:

```
if Assigned(FOnPenColorChange)
  then FOnPenColorChange(self, NewPenColor)
else Ersatzaktion;
```

Dies würde dazu führen, dass die Definition einer völlig sinnlosen Bearbeitungsmethode die *Ersatzaktion* verhindert. Eine sicherere Lösung wäre, die Ausführung der *Ersatzaktion* von einer um einen Variablen-Parameter erweiterten Bearbeitungsmethode abhängig zu machen; der Benutzer der Komponente müsste diesen Parameter dann bewusst auf *False* setzen, um die *Ersatzaktion* abzuschalten.

```
ErsatzErforderlich := True; { Voreinstellung }
if Assigned(FOnPenColorChange)
  then FOnPenColorChange(self, NewPenColor, ErsatzErforderlich);
if ErsatzErforderlich
  then Ersatzaktion;
```

8.4 Komponenten intern

Bei der Programmierung einer Komponente gibt es einige Unterschiede zum Programmieren eines Formulars, zu denen die folgenden gehören:

- Um Nachrichten selbst zu bearbeiten, kann sich eine Komponente nicht der Events bedienen, da diese für den Benutzer reserviert sind, sondern sie muss virtuelle bzw. dynamische Methoden überschreiben.
- Wenn eine Komponente andere Komponenten enthalten soll, müssen Sie diese selbst dynamisch erzeugen, da Komponenten nicht wie Formulare entworfen werden können.
- Auf Komponenten können weitere Aufgaben zukommen, wie das Teilnehmen am Speicherungsprozess des Formulars, die Bereitstellung von speziellen Property-Editoren usw.

8.4.1 Ereignisse in den Komponenten

Standen im letzten Kapitel die Ereignisse im Mittelpunkt, die die Komponente auslöst (die Events), geht es hier um die Ereignisse, die sie selbst bearbeitet.

Hinweis: *Event* meint hier ausschließlich Methodenzeiger-Properties, die zur Verknüpfung eines Komponentenereignisses mit der Methode eines beliebigen Objekts (meistens eines Formulars) dienen. Der Begriff *Ereignis* steht hier allgemein für Ereignisse aller Art. Wenn es später um die Ereignisbearbeitung in Qt geht, wird der Begriff *Event* auch für die allgemeinen Ereignisse von Qt verwendet.

Events und Ereignisse

So wie ein Formular auf äußere Ereignisse reagieren muss, können auch Komponenten Ereignisse empfangen. Einige Beispiele sind:

- Die *TScrollBox*-Klasse bearbeitet die Ereignisse, die von den beiden Bildlaufleisten erzeugt werden, und verschiebt als Reaktion darauf die Kindfenster der Scrollbox.
- Viele Komponenten, die nicht von Qt gezeichnet werden, reagieren auf einen Neuzeichnen-Auftrag, wie Formulare das beispielsweise in einer *OnPaint*-Bearbeitung tun können.

- Die Timer-Komponente bearbeitet die vom Betriebssystem gesendeten Timer-Botschaften.

Manche Komponenten geben diese Ereignisse an den Komponentenbenutzer weiter. So reagiert die *TPaintBox*-Komponente, wenn sie von Qt zum Neuzeichnen aufgefordert wird, indem sie die Methode aufruft, die der Komponentenbenutzer mit dem Ereignis *OnPaint* verknüpft hat. *TEdit* verarbeitet die empfangenen Tastendrücke und gibt sie an die mit *OnKeyDown*, *OnKeyPress* und *OnKeyUp* verbundenen Methoden weiter.

Andere Nachrichten werden nicht weitergegeben. So finden Sie im Objektinspektor für viele Komponenten kein *OnPaint*-Event, obwohl diese von Qt ein *QPaintEvent* gesendet bekommen. Normalerweise ist der einsehbare Grund dafür, dass sich die Komponente bereits selbst auf den Bildschirm malt und das dabei entstandene Kunstwerk nicht von einer Formularmethode verfälscht sehen will.

Virtuelle Methoden

Die Ereignisbearbeitung einer Komponente unterscheidet sich grundlegend von der Reaktion eines Formulars auf Events. Während Sie bei einem Formular den Objektinspektor zur Verfügung haben und (ohne es wissen zu müssen) mit Methodenzeigern arbeiten, findet der Übergang der Ereignisse von Qt zur Komponente über Hooks und *EventFilter*-Methoden statt, wie in Abbildung 3.2 gezeigt.

Die von der CLX vordefinierten Hook- und *EventFilter*-Methoden rufen nun für die meisten wichtigen Ereignisse jeweils eine virtuelle bzw. dynamische Methode auf, die Sie in abgeleiteten Komponentenklassen überschreiben können, um auf das Ereignis zu reagieren. Wenn eine solche dynamische Methode für ein von Qt gesendetes *QEvent*-Ereignis fehlt, können Sie auch die *EventFilter*-Methode überschreiben.

Die virtuellen Methoden der Klassen *TControl*, *TWidgetControl* und *TCustomControl*, die in Verbindung mit Ereignissen auftreten, sind in der folgenden Tabelle zusammengefasst:

Ereignis-Art	**virtuelle Methoden**	**ab Klasse**
Klicks	Click, DblClick	TControl
Drag&Drop	DragDrop, DragOver, DoStartDrag, DoEndDrag	TControl
Maus	MouseDown, MouseMove, MouseUp	TControl
Tastatur	KeyDown, KeyPress, KeyUp	TWidgetControl
Tastaturfokus	DoEnter, DoExit	TWidgetControl
Zeichnen	Paint	TCustomControl

Beispiele für die Verwendung von *DoExit* finden Sie in Kapitel 8.5.3, Beispiele für *Paint* und *MouseDown* in Kapitel 8.6.3.

Während die obige Tabelle nur die allgemeinen, schon in *T(Widget)Control* definierten, virtuellen Methoden nennt, finden Sie in den konkreten Komponentenklassen oft eine Reihe weiterer virtueller Methoden, mit denen Sie Ereignisse behandeln können, die normalerweise als Events im Formular bearbeitet werden. So sind das beispielsweise ür die *OnEdit-*, *OnChange-* und *OnCanChange*-Ereignisse von *TTreeView* die virtuellen Methoden *Edit*, *Change* und *CanChange*.

Es empfiehlt sich also in jedem Fall vor dem Ableiten einer neuen Klasse von einer bestehenden Klasse, sich in der Online-Hilfe, im CLX-Quelltext oder im Browser der IDE einen Überblick über die Erweiterungsmöglichkeiten zu verschaffen.

berschreiben von EventFilter

Da die Komponentenklassen der CLX jedoch nicht für alle Qt-Ereignisse spezielle virtuelle Methoden definieren, kann es manchmal erforderlich werden, die Ereignisbearbeitung an einem früheren Punkt der Ereignisweiterleitung anzusetzen, und zwar an den bereits erwähnten *EventFilter*-Methoden.

on den in diesem Buch entwickelten Komponenten benötigt nur *TColorPalette* eine eine solche *EventFilter*-Methode, die im Folgenden als Beispiel dienen soll. Es geht dabei um das Abfangen des Maus-Doppelklick-Ereignisses. Hierfür *gibt* es zwar die schon in der Tabelle genannte dynamische Methode *DblClick*, jedoch fehlen bei dieser Methode jegliche Parameter. Die Klasse *TColorPalette* möchte jedoch mitgeteilt bekommen, an welcher Position genau sich der Doppelklick ereignet hat (eine Alternativ- ösung ohne *EventFilter* wäre hier, die Position des letzten *OnMouseDown*-Ereignisses zu speichern und als Position des Doppelklicks anzunehmen).

TColorPalette überschreibt also die geerbte *EventFilter*-Methode, die *alle* Qt-Ereignisse ür *TColorPalette* zu sehen bekommt, bevor auf dem weiteren Weg der Ereignisbearbeitung die dynamischen Methoden aufgerufen werden (Abbildung 3.2). *EventFilter* bekommt zwei Qt-Objekte als Parameter überliefert: eine Referenz auf das Qt-Objekt, auf das sich das Ereignis bezieht (vergleichbar mit dem *Sender*-Parameter der Ereignismethoden in einem Formular) und eine Referenz auf das Ereignis selbst.

```
// uses Qt;
(* Der Typ des selbst definierten Events stammt aus QControls.pas:
  TMouseEvent = procedure(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer) of object; *)

TColorPalette = class(CustomControl)
  // CLX-Event, das beim Qt-Event MouseButtonDblClick
  // ausgelöst werden soll:
  property OnMouseDblClk : TMouseEvent read FOnMouseDblClk
```

```
                                        write FOnMouseDblClk;
    function EventFilter(Sender: QObjectH;
                         Event: QEventH): Boolean; override;
    ...
  end;
```

Während *Sender* meist ein Qt-Widget darstellt, ist *Event* ein Objekt einer von *QEvent* abgeleiteten Klasse. Um welche Klasse es sich genau handelt, lässt sich aus dem Typ des Events-Folgern, der in jedem Fall mit der Funktion *QEvent_type* ermittelt werden kann (diese Funktion dient als Brücke zu der in Object Pascal nicht direkt aufrufbaren C++-Methode *QEvent::type*).

Die Namen der verschiedenen Event-Typen finden Sie in *Qt.pas* aufgelistet, eine Dokumentation dazu und zu den entsprechenden speziellen, von *QEvent* abgeleiteten Event-Klassen finden Sie in der Qt-Dokumentation von Trolltech.

In unserem Beispiel ist nur das *MouseButtonDblClick*-Ereignis interessant. Wenn dieses Ereignis eintritt, löst *TColorPalette* selbst ein Event aus (welches im Objektinspektor mit einer Methode verknüpft werden kann) und ruft noch eine interne Verarbeitung des Doppelklicks in *HandlePaletteDoubleClick* auf, was hier nicht weiter von Bedeutung ist.

Wichtig ist hier vor allem, wie *EventFilter* auf die speziellen Daten des Qt-Events zugreifen kann. Hierfür müssen weitere Funktionen bemüht werden, in diesem Fall *QMouseEvent_x* und *QMouseEvent_y*, um die Position des Doppelklicks zu erfahren. Beide erwarten ein *QMouseEventH* als Parameter, wohingegen wir nur über ein *QEventH* verfügen. Wenn jedoch der Typ des Events *MouseButtonDblClick* ist, dann können wir davon ausgehen, dass das *QEventH* in Wirklichkeit ein *QMouseEventH* ist und wir es per Typumwandlung an *QMouseEvent_x/y* übergeben können:

```
function TColorPalette.EventFilter(Sender: QObjectH;
  Event: QEventH): Boolean;
begin
  Result := inherited EventFilter(Sender, Event);
  if (QEvent_type(Event)=QEventType_MouseButtonDblClick) then begin
    if Assigned(FOnMouseDblClk) then
      FOnMouseDblClk(self, mbLeft, [],
               QMouseEvent_x(QMouseEventH(Event)),
               QMouseEvent_y(QMouseEventH(Event)));
    HandlePaletteDoubleClick(QMouseEvent_x(QMouseEventH(Event)),
               QMouseEvent_y(QMouseEventH(Event)));
  end;
end;
```

Das Finden von Dokumentation zu einer Qt-Methode ist wie folgt möglich: Übersetzen Sie den Object-Pascal-Routinen-Name wie z.B. *QMouseEvent_x* in eine C++-Methode wie etwa *QMouseEvent::x*. Unter der zugehörigen Klasse – hier *QMouseEvent* – finden Sie dann in der Qt-Dokumentation alle Informationen. Die Object-Pascal-Deklarationen dieser Methoden sind in *Qt.pas* zu finden.

Weitere Erläuterungen zum Umgang mit den Umsetzung der Qt-Klassen und zur Klassenhierarchie« der Qt-Objekt-Handles gibt Kapitel 5.4.2 (*QMouseEventH* ist intern eine von *QEventH* abgeleitete Klasse).

Vorsicht: Wenn Sie vergessen, mit *inherited EventFilter* die geerbte Ereignisverarbeitung aufzurufen, kann das ein sehr ungewöhnliches Verhalten der Komponente zur Folge haben, das vielleicht erst später auffällt, wenn Sie gar nicht mehr an *EventFilter* denken. Der Autor überprüft jedenfalls, wenn eine seiner Komponenten wieder einmal ein ungewöhnliches Verhalten zeigt, das nicht durch den selbst geschriebenen Programm-Code erklärt werden kann, als Erstes eine etwaige *EventFilter*-Methode.

8.4.2 Komponenten in der Formulardatei

Dieses Kapitel gibt einen Überblick über die Speicher- und Ladeprozesse der Komponenten und wie Sie in diese eingreifen können. Es liefert damit unter anderem den Hintergrund zu Kapitel 8.4.3.

Speichern von Komponenten

Das Datei-Abbild einer Komponente besteht im Wesentlichen aus den Properties und aus den Unterkomponenten der Komponente, wobei die Klasse *TWriter* den größten Teil der Speicherung erledigt. Der gesamte Prozess des Speicherns besteht, wie der CLX-Quellcode zu erkennen gibt, aus einem komplizierten Wechselspiel zwischen *TComponent* und *TWriter*. Für die Entwicklung eigener Komponenten brauchen Sie jedoch die Details davon nicht zu kennen, daher verzichtet die folgende Zusammenfassung darauf, zu erläutern, welche Klasse der CLX die jeweiligen Schritte durchführt.

Während Sie beim Laden einer Komponente drei Gelegenheiten haben, Code Ihrer Komponentenklasse auszuführen, kommt beim Speichern der Komponente normalerweise nur die virtuelle Methode *DefineProperties* zum Überschreiben in Frage. In der folgenden groben Skizzierung der Komponentenspeicherung wird *DefineProperties* im dritten Schritt aufgerufen:

- Zuerst schreibt die CLX die Klasseninformationen in den Stream, damit auch polymorphe Komponenten unverändert wiederherstellbar sind.
- Dann speichert sie alle Properties, die gewisse Regeln erfüllen, zu denen wir in Kapitel 8.4.3 kommen.
- Sie ermöglicht daraufhin den abgeleiteten Komponentenklassen über die virtuelle Methode *DefineProperties* (definiert in *TPersistent*), weitere Daten zu speichern, die außerhalb der automatisch gespeicherten Properties liegen (siehe Kapitel 8.6.6 für ein Beispiel).

- Den Properties lässt sie eine Liste der in der Komponente enthaltenen Komponenten folgen (Property *Components*), von denen jedoch die Steuerelemente zunächst ausgeschlossen bleiben, denn diese werden in einer eigenen Liste gespeichert:
- *TWidgetControl*-Komponenten speichern schließlich auch die in ihnen enthaltenen Steuerelemente (Property *Controls*).

Da die Klasse *TWidgetControl* bereits in der Lage ist, ein komplettes Fenster mit all seinen Unterfenstern, Komponenten und Properties zu speichern, muss die Klasse *TForm* diesem Vorgang nichts mehr hinzufügen.

Laden von Komponenten

Das Laden der Komponente ist nicht einfach nur eine Umkehrung des Schreibens, sondern enthält einige zusätzliche Stationen, die Sie zur korrekten Initialisierung Ihrer Komponente verwenden können (Schritte 2, 4 und 6):

- Zum Beginn des Ladens ist noch gar keine Komponente vorhanden. Die CLX liest zuerst die Klasseninformation, die im ersten Schritt des Speicherns geschrieben wurde. Anhand dieser Daten sucht sie dann mit *FindClass* (siehe Kapitel 4.3) die zugehörige Klassenreferenz, mit deren Konstruktor sie eine neue Objektinstanz erstellt.
- Wenn Sie den virtuellen Konstruktor der Basisklasse überschreiben, erhalten Sie an dieser Stelle die erste Gelegenheit, in den Prozess einzugreifen. Sie können innerhalb des Konstruktors über das Property *ComponentState* sogar feststellen, ob die Komponente gerade aus einer Formulardatei gelesen wird oder ob es sich um einen normalen Konstruktoraufruf handelt.
- Erst nach dem Konstruktoraufruf kann die CLX die automatisch gespeicherten Properties laden.
- Es folgt Ihre zweite Möglichkeit, am Ladeprozess teilzunehmen: Nachdem diese Properties geladen sind, erhalten Sie in der Methode *DefineProperties* die Möglichkeit, weitere Daten zu lesen.
- Der CLX-Code setzt das Laden dann gemäß der Speicherreihenfolge fort und liest die Komponentenliste und eventuell die Liste der untergeordneten Fenster.
- Zum Abschluss gibt Ihnen die CLX noch einmal die Gelegenheit, eigene Initialisierungen vorzunehmen: Wenn alle Daten geladen sind, die Komponente also soweit wiederhergestellt ist, wie es der CLX möglich war, ruft sie die virtuelle Methode *Loaded* auf. Überschreiben Sie diese, um abschließende Initialisierungen vorzunehmen (ein Beispiel finden Sie in Kapitel 8.7.3).

8.4.3 Steuerung der Property-Speicherung

Bereits in der Deklaration eines Properties haben Sie Einfluss auf die Dateioperationen der CLX: Wenn Sie in ihr nur die nötigsten Angaben machen, speichert die CLX genau die Properties, die auch als *published* deklariert sind (wobei Array-Properties nicht *published* sein dürfen). Um von dieser Standardverhaltensweise abzuweichen, können Sie die Direktiven *stored* und *default* einsetzen sowie weitere Daten zur Laufzeit als Property »verkleiden«, um sie ebenfalls speichern zu können.

Default

Mit *default* geben Sie der CLX weitere Informationen, anhand derer diese selbst entscheiden kann, ob das Property gespeichert wird. Diese Direktive ist nur bei Properties erlaubt, die einen ordinalen oder einen Mengentyp haben. (Bei Array-Properties hat *default* eine andere Bedeutung, siehe Kapitel 2.2.5).

Hinter dem Wort *default* geben Sie einen konstanten Wert an, der dadurch als voreingestellter Wert definiert wird. Sie teilen der CLX dadurch mit, dass sie das Property nicht speichern muss, wenn es diesen Wert aufweist.

Zu einer solchen *default*-Angabe gehört immer eine passende Zuweisung im Konstruktor der Komponente. Mit der folgenden Deklaration verpflichtet sich *TForm* beispielsweise, das Property *Position* im Konstruktor mit dem Wert *poDesigned* vorzubelegen:

```
property Position: TPosition read FPosition
  write SetPosition default poDesigned;
```

Da beim Speichern eines Property auch viele Verwaltungsinformationen (darunter auch der Name des Properties) in die Datei geschrieben werden müssen, ein nicht gespeichertes Property aber überhaupt keine Spur in der Datei hinterlässt, kann man sich ausrechnen, dass durch die Vergabe von Default-Werten ein deutlicher Geschwindigkeitsgewinn erzielt wird: Oft werden nur wenige Voreinstellungen der Properties geändert, daher müssen auch nur wenige Properties geschrieben werden. Für die anderen Properties wird noch nicht einmal ein Flag in die Datei geschrieben, welches angibt, dass das Property nicht gespeichert wurde.

Object Pascal macht sich außerdem die Mühe, die Direktive *nodefault* zu definieren. Sie können diese verwenden, um das Fehlen eines *default*-Werts zu verdeutlichen oder beim Überschreiben eines Properties den geerbten *default*-Wert auszuschalten.

Stored

Mit der Direktive *stored* legen Sie fest, ob das Property grundsätzlich gespeichert werden soll (falls sein Wert nicht mit dem *Default*-Wert übereinstimmt):

```
public
  property Zahl: Integer; read Get write Put stored True;
```

Mit »stored True« können Sie erreichen, dass ein nicht veröffentlichtes Property gespeichert wird, während *stored False* sinnvoll ist, um veröffentlichte Properties von der Speicherung auszuschließen (dies könnte bei Properties der Fall sein, die zur Entwurfszeit nützliche Funktionen beinhalten, zur Laufzeit aber nicht gebraucht werden).

Die direkte Angabe einer booleschen Konstante ist jedoch nur eine Möglichkeit für eine *stored*-Direktive. Alternativ dazu können Sie auch ein Element der Klasse, das zu einem booleschen Ergebnis führt, angeben. Dieses Element kann entweder eine *Boolean*-Variable bzw. ein Property sein oder eine Funktion, die keine Parameter, aber ein *Boolean*-Ergebnis hat. In einer solchen Funktion können Sie noch ausgefeiltere Bedingungen zur Speicherung eines Werts stellen, als dies durch die Angabe eines *Default*-Wertes möglich ist.

Nicht zu vergessen ist, dass die CLX nach der *stored*-Angabe auch noch die *default*-Angabe auswertet. Wenn die Überprüfung des *stored*-Teils also grünes Licht für die Speicherung gibt, kann eine Übereinstimmung des Properties mit dem voreingestellten Wert die Speicherung immer noch verhindern.

Überschreiben von Stored und Default

Wenn Sie ein Property überschreiben, können Sie nicht nur die Lese- und Schreibmethoden, sondern auch die *stored*- und *default*-Werte ändern. Daher könnte es sogar sinnvoll sein, dass Sie ein Property mit »stored False«, aber trotzdem mit *Default*-Wert deklarieren. Wenn eine abgeleitete Klasse sich entschließt, das Property doch zu speichern, braucht diese Klasse nur die *Stored*-Direktive zu überschreiben; durch den geerbten *Default*-Wert ist trotzdem eine effektive Speicherung gewährleistet.

Selbst speichern

Wie der letzte Abschnitt erläutert hat, findet bereits eine differenzierte Abwägung statt, *ob* ein Property gespeichert werden soll – was einen großen Effektivitätsgewinn zur Folge hat. Da die CLX weder Array-Properties speichern kann, noch irgendwelche Daten, die nicht als Property vorliegen, müssen Sie ab und zu selbst für die Speicherung sorgen.

Bei Betrachtung der *read*- und *write*-Direktiven eines Properties läge es nahe, der Deklaration von Array-Properties einfach eine *fileread*- und eine *filewrite*-Direktive hinzuzufügen und so Methoden zum Speichern und Laden dieser Properties festzulegen. Borland führt diese Direktiven jedoch nicht in Object Pascal ein, sondern verlässt sich auf eine noch flexiblere Methode, die von der CLX bereitgestellt wird (und die auch die Daten, die nicht als Property vorliegen, berücksichtigt):

Sie können durch Überschreiben der *TPersistent*-Methode *DefineProperties* beliebige Datenblöcke definieren, die – quasi als Property getarnt – in eine Datei geschrieben bzw. aus dieser gelesen werden können. *DefineProperties* bezeichnet diese Datenblöcke nicht zur Verwirrung des Programmierers als Properties, sondern weil die Handhabung dieser selbst definierten Datenblöcke sehr ähnlich zur Handhabung von Properties ist. Allerdings gelten die mit *DefineProperties* definierten Properties ausschließlich beim Lesen und Schreiben der Datei als Properties.

Kapitel 8.6.6 erklärt anhand einer Beispielkomponente, wie eine solche *DefineProperties*-Methode im Detail funktioniert.

8.4.4 Die Schnittstelle zur Kylix-IDE

Um eine oder mehrere Komponenten in die Komponentenpalette einzubinden, genügt bereits ein Aufruf von *RegisterComponents*, den Sie in einer Prozedur namens *Register* vornehmen. Nach der Registrierung steht Ihre Komponente auf der gewünschten Seite der Palette und die veröffentlichten editierbaren Properties sind im Objektinspektor zugänglich. Dem Beispiel für einen *RegisterComponents*-Aufruf aus Kapitel 8.2.2 ist hier nichts mehr hinzuzufügen. Zusätzlich können Sie auch Standardaktionen für Ihre Komponenten registrieren, die bei der Verwendung Ihrer Komponenten leicht in die Menüs einer Anwendung eingebunden werden können. Ein Beispiel hierzu finden Sie in Kapitel 8.6.7. In Zusammenhang mit der Benutzerschnittstelle erwähnenswert sind schließlich noch die schon in Kapitel 8.3.1 besprochenen und besonders leicht zu definierenden Property-Kategorien.

Bitmaps

Die einfachste Aktion, die Sie zusätzlich zur Registrierung durchführen können, besteht im Zeichnen einer Bitmap für die Komponentenpalette. Benötigt wird eine 24x24 Pixel große Bitmap-Datei im BMP-Format, die allerdings in eine Ressourcen-Datei konvertiert werden muss. Üblicherweise erhält diese Datei noch die Endung `.dcr` ür »Delphi Component Ressource«. Leider steht unter Kylix kein Tool wie der Bildeditor von Delphi zur Verfügung, der direkt solche Ressourcen-Dateien erzeugen kann, allerdings können Sie Ressourcendateien, die Sie unter Windows mit Delphis Bildeditor oder einem Ressourcen-Compiler erzeugt haben, unter Kylix weiterverwenden.

Um die Verbindung zu Ihrer Komponente herzustellen, geben Sie der Bitmap in der Ressourcen-Datei den Namen der Komponente, umgeformt in Großbuchstaben. Um die Verbindung zu der Komponenten-Unit herzustellen, geben Sie der `dcr`-Datei denselben Namen wie der Unit und speichern sie im selben Verzeichnis. Schließlich nehmen Sie die `dcr`-Datei noch in die *Contains*-Liste des Package-Editors auf, indem Sie dort HINZUFÜGEN drücken und die Datei auswählen.

Editoren

Zur Implementierung von komplexeren Komponenten genügen oft weder die einfachen Datentypen, die Sie direkt im Objektinspektor editieren können, noch die komplexeren Typen, für die die CLX bereits Editoren in eigenen Fenstern zur Verfügung stellt (z.B. *TStringList*, *TBitmap*, *TMenuItem*). Daher gibt Ihnen Kylix die Möglichkeit, eigene Editoren sowohl für einzelne Properties als auch für ganze Komponenten zu registrieren.

Property-Editoren

Property-Editoren dienen dazu, für einen Property-Typ, der bisher nicht editierbar war, weil er nicht in einer der üblichen Formen dargestellt werden konnte, einen neuen Editor zur Verfügung zu stellen. Dieser Editor kann in einem eigenen Formular definiert sein, das als Dialog aufgerufen wird, oder lediglich die Darstellung des Textes im Objektinspektor beeinflussen. Sie können einen Editor auch für Properties schreiben, die schon über einen Editor verfügen, und damit den bisherigen Editor ersetzen. Property-Editoren müssen nicht für alle Properties eines Typs gelten, sondern können sich auch auf ein bestimmtes Property beschränken.

Um einen eigenen Property-Editor zu schreiben, leiten Sie eine neue Klasse von *TPropertyEditor* ab und installieren diesen mit der Prozedur *RegisterPropertyEditor*, die der Prozedur *RegisterComponents* ähnelt und wie diese in der *Register*-Prozedur der Komponenten-Unit aufgerufen wird. Ein konkretes Beispiel dazu gibt Kapitel 8.6.5. Nach erfolgreicher Registrierung verwendet Kylix diesen Editor für alle Properties, die den angegebenen Typ haben, oder für das speziell angegebene Property.

Komponenten-Editoren

Komponenten-Editoren werden zur Entwurfszeit entweder durch einen Doppelklick auf die Komponente oder durch die Auswahl eines eigenen Menüpunkts im Popup-Menü der Komponente aufgerufen. Ein Komponenten-Editor hat, ähnlich wie ein Property-Editor, nichts mit einem herkömmlichen Editor zu tun.

Nach der Definition der abstrakten Klasse *TComponentEditor* ist ein Komponenten-Editor eine Klasse, deren Hauptaufgabe darin besteht, verschiedene zusätzliche Befehle (so genannte Verben) für eine bestimmte Komponentenklasse zur Verfügung zu stellen und auszuführen. Beispielsweise stellt der Komponenten-Editor für *TTreeView*-Komponenten einen Eintrags- und einen Spalteneditor zur Verfügung.

Sie erstellen einen neuen Komponenten-Editor, indem Sie eine neue Klasse von *TComponentEditor* ableiten und die Routine *RegisterComponentEditor* aufrufen. Auch hierfür finden Sie in Kapitel 8.6.5 ein Beispiel.

omponenten zur Entwurfszeit

Eine der großen internen Neuerungen von Delphi und Kylix gegenüber früheren Pascal-Entwicklungsumgebungen ist die Tatsache, dass Teile des von Ihnen entwickelten Codes nahtlos in die IDE eingebunden werden können, indem Sie Ihren Code an die Komponentenschnittstelle anpassen. Tatsächlich könnte sogar der gesamte Code Ihrer Unit schon dann aktiv werden, wenn der Benutzer Ihrer Komponenten sich noch im Entwurfsmodus befindet.

Der einzige Grund, warum die Komponenten zur Entwurfszeit auf den ersten Blick etwas bewusstlos erscheinen, ist, dass der CLX-Code die allermeisten Ereignisse der IDE überlässt. Mauseingaben werden zum Verschieben der Komponente oder zum Aufruf des Popup-Menüs verwendet und Tastatureingaben aktivieren den Objektinspektor. Beide Ereignisse gelangen nicht mehr zu den von Ihnen geschriebenen Methoden.

rotzdem wird auch zur Entwurfszeit ständig ein Teil des von Ihnen geschriebenen Codes aufgerufen: Wenn Sie ein Property im Objektinspektor verändern, ruft Kylix die zugehörige Schreibmethode auf. Wenn Sie beispielsweise bei der Farbpalette aus Kapitel 8.6 das Property *Columns* ändern, ruft die Palette *Invalidate* auf, so dass die Entwurfszeit-Darstellung der Palette sofort angepasst wird.

Lagebestimmung

Damit die CLX die reservierten Nachrichten nicht auch zur Laufzeit des Programms herausfiltert, muss sie natürlich feststellen, ob die Komponente sich zurzeit im Formular-Designer oder in der freien Natur befindet. Dazu bedarf es keiner Geheimabsprache zwischen CLX und Kylix-IDE, sondern lediglich der Abfrage einer dokumentierten Schnittstelle. Wenn Sie das Verhalten Ihrer Komponente ebenfalls davon abhängig machen wollen, ob die Komponente sich noch im Formular-Designer befindet, können Sie dieselbe Schnittstelle benutzen.

Dazu gehört zuerst einmal das Property *TComponent.ComponentState*, das eine Menge von Flags enthält, zu denen auch *csDesigning* gehören kann. Ist das der Fall, befindet sich die Komponente mitsamt ihren Eltern, Besitzern und Kindern im Entwurfsprozess innerhalb der IDE. Sie können dann sogar Zugriff auf eine abstrakte Formular-Designer-Schnittstelle erhalten, welche in der Unit *Forms* deklariert ist und über die Sie in gewissem Umfang mit der IDE kommunizieren können.

ommunikation mit dem Designer

Wird das Formular in der Kylix-IDE bearbeitet, so weist sein Property *DesignerHook* auf eine Schnittstellenvariable des Typs *IDesignerHook*, über die Sie mit einem Designer-Objekt der IDE kommunizieren können. Diese Kommunikation ist natürlich der

Einschränkung unterworfen, dass Sie maximal die in der abstrakten Klasse definierten Methoden aufrufen können, und nicht alle Methoden, die der Designer tatsächlich beinhaltet.

IDesignerHook enthält beispielsweise die Methode *IsDesignEvent*, die den Designer fragt, ob er eine bestimmte Nachricht bearbeitet. Mit dieser Methode filtert die *TWidgetControl.MainEventFilter* zur Entwurfszeit die meisten Eingabenachrichten aus dem Nachrichtenfluss heraus.

Für die Programmierung einer Komponente kann besonders die Methode *Modified* wichtig sein (definiert im Interface *IDesignerNotify*). Sie teilt der IDE mit, dass sich die Komponente geändert hat, so dass beispielsweise der Objektinspektor Werte neu ausgeben kann (siehe Kapitel 8.6.6).

8.5 Erweiterung bestehender Komponenten

In diesem Kapitel wenden wir uns wieder der Praxis zu, und zwar der einfachsten Art, eigene Komponenten zu schreiben. Diese besteht darin, eine bereits existierende Komponente zu verändern oder zu erweitern. Besonders einfach ist die erstgenannte Möglichkeit, da es zum Verändern einer Komponente bereits genügt, Property-Definitionen zu überschreiben.

8.5.1 Verändern bestehender Komponenten

In diesem Abschnitt ist mit dem »Verändern« einer Komponente all das gemeint, was ohne das Hinzufügen von neuen Methoden, Variablen oder Properties geschehen kann. In diesem Sinne war die *TNumEdit*-Komponente aus Kapitel 8.2.4 schon zu groß, denn sie erweiterte ja die vordefinierte *TEdit*-Komponente um ein neues Property.

Häufig genutzter Ansatzpunkt für Veränderungen an bestehenden Komponenten sind die schon vorhandenen Properties. In einer abgeleiteten Komponentenklasse können Sie die folgenden Eigenschaften der geerbten Properties ändern:

- den Zugriffsschutz,
- mit der *default*-Direktive als »voreingestellt« deklarierte Werte,
- die mit der *stored*-Direktive vorgenommene Speicherungseinstellung sowie
- Lese- und Schreib-Methoden.

Die wohl einfachste Veränderung an Komponenten besteht darin, bisher nicht veröffentlichte Properties zu veröffentlichen. Damit der Benutzer nicht von einer extrem langen Eigenschaftsliste im Objektinspektor überrascht wird, lassen viele Komponenten der CLX einen großen Teil der Properties unveröffentlicht. Meistens geschieht das auch aus dem Grund, dass die Properties bei der jeweiligen Komponente keine sinn-

volle Funktion ausüben. Wie bereits in Kapitel 8.3.1 gezeigt, brauchen Sie in Ihrer Komponentenklasse ein geerbtes Property nur kurz in dem Schutzbereich (*public*, *published* usw.) zu nennen, in dem Sie es haben wollen.

ndern der Default-Werte

Wenn Sie mit den Vorbelegungen der Properties einer bestimmten Komponente im Objektinspektor unzufrieden sind, können Sie eine neue Komponente von dieser ableiten, um den Default-Wert zu überschreiben. Im Prinzip brauchen Sie den gewünschten Wert nur in einem neuen Konstruktor der Komponente in das Property zu schreiben. Wenn Sie z.B. eine Panel-Komponente ohne anfängliche Beschriftung wünschen, önnten Sie folgende Komponente definieren:

```
type
  TEmptyPanel = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TEmptyPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption := ' ';
end;
```

In diesem speziellen Fall muss der String mindestens ein Leerzeichen enthalten, weil die Beschriftung sonst beim Einzeichnen der Komponente in das Formular wieder auf den vorläufigen Namen des Panels gesetzt wird. Das obige Beispiel ist allerdings für neue Default-Werte unvollständig, weil es keine *Default*-Direktive verwendet. Bei Properties, die einen ordinalen Typ, einen Zeigertyp oder einen Mengentyp haben, können Sie den Default-Wert wie folgt deklarieren (zusätzlich zu einer Zuweisung im Konstruktor):

```
property ModalResult default mrOK;
```

Die Folgen einer solchen Deklaration sind in Kapitel 8.4.3 erläutert. Beispiele für Default-Werte gibt die Klasse *TColorPalette* in Kapitel 8.6; auch *TScrollBoxEx* in Kapitel 8.5.2 enthält einen *Default*-Wert.

Hinweis: Komponenten, in denen nur die Property-Voreinstellungen geändert sind, können Sie, ohne eine neue Komponentenklasse zu definieren, auch als Komponentenvorlage definieren, indem Sie die Komponente einmal in einem Formular vorbereiten und dann mit KOMPONENTE | KOMPONENTENVORLAGE ERZEUGEN in die Komponentenpalette einfügen.

Überschreiben virtueller Methoden (Schriftwahl-Kombobox) R52

Das folgende Beispiel einer Kombobox, die eine Liste der im System installierten Schriftarten enthält und dabei die Schriftarten gleichzeitig anzeigt (Abbildung 8.5), erweitert die vorhandene Komponente *TComboBox* ebenfalls ohne neue Properties und Methoden. Sie überschreibt zwei geerbte virtuelle Methoden und den Konstruktor:

```
type
  TFontComboBox = class(TComboBox)
  protected
    procedure Loaded; override;
  public
    constructor Create(AOwner: TComponent); override;
    function DrawItem(Index: Integer; Rect: TRect;
      State: TOwnerDrawState): Boolean; override;
  end;
```



Abbildung 8.5: Eine TFontComboBox-Komponente zur Laufzeit

Im Konstruktor wird die Kombobox durch Ändern des *Style*-Properties zu einer besitzergezeichneten Komponente gemacht:

```
constructor TFontComboBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Style := csOwnerDrawFixed; // besitzergezeichnet mit fester Höhe
end;
```

Nachdem alle automatischen Initialisierungen der Komponente während des Ladevorgangs abgeschlossen sind, kopiert sie in der *Loaded*-Methode die Liste der Schriftarten aus dem globalen *TScreen*-Objekt in ihr eigenes *Items*-Property:

```
procedure TFontComboBox.Loaded;
begin
  inherited Loaded;
  Items.Assign(Forms.Screen.Fonts);
end;
```

Nun muss sie noch dafür sorgen, dass jeder Eintrag auch mit seiner eigenen Schriftart ausgegeben wird. Der Name für die Schriftart des *i*-ten Eintrags ist also in *Items[i]*

selbst gespeichert. Die allgemeinen Grundlagen besitzergezeichneter Komponenten sind in Kapitel 4.4.4 beschrieben, ein Beispiel für eine besitzergezeichnete ListBox finden Sie in Kapitel 4.5.4. Im Stil *csOwnerDrawFixed* erhalten alle Listeneinträge die Höhe, die der Benutzer im Property *ItemHeight* angibt.

Um auf die Nachricht zum Zeichnen eines Eintrags zu reagieren, verwendet die Komponente nicht das *OnDrawItem*-Ereignis, sondern überschreibt eine diesem Ereignis entsprechende virtuelle Methode:

```
function TFontComboBox.DrawItem(Index: Integer; Rect: TRect;
  State: TOwnerDrawState): Boolean;
begin
  Canvas.FillRect(Rect);
  Canvas.Font.Name := Items[Index];
  Canvas.TextOut(Rect.Left, Rect.Top, Items[Index]);
  Result := False;
end;
```

Da dies nur ein kleines Beispiel sein soll, belässt es die Methode beim Setzen des Schriftart-Namens. Andere Schriftattribute, vor allem die Größe, werden im Vertrauen auf eine sinnvolle Einstellung unverändert übernommen. So müssen Sie, um eine größere Schrift zu erhalten, im Objektinspektor zwei Properties von *TFontComboBox* manuell einstellen: *ItemHeight* und *Font.Height*.

8.5.2 Erweiterung von TScrollBox

Erstes Beispiel für eine kleine Ergänzung der Schnittstelle einer bestehenden Komponente soll die Klasse *TScrollBoxEx* sein, die im TreeDesigner benötigt wird:

- Das neue Ereignis *OnScroll* wird in Kapitel 6.5.4 dazu verwendet, während des fließenden Scrollvorgangs auch die Anzeige der Lineale des TreeDesigners zu aktualisieren (Methode *TDocumentForm.ScrollBoxScroll*).
- Die von *TScrollBox* geerbten, aber erst in *TScrollBoxEx* veröffentlichten Tastaturereignisse werden in Kapitel 6.8.1 genutzt, um Tastatureingaben in die Zeichenfläche des TreeDesigners abzufangen.

Das folgende Listing zeigt die komplette Komponenten-Unit für *TScrollBoxEx* mit den als *published* redeklarierten *OnKey...*-Ereignissen und einer Ereignisbehandlungs-Methode, die das Bewegen der Scrollbars abfangen soll. Da diese Scrollbar-Bewegung weder mit einer virtuelle Methode verbunden ist noch als Qt-Event an die ScrollBox gesendet wird, muss sich *TScrollboxEx* eines Tricks bedienen: Indirekt wird bei jeder Bewegung der Scrollbars ein Zeichen-Ereignis ausgelöst, in dessen Folge wiederum die virtuelle Methode *Painting* aufgerufen wird, die von *TScrollboxEx* überschrieben wird. Da das Zeichnen jedoch auch ohne Bewegung der Scrollbars vorkommen kann, vergleicht die neue Methode noch die aktuelle Scrollbar-Position mit der letzten

bekannten Position. Wenn hier ein Unterschied festgestellt wird, interpretiert die Methode das so, dass die Scrollbars bewegt wurden, und erzeugt das von der Komponente neu eingeführte *OnScroll*-Ereignis.

```
unit ScrollboxEx;

interface

uses
  SysUtils, LibC, Classes, Types, Qt, QGraphics, QControls, QForms,
  QDialogs, QStdCtrls, EWBuchGlobalDefs;

type
  TScrollNotifyEvent = procedure (Sender : TObject; Code : word;
                                  Horz : Boolean) of object;

  TScrollBoxEx = class(TScrollBox)
  private
    FOnScroll : TScrollNotifyEvent;
  published
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnScroll: TScrollNotifyEvent read FOnScroll write FOnScroll;
  private
    LastContentsX, LastContentsY: Integer;
  published
    procedure Painting(Sender: QObjectH; EventRegion: QRegionH); override;
  end;

procedure Register;

implementation

procedure TScrollboxEx.Painting(Sender: QObjectH; EventRegion: QRegionH);
var
  cx, cy: Integer;
begin
  inherited;
  cx := self.HorzScrollBar.ScrollPos;
  cy := self.VertScrollBar.ScrollPos;
  if Assigned(FOnScroll) then begin
    if cx <> LastContentsX then
      FOnScroll(self, 0, True);
    if cy <> LastContentsY then
      FOnScroll(self, 0, False);
  end;
  LastContentsX := cx;
  LastContentsY := cy;
end;
```

```
procedure Register;
begin
  RegisterComponents(Zielseite, [TScrollBoxEx]);
end;

end.
```

Hinweis: Die Methode *Painting* ist nicht der richtige Ort für Grafikausgabe und sollte daher nicht mit *Paint* verwechselt werden. In *Paint* können Sie problemlos Grafik ausgeben, da die CLX vor dem Aufruf von *Paint* alle notwendigen Vorbereitungen trifft. In diesem Beispiel kann nur *Painting* verwendet werden, weil *TScrollbox* gar keine *Paint*-Methode besitzt. Da *TScrollboxEx* aber auch keine Grafik selbst zeichnen will, ist hier auch die *Painting*-Methode vollkommen ausreichend.

8.5.3 Die automatische History-Kombobox

Die Kylix-IDE macht regen Gebrauch von so genannten History-Listen, das sind Editierfelder, die sich die zuletzt gemachten Eingaben merken und in einer aufklappbaren Liste zur Wahl stellen, so dass Sie denselben Text nicht mehrmals eingeben müssen. Besonders hilfreich sind History-Listen, wenn sie zwischen zwei Sitzungen nicht verloren gehen (Kylix speichert sie in seiner Konfigurations-Datei). Beispiele für häufig verwendete History-Listen sind die des *Suchen*- und die des *Auswerten/Ändern*-Dialogs.

Um eine History-Liste selbst zu verwalten, müssen Sie zur Programmlaufzeit alle Eingaben, die der Benutzer in das Eingabefeld macht, zur Liste der Eingabestrings hinzu- ügen, dabei doppelte Strings entfernen und die Liste wieder verkürzen, wenn sie zu lang wird. Besonders wichtig ist auch, dass Sie solche Listen zwischen zwei Programmläufen speichern.

THistoryCombo *R65*

Dies alles erledigt die Komponente *THistoryCombo* automatisch für Sie: Sie müssen die Komponente nur in ein Formular einfügen und in den Properties *IniFileName* und *IniSection* festlegen, in welcher INI-Datei und unter welcher Überschrift die Liste gespeichert werden soll, und schon erledigt *THistoryCombo* die gesamte Verwaltungsarbeit automatisch.

on der Basisklasse *TComboBox* erbt *THistoryCombo* das Aussehen auf dem Bildschirm, die Bedienungsweise und das wesentliche Property *Items*, das den Inhalt der aufklappbaren Liste als *TStrings*-Objekt zur Verfügung stellt. Wir beginnen diesmal mit der vollständigen Klassendeklaration von *THistoryCombo*:

```
type
  THistoryCombo = class(TComboBox)
  { Combobox, die beim Laden aus einem Stream automatisch
    einen in der Registry oder einer INI-Datei gespeicherten
    Zustand wiederherstellt und die Liste bei Programmende
    wieder speichert. }
  private
    property Items;
  protected
    FRegistryPath: string;
    FIniFileName: string;
    FIniSection: string;
    FHistoryList: THistoryList;

    constructor Create(AOwner: TComponent); override;
     { in Destroy wird die Liste gespeichert: }
    destructor Destroy; override;
    procedure DoExit; override; { hier wird die Liste aktualisiert }
    procedure SetIniData(index: Integer; str: string);
{ Laden von HistoryList in die eigenen Items: }
    procedure UpdateComboList;
  public
     { Verwaltung der Stringliste (automatische Längenbeschränkung etc): }
    property HistoryList: THistoryList read FHistoryList;
    procedure UpdateList;
  published
    { Für Windows: Pfad der Speicherung in der Registrierung: }
    property RegistryPath: string index 0 read FRegistryPath
                                          write SetIniData;
    { Für Linux: Name der zu verwendenden INI-Datei: }
    property IniFileName: string index 1 read FIniFileName
                                          write SetIniData;
    { Name des zu verwendenden Registry-/INI-Datei-Abschnitts: }
    property IniSection: string index 2 read FIniSection
                                          write SetIniData;
  end;
```

Das Property *RegistryPath* sowie die zugrunde liegende Variable *FRegistryPath* werden nur in der Windows-Version der Komponente verwendet, welche die History-Liste in der Windows-Registry speichert.

THistoryList

Nun geht es darum, wie die Liste der vergangenen Strings verwaltet wird. *THistoryCombo* verwendet dazu nicht die geerbte *Items*-Liste, sondern ein Objekt der Klasse *THistoryList* aus den Kapiteln 4.1.2 und 4.2.2, die ebenfalls in der Unit *HList* definiert ist. *THistoryList* ist von *TStringList* abgeleitet und erweitert diese Klasse um das Management der INI-Dateien sowie um die Methode *AddString*, die einen String genau so in die

Liste einfügt, wie das für History-Listen erforderlich ist (als ersten Eintrag, unter Beibehaltung der Maximallänge der Liste und unter Beseitigung doppelter Einträge).

Der wichtigste Vorteil davon, für die History-Listenverwaltung eine eigene Klasse zu definieren, liegt darin, dass sich diese Klasse auch zu anderen Zwecken außerhalb von *THistoryCombo* verwenden lässt. So zeigt beispielsweise Kapitel 4.6.2, wie Sie *THistoryList* zur Buchführung über die zuletzt geladenen Dateien verwenden können (*THistoryList* kann die Einträge nämlich automatisch an ein Menü anhängen.)

Anlässe zum Verlängern der History-Liste

Die History-Kombobox soll die History-Liste völlig selbstständig verwalten, das heißt, dass Sie die neuen Einträge nicht von außen durch einen Methodenaufruf (beispielsweise vom Formular aus) hinzufügen müssen. Da jeder neue Eintrag in ihrem eigenen *xt*-Property steht, bereitet es der Kombobox keine Schwierigkeit, die neuen Einträge ausfindig zu machen.

Problematisch ist lediglich, den richtigen Zeitpunkt, genauer gesagt, das passende Event zur Erweiterung der Liste zu finden. Die üblichen Events wie *OnKeyDown* und *OnChange* sind nicht der passende Anlass für die Verlängerung der Liste, denn schließlich will der Benutzer nicht für jedes eingegebene Zeichen einen eigenen Eintrag in der Liste vorfinden. Zur Lösung dieses Problems sind mehrere Ansätze denkbar, die Unit *HList* löst die Aufgabe wie folgt:

Sie fügt den *Text* bei jedem *OnExit*-Ereignis in die History-Liste ein. *OnExit* tritt jedes Mal auf, wenn die Liste den Fokus verliert, wenn der Benutzer also z.B. auf einen normalen Schalter drückt oder zum nächsten Eingabefeld wechselt. Da *TWidgetControl* für das *OnExit*-Event die virtuelle Methode *DoExit* bereitstellt, überschreibt *THistoryCombo* diese:

```
procedure THistoryCombo.UpdateComboList;
begin
  { Die Kombobox-Liste mit der THistoryList neu aufbauen: }
  Items := FHistoryList;
end;

procedure THistoryCombo.UpdateList;
begin
  { Einfügen des Textes in die privat verwaltete THistoryList: }
  FHistoryList.AddString(Text);
  { Füllen der Kombobox-Liste mit FHistoryList: }
  UpdateComboList;
end;

procedure THistoryCombo.DoExit;
{ Fokus-Wechsel -> Eintrag speichern. }
begin
```

```
    inherited DoExit;
    UpdateList;
  end;
```

Wenn Sie *THistoryCombo* in der Praxis ausprobieren, werden Sie wahrscheinlich feststellen, dass es sich damit gut arbeiten lässt. Trotzdem ist das alleinige Abfangen von *OnExit* nicht immer die optimale Lösung: Unter bestimmten Voraussetzungen kann der Benutzer den aktuellen Eintrag auch für irgendeine Aktion verwenden, ohne dass die Kombobox den Fokus verliert, so z.B. wenn er per ↵ den Standardschalter der Dialogbox betätigt oder wenn er einen SpeedButton anklickt, der ja den Tastaturfokus nicht erhalten kann.

Automatisches Konfigurations-Management

THistoryCombo enthält die oben beschriebenen Properties, um die Speicherposition der Liste in einer INI-Datei festzulegen. Wenn Sie diese im Objektinspektor setzen, brauchen Sie sich nicht mehr weiter um die Funktion und die Speicherung der History-Liste zu kümmern.

Bei der Implementierung dieser Automatik kommt es für *THistoryCombo* wieder auf die passenden Ereignisse an, bei denen die INI-Datei gelesen und geschrieben werden soll, wobei als »Ereignisse« in erster Linie wieder virtuelle Methoden in Betracht kommen.

Das Lesen der INI-Datei soll bei Programmstart stattfinden bzw. dann, wenn die Kombobox zu existieren beginnt. Im Konstruktor ist es jedoch noch zu früh dafür: Die Properties sind zu diesem Zeitpunkt noch nicht mit den im Objektinspektor eingestellten Werten geladen. Ein möglicher Zeitpunkt ist die virtuelle Methode *Loaded* (siehe Kapitel 8.4.2), die dann aufgerufen wird, wenn alle Properties aus der Formulardatei geladen wurden.

In diesem Fall genügt es jedoch, die History-Liste zu laden, wenn die Properties gesetzt werden; wir benötigen also eine Property-Schreibmethode, die die Liste beim Setzen der Properties aus der Datei lädt. Wie Sie der obigen Klassendeklaration entnehmen können, übernimmt eine einzige Methode, *SetIniData*, das Schreiben aller drei Properties. Über den Parameter *index* und den jedem Property zugewiesenen Index kann sie zwischen den drei Properties unterscheiden.

Innerhalb der Methode wird zwischen den Delphi-Versionen unterschieden: Nur in den Versionen für 32-Bit-Windows verwendet die Methode das Property *FRegistryPath*, da auch die Methode *THistoryList.LoadFromIni* aus Kapitel 4.2.2 nur dann die Registry verwendet. Außerdem stellt die Methode sicher, dass die INI-Datei nicht jedes Mal wenn eines der Properties gesetzt wird, sondern nur einmal geladen wird. Dies geschieht dann, wenn *RegistryPath* und *IniSection* (unter Windows) bzw. *IniFileName* und *IniSection* (unter Linux) mit Werten gesetzt wurden:

```
procedure THistoryCombo.SetIniData(index: Integer; str: string);
begin
  case index of
    0: FRegistryPath := str;
    1: FIniFileName := str;
    2: FIniSection := str;
  end;
  { Automatisches INI-Datei-Lesen, sobald alle notwendigen
    Angaben gemacht sind: }
{$ifdef CompileForRegistry}
  if (FRegistryPath <> '') and (FIniSection <> '') then begin
    FHistoryList.LoadFromIni(FRegistryPath, FIniSection);
    UpdateComboList;
  end;
{$else}
  if (FIniFileName <> '') and (FIniSection <> '') then begin
    FHistoryList.LoadFromIni(FIniFileName, FIniSection);
    UpdateComboList;
  end;
{$endif}
end;
```

Einfach ist es auch, ein »Ereignis« für das Speichern der Liste zu finden: Hier bietet sich der Destruktor an, der quasi als Allerletzter das sinkende Schiff der freigegebenen Komponente verlässt (auch beim Destruktor handelt es sich um eine virtuelle Methode, die mit der *override*-Direktive überschrieben werden muss). Er ruft die *THistoryList*-Methode *SaveToIni* auf, gibt die History-Liste frei und ruft natürlich den geerbten Konstruktor auf:

```
destructor THistoryCombo.Destroy;
begin
  { Automatische Speicherung, wenn die notwendigen
    Angaben gemacht sind: }
  UpdateFile;
  { da FHistoryList nicht in der Komponentenliste ist,
    muss sie manuell freigegeben werden: }
  FHistoryList.Free;
  inherited Destroy;
end;

procedure THistoryCombo.UpdateFile;
begin
{$ifdef CompileForRegistry}
  if (FRegistryPath<>'') and (FIniSection<>'') then begin
    FHistoryList.UseRegistry:=True;
    FHistoryList.SaveToIni(FRegistryPath, FIniSection);
  end else
{$else}
  if (FIniFileName<>'') and (FIniSection<>'') then begin
    FHistoryList.SaveToIni(FIniFileName, FIniSection);
```

```
  end;
{$endif}
end;
```

Der Konstruktor

Das Einzige, was jetzt noch fehlt, ist ein Konstruktor, der das *FHistoryList*-Objekt der Komponente, deren Freigabe sich bereits im letzten Listingabschnitt befindet, per Konstruktor-Aufruf initialisiert. Properties mit Default-Werten sind nicht vorhanden, so dass dies die einzige neue Aufgabe für den Konstruktor bleibt:

```
constructor THistoryCombo.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FHistoryList := THistoryList.Create;
end;
```

Testen der Komponente

Wie schon erwähnt, genügt es, eine *THistoryCombo*-Komponente in das Formular einzufügen und die Properties für die Registry zu setzen, um die Komponente in Betrieb zu nehmen. Das Testprogramm *HlTest* (Abbildung 8.6) enthält außer der History-Liste einen Schalter, dem Sie zur Laufzeit mit [⇆] den Tastaturfokus verleihen können. Die History-Liste wird dann immer aktualisiert, wenn Sie sie mit [⇆] verlassen, und außerdem natürlich, wenn das Programm beendet wird (denn auch dann tritt ein *OnExit*-Ereignis auf). Unter Windows speichert *HlTest* die History-Liste im Pfad *Software\HistList* der Registry, unter Linux verwendet es die Datei *Hist.ini*.

Hinweis: Die History-Liste auf der CD ist um ein weiteres Detail erweitert: Wenn Sie die Erweiterung der Liste manuell vornehmen wollen, z.B. falls der Benutzer per [↵] einen vordefinierten Schalter drückt, können Sie die oben gezeigte Methode *UpdateList* manuell aufrufen (beispielsweise in der *OnClick*-Methode des Schalters). Damit *UpdateList* dann nicht von *DoExit* noch ein zweites Mal aufgerufen wird, setzen Sie das oben nicht gezeigte *THistoryCombo*-Property *ManualUpdate* auf *True* (dieses Property steht auch im Objektinspektor zur Verfügung).

8.5.4 Zusammenfassen mehrerer Komponenten

Als letztes kleines Beispiel für das Verändern bestehender Komponenten dient die Kombination zweier existierender Komponenten zu einer Komponente: die häufig vorkommende Kombination aus *TLabel* und *TEdit*.

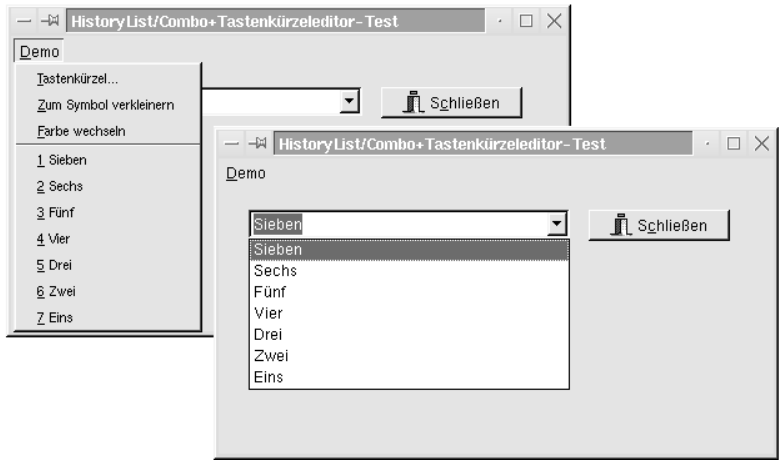


Abbildung 8.6: Das THistoryCombo-Testprogramm

Label und Edit als TCustomControl *R141*

Der Benutzer soll zur Entwurfszeit beide über einen gemeinsamen Rahmen verschieben und in der Größe ändern können. Um diesen Rahmen zur Verfügung zu stellen, muss die neue Klasse *TLabelEditCombi* selbst ebenfalls ein Steuerelement sein. Sie wird daher von der vordefinierten *TCustomControl* abgeleitet, die von sich aus nach außen unsichtbar ist, im Entwurfsmodus aber an der Markierung des besagten Rahmens erkannt werden kann (siehe Abbildung 8.7).

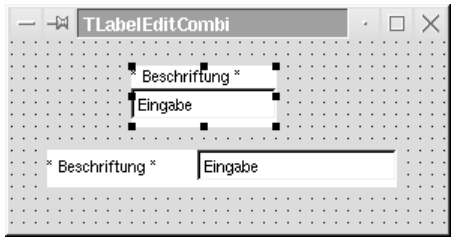


Abbildung 8.7: Die weiße Hintergrundfarbe der beiden TLabelEditCombi-Komponenten soll die von diesen eingenommene Fläche hervorheben; die in ihnen enthaltenen Einzelkomponenten lassen sich im Entwurfsmodus nicht selektieren.

Ein Nachteil gegenüber der in Kapitel 3.7 behandelten Frame-Technik ist, dass sich die beiden in *TLabelEditCombi* enthaltenen Komponenten nicht visuell zusammenstellen lassen wie beim Entwurf eines Formulars. Dafür sind die beiden Komponenten später

im Formular von außen nicht direkt ansprechbar und bleiben so voll unter der Kontrolle der neuen Komponente (je nach Einsatzzweck bedeutet diese Kapselung entweder einen Vorteil oder einen Nachteil).

Das folgende Listing zeigt das Grundgerüst der neuen Komponente. Es besteht aus den intern verwendeten Instanzen von *TLabel* und *TEdit*, aus einem überschriebenen Konstruktor und aus den zwei Properties *Text* und *Caption*, über die die beiden wichtigsten Properties von Beschriftungs- und Editierfeld nach außen hin zugänglich gemacht werden:

```
TLabelEditCombi = class(TCustomControl)
private
  FLabel: TLabel;
  FEdit: TEdit;
  procedure SetCaption(const Value: String);
  procedure SetText(const Value: String);
  function GetCaption: String;
  function GetText: String;
protected
  constructor Create(AOwner: TComponent); override;
published
  property Text: String read GetText write SetText;
  property Caption: String read GetCaption write SetCaption;
end;
```

Der Konstruktor hat die Aufgabe, die beiden »Unterkomponenten« dynamisch zu erzeugen und ihre Properties einzustellen:

```
constructor TLabelEditCombi.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);

  FLabel := TLabel.Create(self);
  with FLabel do begin
    Parent := self;
    AutoSize := True;
    Visible := True;
    Caption := '* Beschriftung *';
    SetBounds(0, 0, 100, Height);
  end;

  FEdit := TEdit.Create(Self);
  with FEdit do begin
    Parent := self;
    Visible := True;
    SetBounds(110, 0, 200, Height);
  end;
end;
```

In den Property-Zugriffsmethoden werden sowohl Lese- als auch Schreibzugriffe einfach auf die entsprechenden Properties der Unterkomponenten weitergeleitet, zum Beispiel auf das *Caption*-Property:

```
function TLabelEditCombi.GetCaption: String;
begin
  Result := FLabel.Caption;
end;

procedure TLabelEditCombi.SetCaption(const Value: String);
begin
  FLabel.Caption := Value;
end;
```

Entsprechend funktioniert dies für beliebig viele weitere Properties. Im vorliegenden Fall ist sicher am wichtigsten, dass Position und Größe der Unterkomponenten angepasst werden können. *TLabelEditCombi* deklariert zu diesem Zweck keine weiteren Properties, sondern überschreibt eine Methode, die bei jeder Größenänderung (der *TLabelEditCombi*-Komponente) aufgerufen wird. Da das *TLabel* wegen der *AutoSize*-Einstellung selbst für seine Größe sorgt, muss *TLabelEditCombi* nur noch für die Positionierung des Editierfelds sorgen. Diese findet in Abhängigkeit von der Größe des Labels statt. Nimmt das Label zu viel von der Breite der Kombi-Komponente weg, werden die beiden Komponenten untereinander positioniert.

```
procedure TLabelEditCombi.Resize;
begin
  inherited;
  ReAlign;
end;

procedure TLabelEditCombi.ReAlign;
begin
  if (FEdit = nil) or (FLabel = nil) then exit;
  if (FAlignment = leVertical) or
     ((FAlignment = leAuto) and (FLabel.Width > Width - 10)) then
  begin
    FLabel.Top := 0;
    FEdit.SetBounds(0, FLabel.Height + 2, Width, FEdit.Height)
  end else begin
    FEdit.SetBounds(FLabel.Width, 0, Width - FLabel.Width, FEdit.Height);
    FLabel.Top := (FEdit.Height-FLabel.Height) div 2;
  end;
end;
```

Die Position der Label-Komponente wird niemals geändert, da sie mit der Koordinate (0, 0) relativ zur oberen Ecke der Kombi-Komponente immer einigermaßen verwendbar ist. Diese automatische Positionierung könnte natürlich noch erheblich verbessert und flexibler gestaltet werden, aber die obige Version dürfte das Prinzip der Kombination zweier Komponenten zu einer Komponente genügend verdeutlichen.

8.6 Entwicklung neuer Steuerelemente

Dieses Kapitel beschreibt mit der Farbpalette die umfangreichste der in diesem Buch behandelten Komponenten. *TColorPalette* ist ein interaktives Steuerelement, das auf Mausaktionen reagieren muss und Events im Objektinspektor zur Verfügung stellt. Im Unterschied zur History-Kombobox aus dem letzten Kapitel erbt die Farbpalette jedoch dieses interaktive Verhalten und das äußere Erscheinungsbild nicht von einer Basisklasse, sondern muss es selbst implementieren.

Anhand der Farbpalette zeigt dieses Kapitel auch,

- wie Sie Property- und Komponenteneditoren schreiben,
- wie Sie Daten speichern, die nicht in Properties vorliegen, und
- wie Sie Popup-Menüs fest in Komponenten verankern und dem Benutzer gleichzeitig erlauben, sie zu erweitern.

8.6.1 Eine Testumgebung aufbauen

Die Farbpalette gehört zu den Komponenten, die so umfangreich sind, dass Sie sie wahrscheinlich nicht an einem Stück entwickeln wollen, sondern sie schon bei der Entwicklung hin und wieder testen wollen. Wenn Sie eine erste Version der Komponente bereits in der IDE installiert haben, können Sie gleichzeitig eine Testanwendung und das Package mit der Komponente in der IDE bearbeiten und die Komponente über den Package-Editor neu übersetzen lassen, wenn Sie ihren Quelltext verändert haben.

Testen durch dynamische Erzeugung *R135*

Sie können eine Komponente jedoch auch schon testen, bevor Sie sie in der Komponentenpalette installieren, indem Sie die Unit der Komponente in Ihr Projekt einbinden und ein Exemplar der Komponente dynamisch zur Laufzeit anlegen.

So könnte die *OnCreate*-Methode eines solchen Testformulars beispielsweise wie folgt aussehen:

```
uses
  ..., ColorPal { muss von Hand eingebunden werden }
...

procedure TForm1.FormCreate(Sender: TObject);
begin
  with TColorPalette.Create(self) do begin
    Parent := self;
    { Properties können noch nicht im Objektinspektor
      eingestellt werden: }
    Align := alClient;
```

```
      Columns := 4;
      Lines := 4;
    end;
  end;
```

Diese Methode funktioniert schon dann, wenn die Komponentenklasse noch aus dem Rumpf besteht, der vom »Komponenten-Experten« erstellten wurde. Mit einem solchen Testprogramm lassen sich die bei Kylix gewohnten schnellen Zykluszeiten von estlauf zu Testlauf beibehalten.

Erst später, wenn zum Laufzeitverhalten der Komponente noch Funktionen hinzukommen, die schon zur Entwurfszeit funktionieren sollen, muss die Komponente in der Kylix-IDE installiert werden.

ersionsunterschiede zwischen Laufzeit und Entwurfzeit

Wie schon erwähnt können Sie bei einem geöffneten Package-Editor neue Versionen der Komponenten schnell in die Komponentenpalette installieren. Bis zu dieser manuellen Aktualisierung der Komponentenpalette verwendet Kylix im Formular-Editor weiter die alte Version einer Komponente – im Gegensatz zu einer Anwendung, bei der die Komponenten-Unit in die ausführbare Datei eingebunden wird (also bei einer Anwendung, die nicht für die Benutzung von Laufzeit-Packages kompiliert wird). In einer solchen Anwendung wird die neue Komponentenversion verwendet, sobald Sie die Anwendung neu übersetzen (und dies geschieht beim nächsten Start der Anwendung aus der IDE automatisch, sofern der Pfad der Komponenten-Unit im Suchpfad des Projekts enthalten ist).

Während daher ein Testprogramm, das keine Laufzeit-Packages verwendet, immer die aktuelle Komponentenversion verwendet, enthält die Komponentenpalette der Kylix-IDE ohne manuelle Aktualisierung noch die alte Version. Solange Sie in der Komponente keine speziellen Dinge wie das Schreiben und Lesen von Daten aus der Formulardatei ändern, macht es nichts, wenn Sie Kylix mit einer leicht veralteten Komponentenversion arbeiten lassen und das Package nur ab und zu neu übersetzen. Jedoch können unterschiedliche Versionen einer Komponente zur Laufzeit und zur Entwurfszeit auch zu einer erheblichen Verwirrung des Entwicklers führen.

8.6.2 Die Schnittstelle der neuen Farbpalette

Der abgeschlossene Charakter einer Komponente bringt es mit sich, dass die Schnittstelle der Komponente – sowohl zum Komponentenbenutzer als auch zum Anwender schon zu Beginn einigermaßen klar umrissen sein sollte. Da die Farbpaletten-Komponente bereits auf CD-ROM vorliegt, können wir hier gleich mit der endgültigen Schnittstellen-Definition beginnen.

TColorPalette-Referenz

Das folgende Listing enthält die vollständige Klassendeklaration von *TColorPalette*, deren Elemente im weiteren Verlauf des Kapitels nach und nach zur Sprache kommen:

```
const
  MaxColors = 256; { mit 256 Farben verbraucht die Tabelle 1 kByte }

type
  TColorMark = (cmPen, cmBrush, cmFont);
  TColumns = 1..255;
  TRGBSteps = 1..5;

(** Methodenzeigertypen für die Events **)
  TColorChangeEvent =
    procedure(Sender: TObject; Color: TColor) of object;
  TPopupExpandEvent =
    procedure (Sender: TObject; var ExpandPoints: TPopupMenu)
       of object;
  TColorDefinitionEvent =
    procedure(Sender: TObject; Index: Integer;
              var ColorValue: TColor) of object;

(** Die Komponentenklasse **)
  TColorPalette = class(TCustomControl)
  private
  (* Events *)
    FOnPenColorChange,
    FOnBrushColorChange,
    FOnFontColorChange: TColorChangeEvent;
    FOnDefineColor: TColorDefinitionEvent;
    FOnExpandPopup: TPopupExpandEvent;
    FOnMouseDblClk: TMouseEvent;
  (* Variablen für Properties *)
    FColors: array[0..MaxColors-1] of TColor;
    SelColors: array[TColorMark] of Integer; { Indizes in FColors }
    FColumns,   { Zahl der im Fenster angezeigten Spalten }
    FLines: TColumns;  { Zahl der Zeilen }
    FSwapAlign: Boolean;   { dreht Spalten/Zeilen um}
    FNoSelection: Boolean; { schaltet die Selektierungsfunktion
                             der Maus aus  }
    FRGBSteps: TRGBSteps;
  (* eine Hilfsvariable *)
    RightClicked: Integer;
  (* Variable für die Entwurfszeit *)
    FSaveColors: Boolean;
  (* eine "Unterkomponente" *)
    ColorDialog: TColorDialog;
  private
  (* private Hilfsroutinen *)
    procedure InitColors;
```

```
  function GetColorOn(x, y: Integer): TColor;
  procedure CalcSides(var XSeite, YSeite: Integer);
  procedure DoSwapAlign(var X, Y: Integer);
  procedure InvalidateCell(i: Integer);
(* Stream-Methoden *)
  procedure ReadColors(Reader: TReader); virtual;
  procedure WriteColors(Writer: TWriter); virtual;
(* Property-Zugriffsmethoden *)
  function GetSelColor(Index: Integer): TColor;
  procedure SetSelColor(Index: Integer; NewValue: TColor);
  procedure SetSwapAlignFlag(Value: Boolean);
  procedure SetColumnNumber(Number: TColumns);
  procedure SetLineNumber(Number: TColumns);
  procedure SetRGBSteps(Val: TRGBSteps);
  function DummyFunc: Boolean;
public
  function GetColor(i: Byte): TColor;
  procedure SetColor(i: Byte; val: TColor);
(* öffentliche Hilfsroutinen *)
  function PosToCell(x,y: Integer): Integer;
(* überschriebene virtuelle Methoden *)
  constructor Create(AOwner: TComponent); override;
  procedure Paint; override;
  procedure MouseDown(Button: TMouseButton;
                      Shift: TShiftState; X, Y: Integer); override;
  procedure ChangeColorClick(Sender: TObject);
  procedure InitColorsClick(Sender: TObject);
  procedure DefineProperties(Filer: TFiler); override;
(* Bearbeitung des Doppelklicks *)
  function EventFilter(Sender: QObjectH; Event: QEventH): Boolean;
             override;
(* geerbtes PopupMenu-Property darf nicht veröffentlicht sein: *)
  property PopupMenu stored False;
(* ein unveröffentlichtes Property *)
  property Colors[i: Byte]: TColor read GetColor write SetColor;
published
(* Properties *)
  property SwapAlign: Boolean
     read FSwapAlign write SetSwapAlignFlag default False;
  property Columns: TColumns
     read FColumns write SetColumnNumber default 25;
  property Lines: TColumns
     read FLines write SetLineNumber default 5;
  property NoSelection: Boolean
     read FNoSelection write FNoSelection default False;
  property RGBSteps: TRGBSteps
     read FRGBSteps write SetRGBSteps default 5;
  property SaveColors: Boolean
     read FSaveColors write FSaveColors stored False;

  property PenColor: TColor index 0
```

```
      read GetSelColor write SetSelColor;
    property BrushColor: TColor index 1
      read GetSelColor write SetSelColor;
    property FontColor: TColor index 2
      read GetSelColor write SetSelColor;
    { ermöglicht den Zugriff auf den Property-Editor
      im Objektinspektor: }
    property ColorDummy: Boolean read DummyFunc stored False;

  (* Events *)
    { Benutzer setzt Farbmarke cmPen/cmBrush/cmFont neu: }
    property OnPenColorChange: TColorChangeEvent
      read FOnPenColorChange write FOnPenColorChange;
    property OnBrushColorChange: TColorChangeEvent
      read FOnBrushColorChange write FOnBrushColorChange;
    property OnFontColorChange: TColorChangeEvent
      read FOnFontColorChange write FOnFontColorChange;
    { Benutzer ändert einen Farbwert}
    property OnColorEdit: TColorEditEvent
      read FOnColorEdit write FOnColorEdit;
    { zu den folgenden beiden Events siehe zugehöriges Kapitel: }
    property OnDefineColor: TColorDefinitionEvent
      read FOnDefineColor write FOnDefineColor;
    property OnExpandPopup: TPopupExpandEvent
      read FOnExpandPopup write FOnExpandPopup;
    property OnMouseDblClk: TMouseEvent
      read FOnMouseDblClk write FOnMouseDblClk;

    (* zusätzliche Veröffentlichung von geerbten
      und geschützten Properties *)
    property Align;
    property OnEndDrag;
    property OnDragOver;
    property OnDragDrop;
    property DragMode;
    property OnMouseDown;
    property OnMouseUp;
    property OnMouseMove;
  end;
```

Bedienung der Komponente

Das Ziel der Farbpalette, Objekte mit Farben zu versehen, können Sie auf zwei verschiedene Weisen erreichen:

- Per Drag&Drop: Das Ziehen einer Farbe von der Palette auf ein grafisches Objekt liegt in der Verantwortung des Formulars, das die *TColorPalette*-Komponente benutzt. *TColorPalette* stellt zu diesem Zweck nur die von der CLX geerbten Drag&Drop-Events zur Verfügung. Kapitel 6.8.3 zeigt die Implementation dieses Drag&Drop im TreeDesigner.

- Per Markierung von Farben in der Palette. Drei dieser Farbmarkierungen sind vorgesehen, und zwar eine Vordergrund-, eine Hintergrund- und eine Schriftfarbe (anwählbar mit linker Maustaste, rechter Maustaste, sowie ⇧+linker Maustaste). Wofür die gewählten Farben tatsächlich verwendet werden, liegt am Entwickler, der die Komponente benutzt. Im TreeDesigner werden sie beispielsweise dazu verwendet, die Farben der neu eingezeichneten und der markierten Objekte einzustellen.

Die Unit *ColorPal* definiert die drei Farbmarken in einem Aufzählungstyp, dessen Bezeichner sich im Namen an den Zeichenwerkzeugen der *TCanvas*-Klasse orientieren:

```
TColorMark = (cmPen, cmBrush, cmFont);
```

Damit eine Farbe gleichzeitig über alle drei Markierungen verfügen kann und alle drei Marken gleichermaßen lesbar bleiben, schreibt *TColorPalette* nicht die Kürzel »HG« und »VG« auf die gewählte Farbe (wie *TColorGrid*), sondern zeichnet eine Linie für die Stiftfarbe, ein leeres Rechteck für die Füllfarbe und ein »T« für die Schriftfarbe.

Die Properties Lines, Columns und die Farben

Die wichtigsten Properties von *TColorPalette* sind *Lines* und *Columns*. Sie zeigen an, aus wie vielen Zeilen und Spalten die Palette aufgebaut sein soll.

Intern organisiert *TColorPalette* die Farben im Array *FColors*, das bis zu 256 *TColor*-Werte aufnehmen kann. Dabei enthält das Array an den Indizes 0 bis *Columns-1* die Farben der ersten Zeile, gefolgt von den Farben der zweiten Zeile usw. Die Farben des Arrays werden also von links nach rechts am Bildschirm ausgegeben.

Die automatische Initialisierung der Farben können Sie durch das Property *RGBSteps* steuern. Dieses zeigt an, aus wie vielen verschiedenen Blau-, Grün- und Rot-Werten die Farben kombiniert werden. Wählen Sie mit *RGBSteps*-Werten zwischen 2 und 5, ob die Farbpalette 8, 27, 64 oder 125 verschiedene Farbwerte enthalten soll. Bei jeder Änderung von *RGBSteps* initialisiert die Palette sämtliche Farben neu, Sie können diese Funktion also gut zur Entwurfszeit ausprobieren. Zur Berechnung der Farben siehe den Quelltext zu *TColorPalette.InitColors* auf der CD.

SwapAlign

Mit Hilfe des Flags *SwapAlign* können Sie die beschriebene Richtung der Farbanordnung umdrehen. Ist *SwapAlign* eingeschaltet, so bilden die ersten Felder von *FColors* nicht die erste Zeile, sondern die erste Spalte der am Bildschirm gezeichneten Palette. Darüber hinaus werden Spalten und Zeilen vertauscht, die Farben der ersten echten Spalte erhalten also die Indizes *0..Columns-1* (wobei hier das Property *Columns* durch die Vertauschung hier die Zeilen meint).

Für die Praxis bedeutet das, dass Sie die Palette mit *SwapAlign* umdrehen können und dass dabei eine vollständige Zeile zu einer vollständigen Spalte wird (anstatt dass die Zeile quasi mit einem neuen »Zeilenumbruch« auf mehrere Spalten zerstückelt wird, wie es der Fall wäre, wenn Sie einfach die Werte von *Columns* und *Lines* vertauschen).

Im Quelltext beschränkt sich der Einfluss des Properties *SwapAlign* auf die Methode *DoSwapAlign*. Diese wird von verschiedenen der in diesem Kapitel noch besprochenen Methoden zur möglichen Vertauschung von Zeilen- und Spaltenangaben verwendet, daher sei sie hier kurz aufgeführt:

```
procedure TColorPalette.DoSwapAlign(var X, Y: Integer);
var  Swap: Integer;
begin
  if SwapAlign then begin
    Swap:=X;   X:=Y;   Y:=Swap;
  end
end;
```

Die ausgewählten Farben

Das zweite Array-Property von *TColorPalette* ist *SelColors*. Es speichert die Farbindizes der Farben, die gerade ausgewählt sind. Um den *TColor*-Farbwert für die Stiftfarbe zu erhalten, benötigt die Palette also den folgenden Ausdruck:

```
FColors[SelColors[cmPen]];
```

Für den Benutzer der Komponente sollte natürlich eine einfachere Möglichkeit bestehen, die gewünschten Farbwerte zu erhalten. Zu diesem Zweck definiert *TColorPalette* die Properties *BrushColor*, *FontColor* und *PenColor*. Sie liefern direkt die *TColor*-Werte für die drei Farbmarken zurück.

Alle drei Properties sind auch beschreibbar. Das Schreiben übernimmt die Methode *SetSelColor*, die noch mehr zu tun hat als die Methode *GetSelColor*, die nur den oben gezeigten Ausdruck berechnen muss. Im einfachen Fall erhält *SetSelColor* einen *TColor*-Wert als Parameter, der sich bereits unter den bis zu 256 aktuellen Farben von *FColors* befindet. Dann muss *SetSelColor* lediglich den Index *SelColors* auf die entsprechende Position von *FColors* setzen. Ist der Farbparameter noch nicht im Array *FColors*, ersetzt die Methode einfach die gerade gewählte Farbe durch die neue Farbe.

GetSelColor und *SetSelColor* sind weitere Beispiele für Methoden, die mehrere über einen Index unterschiedene Properties gleichzeitig behandeln, wobei sie den Property-Index als zusätzlichen Parameter erwarten. *GetSelColor* beispielsweise wandelt diesen Index in einen *TColorMark*-Wert um, um damit im Array *SelColors* den Index der gewünschten Farbe zu finden:

```
function TColorPalette.GetSelColor(Index: Integer): TColor;
begin
  Result := FColors[SelColors[TColorMark(Index)]];
end;
```

Mausfunktion

TColorPalette unterstützt neben dem Markieren von Farben noch zwei weitere Mausaktionen: Sie können mit ⇧ und der rechten Maustaste ein Popup-Menü aufrufen, mit dem Sie die Farben ändern und zurücksetzen können. Mit einem Doppelklick auf eine Farbe können Sie diese direkt editieren (für das Editieren ist der Standardfarbendialog zuständig).

Allerdings wird beim Doppelklick auch die Pinselfarbe gesetzt, denn der erste Teil des Doppelklicks erzeugt ein *MouseDown*-Ereignis. Außerdem wäre es wünschenswert, mit der rechten Maustaste das Popup-Menü direkt aufrufen zu können. Zu diesem Zweck definiert *TColorPalette* als vorletztes Property *NoSelection*; es ist hauptsächlich ür den Paletteneditor in Kapitel 8.6.5 vorgesehen. Standardmäßig ist es auf *False* eingestellt, was bedeutet, dass Sie mit der Maus die drei verschiedenen Farbmarken setzen können. Wenn Sie dieses Property auf *True* setzen, werden die Farbmarken nicht mehr verändert. Das bedeutet, dass ein Doppelklick nun keine Nebeneffekte mehr hat und dass Sie das Popup-Menü mit der rechten Maustaste ohne ⇧ aufrufen können.

Die folgende Tabelle fasst die von *TColorPalette* bearbeiteten Mausaktionen bei ein- und ausgeschaltetem *NoSelection* zusammen:

NoSelection	Tastatur	Maus	Funktion
False		Links	Stiftfarbe wechseln
False		Rechts	Pinselfarbe wechseln
False	⇧	Links	Schriftfarbe wechseln
False	⇧	Rechts	Popup-Menü aufrufen
rue		Rechts	Popup-Menü aufrufen
ue/False		Doppelklick	Farbe editieren

Das letzte Property ist *SaveColors*, es ist hauptsächlich zu Demonstrationszwecken verffentlicht und wird in Kapitel 8.6.6 besprochen.

berschriebene Properties

In *TCustomControl* sind noch sehr viele häufig verwendbare Properties als *protected* deklariert. Es gehört daher zu den Standardaufgaben einer von *TCustomControl* abgeleiteten Klasse, sich daraus diejenigen Properties auszusuchen, die für ihre Zwecke am

besten geeignet sind. Bei *TColorPalette* fällt die Wahl auf die Mausereignisse, die Drag&Drop-Ereignisse und die Properties *Align* und *DragMode*.

Events

Den letzten Teil der Komponentenschnittstelle für den Objektinspektor bilden die Events. Ein Anlass für Events liegt bereits auf der Hand: Wie die meisten anderen Steuerelemente auch, definiert *TColorPalette* Events, die bei Veränderungen durch den Anwender ausgelöst werden, hier also jedes Mal, wenn sich eine der drei Farbmarken ändert. *TColorPalette* definiert für jede Farbmarke ein eigenes Event: *OnPenColorChange*, *OnBrushColorChange* und *OnFontColorChange*

Dazu kommen noch zwei etwas exotischere Events: *OnDefineColor* und *OnExpand Popup*. Ersteres Event ermöglicht dem Benutzer der Komponente, die Standardfarbeinstellung der Palette zu überschreiben.

Auch das Event *OnMouseDblClk* ist selbst definiert, denn das von *TControl* geerbte Event *OnDblClick* gibt ein Beispiel für die Unterschlagung von Qt-Ereignis-Parametern durch die CLX: Das CLX-Event enthält nicht die Mausposition als Parameter, obwohl Qt diese mit seinem *QMouseEvent* liefert. *TColorPalette* fängt daher das Qt-Mausereignis in seiner *EventFilter*-Methode selbst ab, um an diese Koordinaten zu gelangen. Das entsprechende Listing wurde bereits in Kapitel 8.4.1 gezeigt.

8.6.3 Implementierung der Komponente

In diesem Abschnitt werfen wir einen Blick auf die Implementierung der Hauptfunktionen der Farbpalette. Dabei geht es vorwiegend um Quelltextteile, die spezifisch für Komponenten sind.

Zeichnen der Komponente *R107*

Beim Zeichnen der Farben muss *TColorPalette* auf ein Ereignis wie *OnPaint* verzichten, kann aber stattdessen die geerbte virtuelle Methode *Paint* überschreiben. Innerhalb einer solchen *Paint*-Methode haben Sie ein *Canvas*-Objekt zur Verfügung, so wie beim direkten Zeichnen in das Formular innerhalb einer *OnPaint*-Bearbeitungsmethode (für dieses *Canvas*-Objekt muss Ihre Komponentenklasse allerdings von *TCustomControl* oder *TGraphicControl* abgeleitet sein).

TColorPalette macht sich keine Mühe, einzelne Farben mit einer dreidimensionalen Umrandung zu versehen, so dass mehr Platz für die Farben bleibt und diese auch bei einer sehr kleinen Palette nicht zu Farbtupfern verkommen:

```
procedure TColorPalette.Paint;
var
  ColumnIndex, LineIndex, { Zählvariablen }
```

```
    XSeite, YSeite: Integer;   { Seitenlängen }
    x, y: Integer;      { Position des gerade gezeichneten Eintrags }
    Farbe: TColorRef;   { Farbe und ...}
    FarbIndex: Integer; { ...Index der Farbe dieses Eintrags }
    THeight, TWidth: Integer;

    procedure DoPenMark; { Setzen der Stiftfarbenmarkierung }
    begin
      if FarbIndex = SelColors[cmPen] then begin
        Canvas.MoveTo(x*XSeite, y*YSeite+YSeite div 2);
        Canvas.LineTo((x+1)*XSeite, y*YSeite+YSeite div 2);
      end;
    end;
    procedure DoBrushMark;
    procedure DoFontMark;
    ... { prinzipiell wie DoPenMark }
  begin
    CalcSides(XSeite, YSeite);
    for LineIndex := 0 to Lines-1 do
      for ColumnIndex := 0 to Columns-1 do begin
        FarbIndex := LineIndex*Columns+ColumnIndex;
        Farbe := FColors[FarbIndex];
    (* Berechnung der x/y-Werte und der Farbe
          des gerade gezeichneten Eintrages, Einstellung
             der Attribute für das Farbrechteck *)
        y := LineIndex;
        x := ColumnIndex;
        DoSwapAlign(x, y);
        Canvas.Pen.Color := clBlack;
        Canvas.Pen.Mode := pmCopy;
        Canvas.Brush.Color := Farbe;
    (* Rechteck zeichnen *)
        Canvas.Rectangle(x*XSeite, y*YSeite,
                         (x+1)*XSeite+1, (y+1)*YSeite+1);
    (* Nun wird der Eintrag mit der Farbmarkierung
          gekennzeichnet, falls eine solche vorhanden ist: *)
        Canvas.Pen.Color := clWhite;
        { Der Stift soll auf jedem Hintergrund sichtbar sein: }
        Canvas.Pen.Mode := pmXor;
        DoBrushMark;
        DoPenMark;
        DoFontMark;
      end;
  end;
```

Ungültigerklärung

Wie bei der Zeichenfläche des TreeDesigners ändern sich auch bei der Farbpalette oft nur kleine Teile der Bildschirmausgabe. Wenn Sie mit der Maus eine andere Farbe mar-

kieren, ändert sich die Darstellung der bisher markierten und der neu markierten Farbe. Es gibt auch hier wieder drei Möglichkeiten, diese Änderung sichtbar zu machen:

- Ungültigerklären der gesamten Palette durch einen *Invalidate*-Aufruf. Dies kommt einem Neuzeichnen der Palette gleich.
- Direktes Zeichnen der veränderten Farbrechtecke, beispielsweise in der Methode *OnMouseDown*.
- Ungültigerklären nur der veränderten Rechtecke.

Wie im TreeDesigner, so wurde auch hier die dritte Möglichkeit gewählt und in einer eigenen Ungültigerklärungs-Methode, die *InvalidateRect* aufruft, gekapselt (weitere Details zum Ungültigerklären mit *InvalidateRect* finden Sie in Kapitel 6.5.3):

```
procedure TColorPalette.InvalidateCell(i: Integer);
var
  XSeite, YSeite: Integer;   { Seitenlängen }
  x, y: Integer;
  R: TRect;
begin
  y := i div Columns;
  x := i mod Columns;
  CalcSides(XSeite, YSeite);
  DoSwapAlign(x, y);
  R := Rect(x*XSeite, y*YSeite, (x+1)*XSeite+1, (y+1)*YSeite+1);
  InvalidateRect(R, False);
end;
```

Wichtigstes Einsatzgebiet für die neue Methode *InvalidateCell* ist das Aktualisieren zweier Farbzellen, wenn der Benutzer eine andere Farbe auswählt. Dies findet in der als Nächstes besprochenen Methode statt.

Verarbeitung der Mausereignisse

Auch die Mausereignisse stehen, abgesehen vom Doppelklick, in virtuellen Methoden zur Verfügung. Die Hauptaufgabe der neuen *MouseDown*-Methode von *TColorPalette* ist, die angeklickte Farbe herauszufinden und je nach Maustaste und dem Status von ⇧ eines der drei Farbänderungs-Ereignisse aufzurufen:

```
procedure TColorPalette.MouseDown(Button: TMouseButton;
                                  Shift: TShiftState; X, Y: Integer);
var
  ColorSelection: TColorMark;
begin
  if not NoSelection then begin
    if Button = mbRight then ColorSelection := cmBrush
      else if ssShift in Shift then ColorSelection := cmFont
        else ColorSelection := cmPen;
```

```
      InvalidateCell(SelColors[ColorSelection]);
      SelColors[ColorSelection] := PosToCell(X, Y);
      InvalidateCell(SelColors[ColorSelection]);

      case ColorSelection of
        cmPen: if Assigned(FOnPenColorChange) then
          FOnPenColorChange(self, FColors[SelColors[ColorSelection]]);
        cmBrush: if Assigned(FOnBrushColorChange) then
          FOnBrushColorChange(self,
             FColors[SelColors[ColorSelection]]);
        cmFont: if Assigned(FOnFontColorChange) then
          FOnFontColorChange(self,
             FColors[SelColors[ColorSelection]]);
      end;
    end;
    inherited MouseDown(Button, Shift, X, Y);
  end;
```

omponenten in Komponenten

Nachdem wir nun verschiedene Ereignisse durch virtuelle Methoden bearbeitet haben, kommen wir nun zum zweiten großen Unterschied zwischen der Komponenten- und der Formularprogrammierung, der *Verwendung* von Komponenten.

Wie in Kapitel 8.6.2 beschrieben, können Sie mit ⇧+linker Maustaste ein lokales Popup-Menü für die Farbpalette aufrufen und mit einem Doppelklick die aktuelle Farbe in einem Standard-Farbdialog editieren. Wäre die Farbpalette ein Formular, ätte sie zur Entwurfszeit eine *TPopupMenu*- und eine *TColorDialog*-Komponente erhalten.

Als Komponente muss *TColorPalette* diese beiden Komponenten selbst erzeugen und auch die zugehörigen Variablen selbst deklarieren. Die Klassendeklaration enthält für den Dialog die Variable *ColorDialog*, das Popup-Menü wird nicht in einer neuen Variablen, sondern im geerbten Property *PopupMenu* gespeichert.

Der Konstruktor einer Komponente ist normalerweise immer der richtige Ort, Unterkomponenten zu erstellen und deren Properties zu initialisieren. Während die Erstellung eines Standarddialogs noch sehr einfach ist, besteht die Initialisierung eines Popup-Menüs von Hand aus geschachtelten Aufrufen von Routinen der Unit *Menus*, die in Kapitel 4.6.1 beschrieben sind:

```
constructor TColorPalette.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
{ geerbte Properties anpassen: }
  Height := 100;
  Width := 100;
  PopupMenu := NewPopupMenu(self, 'internal', paRight, False,
```

```
    [NewItem('Farbe ändern', 0, False, True,
             ChangeColorClick, 0, 'intrn2'),
     NewItem('Farben zurücksetzen', 0, False, True,
             InitColorsClick, 0, 'intrn3')]);
{ eigene Properties anpassen: }
  RGBSteps := 5; { setzt auch Columns/Lines }
{ eine Komponente einfügen  }
  ColorDialog := TColorDialog.Create(self);
  ColorDialog.Options := [cdFullOpen];
{ den Rest erledigt: }
  InitColors;
end;
```

Nach der Ausführung des Konstruktors sind Popup-Menü und Farbdialog Teil der Komponente, wie sie auch Teil eines Formulars wären, wenn sie dort zur Entwurfszeit eingefügt worden wären. Die Programmierung dieser Komponenten unterscheidet sich also nicht mehr von der Programmierung der Komponenten eines Formulars.

Wichtig ist sowohl bei *TPopupMenu.Create* als auch bei *TColorDialog.Create*, dass die Komponente sich selbst als Elternkomponente und damit als ersten Parameter für diese Konstruktoren angibt. Das führt dazu, dass die CLX diese Komponenten automatisch freigibt, wenn die Farbpalette freigegeben wird (was wiederum automatisch geschieht, wenn das Formular freigegeben wird, in dem die Palette enthalten ist). Wenn Sie die Unterkomponenten nicht in den Besitz Ihrer Komponente überführen (indem Sie beispielsweise *nil* als ersten *NewPopupMenu*- oder *Create*-Parameter übergeben), müssen Sie diese Komponenten im Destruktor selbst freigeben.

Der Aufruf des Farbdialogs findet statt, wenn der Anwender den Popup-Menüpunkt *Farbe editieren...* auswählt oder wenn er auf die Farbe doppelklickt. Der Menüpunkt ist mit der Methode *ChangeColorClick* verknüpft, deren Dialogaufruf im folgenden Quelltextauszug gezeigt ist:

```
ColorDialog.CustomColors.Clear;
{ RightClicked enthält den Index der zu verändernden Farbe }
ColorDialog.CustomColors.
  Add('ColorA='+IntToHex(FColors[RightClicked], 6));
ColorDialog.Color := FColors[RightClicked];
if ColorDialog.Execute then begin
   ...
```

Interessant ist hier weniger der immer gleiche Aufruf von *Execute*, als die Initialisierung des Farbdialog-Properties *CustomColors*, bei dem es sich um eine Stringliste handelt (!). Um die erste Farbe des Dialogabschnitts *Selbst definierte Farben* zu setzen, fügen Sie der leeren Stringliste mit *Add* beispielsweise den String »ColorA=0A0B0C« hinzu, wobei der Farbwert hexadezimal angegeben werden muss und Sie bis »ColorP« gehen können.

8.6.4 Events mit Eingriffsmöglichkeiten

Neben den einfachen Farbänderungs-Events, deren Aufruf bereits im letzten Kapitel gezeigt wurde, definiert *TColorPalette* noch zwei weitere Events, die einen besonderen Unterschied zu den bisher beschriebenen Events aufweisen: Sie benachrichtigen das Formular weniger von einem Ereignis, als dass sie es ihm erlauben, das weitere Verhalten der Komponente durch Verändern eines Variablenparameters aktiv mitzugestalten.

OnExpandPopup *R137*

Da die Farbpalette ein eigenes Popup-Menü definiert, damit das *PopupMenu*-Property belegt und dieses sogar noch vor dem Komponentenbenutzer versteckt, könnte der Benutzer der Komponente kein Popup-Menü für die Farbpalette mehr definieren, wenn die Farbpalette keinen Ersatz für das Property *PopupMenu* bereitstellen würde.

Der Ersatz ist das Event *OnExpandPopup*, das über einen Variablen-Parameter verfügt, in dem der Komponentenbenutzer ein Objekt der Klasse *TPopupMenu* angeben kann. *TColorPalette* hängt dieses an das vordefinierte Popup-Menü an, sobald es aufgerufen ird. Die Voreinstellung des Parameters liegt bei *nil*, so dass der Benutzer nicht dazu verpflichtet ist, eine sinnvolle Methode zu schreiben. Der entsprechende Ausschnitt der Methode *MouseDown* (von der ein größerer Ausschnitt noch folgt) ist:

```
SecondMenu := nil;
if Assigned(FOnExpandPopup) then begin
  FOnExpandPopup(self, SecondMenu);
  if Assigned(SecondMenu) then...
```

Das Hauptformular des TreeDesigners, das die Farbpalette in einer Toolbar verwendet, nutzt dieses Ereignis und erweitert das Popup-Menü der Palette um die Punkte des Popup-Menüs, das auch für alle anderen Toolbars angezeigt wird (und über das die einzelnen Leisten angezeigt und versteckt werden können):

```
procedure TColorPaletteForm.PaletteExpandPopup(Sender: TObject;
  var ExpandPoints: TPopupMenu);
begin
  ExpandPoints := ControlBarPopup;
end;
```

Während die Benutzung dieses Events für den Benutzer äußerst einfach ist – er kann das Menüobjekt ja bequem mit dem Menü-Designer erstellen und muss dann nur eine einzeilige Methode wie *PaletteExpandPopup* schreiben –, muss *TColorPalette* das Menü in Handarbeit (mit erneuter Hilfe der in Kapitel 4.6.1 besprochenen Menüfunktionen) in sein eigenes Menü verschieben und nach dem Aufruf wieder zurückbefördern:

```
procedure TColorPalette.MouseDown(Button: TMouseButton;
                                  Shift: TShiftState; X, Y: Integer);
var
```

```
  P: TPoint;
  SecondMenu: TPopupMenu;
  i, StartExpand: Integer;
  MovedItem: TMenuItem;
begin
  if (NoSelection and (Button = mbRight)) or
    ((Button = mbRight) and (ssShift in Shift)) then begin
  (* Position für das Menü feststellen *)
    RightClicked := PosToCell(X, Y); { Formularvariable }
    P.X := X; P.Y := Y;
    P := ClientToScreen(P);
  (* Erfragen eines Menüs durch den Event-Aufruf *)
    StartExpand := PopupMenu.Items.Count;
    SecondMenu := nil;
    if Assigned(FOnExpandPopup) then begin
      FOnExpandPopup(self, SecondMenu);
      if Assigned(SecondMenu) then begin
        (* Für den Fall, dass das erweiternde Popup-Menü irgendwelche
           Vorbereitungen treffen muss, wird nun dessen OnPopup-Ereignis
           aufgerufen. (Dies wurde für den TreeDesigner 2.5 notwendig.) *)
        if Assigned(SecondMenu.OnPopup) then
          SecondMenu.OnPopup(SecondMenu);
        (* Menüverschmelzung *)
        while SecondMenu.Items.Count > 0 do begin
          MovedItem := SecondMenu.Items[0];
          SecondMenu.Items.Delete(0);
          PopupMenu.Items.Add(MovedItem);
        end;
      end;
    end;
  (* Menüaufruf *)
    PopupMenu.Popup(P.X, P.Y);
  (* Eingefügte Punkte wieder zurückkopieren *)
    while PopupMenu.Items.Count > StartExpand do begin
      MovedItem := PopupMenu.Items[StartExpand];
      PopupMenu.Items.Delete(StartExpand);
      SecondMenu.Items.Add(MovedItem);
    end;
  end;
  ... nun folgt die Behandlung der
        linken Maustaste (siehe Kapitel 8.6.3) ...
```

OnDefineColor

Eine weitere, ähnliche Eingriffsmöglichkeit erhält der Benutzer der Komponente mit dem Event *OnDefineColor*. Es wird jedes Mal aufgerufen, wenn eine Farbe auf ihren Standardwert zurückgesetzt wird, und erlaubt somit die Änderung der Standardfarben. Normalerweise geschieht dies immer mit allen Farben auf einmal – und zwar entweder bei der Konstruktion der Farbpalette oder wenn der Anwender aus dem Popup-Menü der Palette den Menüpunkt FARBEN ZURÜCKSETZEN auswählt.

Die mit *OnDefineColor* verknüpfte Methode erhält außer dem üblichen Absender-Parameter den Index der gefragten Farbe und einen Variablenparameter mit dem voreingestellten Farbwert. Wenn der Komponentenbenutzer diesen Wert ändern will, schreibt er ihn in diesen dritten Parameter.

Der Aufruf des Events findet in der Methode *InitColors* statt, nachdem das Array *FColors* bereits komplett initialisiert wurde:

```
if (Assigned(FOnColorDefine)) then
  for i := 0 to Columns*Lines-1 do begin
    UserDefinedColor := FColors[i];
    FOnColorDefine(self, i, UserDefinedColor);
    FColors[i] := UserDefinedColor;
  end;
```

Mit dieser Vorgehensweise hält sich *TColorPalette* an einen wichtigen Grundsatz der Komponentenentwicklung: Sie folgert aus der Tatsache, dass der Benutzer das *OnDefineColor*-Event bearbeitet, nicht, dass er in diesem den Wert von *ColorValue* setzt. Falls er das nicht tut, speichert der obige Code denselben Farbwert, der vorher in *UserDefinedColor* gespeichert wurde, wieder an Ort und Stelle zurück. Die vorgegebene Initialisierung der Farben wird also nicht durch zufällige Rückgabewerte der Event-Bearbeitungsmethode gestört.

orteil der Rückmeldungs-Events

Eine nicht gleichwertige Alternative zu *OnDefineColor* wäre es, wenn *TColorPalette* alle aktuell gültigen Farbwerte in einem Property zugänglich machen würde. Der Komponenten-Benutzer könnte dann zwar anfangs seine eigenen Werte in dieses Property schreiben, beim nächsten Farb-Reset mit dem Popup-Menüpunkt FARBEN ZURÜCKSETZEN wären diese Vorgaben allerdings außer Kraft gesetzt – wenn die Palette den Benutzer nicht wieder über ein Event von der Initialisierung informieren würde.

Das Event *OnDefineColor* erlaubt es dem Benutzer, auch nur eine einzelne aus 256 Farben mit einem neuen Wert zu definieren und sich in allen anderen Farben auf die Vorbelegung zu verlassen. Auch eine Berechnung der Standard-Farbwerte zur Laufzeit, in Abhängigkeit der Spalten- und Zeilenzahl der Farbpalette, wäre mit diesem Event sehr leicht möglich.

Auch wenn zumindest letztere Möglichkeit eher selten genutzt werden sollte, so gibt *OnColorDefine* doch ein gutes Beispiel dafür, wie sich die Schnittstelle einer Komponente erheblich flexibler gestalten lässt als bei ausschließlicher Verwendung von Properties. Das vorher besprochene Event *OnPopupExpand* hätte dagegen auch durch ein zweites *TPopupMenu*-Property ersetzt werden können.

8.6.5 Komponenten- und Property-Editoren

Die Farbpalette bietet eine gute Gelegenheit, Property- und Komponenteneditoren zu implementieren, mit denen sich eine Komponente noch weiter beeinflussen lässt als nur mit den Properties im Objektinspektor. Ebenfalls in der Unit *ColorPal* definiert sind:

- ein Property-Editor, in dem Sie die Farben der Palette einzeln editieren und Farbverläufe definieren können (Abbildung 8.8),
- ein Komponenteneditor, der zur Entwurfszeit nur in Form zweier lokaler Menüpunkte für die Komponente sichtbar ist.

Natürlich wäre es für den Anwender, der es letztlich mit der Palette zu tun bekommt, wichtiger, wenn die Palette die Farben zur Laufzeit ändern und in eigenen Palettendateien speichern könnte. Zum Lösen dieser Aufgabe benötigt jedoch das Formular, in dem die Palette später verwendet wird, lediglich zwei Speedbuttons, zwei Standard-Dateiwahldialoge und zwei *OnClick*-Methoden, weshalb wir uns hier sofort der größeren Herausforderung der Editoren zuwenden können. Die allgemeinen Grundlagen zu beiden Editortypen sind in Kapitel 8.4.4 beschrieben.

Entwurfszeit-Packages

Property- und Komponenteneditoren benötigen spezielle Units, die nicht in normalen Anwendungen, sondern nur in Packages enthalten sein können. Hierzu gehört beispielsweise die Unit *DesignEditors*, die einige weitere Units benötigt, die in Kylix nur innerhalb des Packages *designide* vorliegen (zu finden im *lib*-Verzeichnis der Kylix-Installation). Dieses Package müssen Sie der *requires*-Klausel des Packages hinzufügen, in dem sich Ihre Komponente befindet, für die Sie Editoren bereitstellen wollen.

Aus diesen Bedingungen folgt, dass Sie die Editoren in einer eigenen Unit definieren müssen, die nicht in normale Anwendungen eingebunden wird, wie im Abschnitt Auslagern der Editoren in eigene Units noch genauer erläutert werden wird.

Der Property-Editor *R139*

Ein Property-Editor ist ein Objekt einer von *TPropertyEditor* abgeleiteten Klasse. *TPropertyEditor* ist in der Unit *DesignEditors* deklariert, welche nur in Units eingebunden werden kann, die Teil eines Packages sind. In diesem Package muss auch das Kylix-Package *designide.dcp* eingebunden sein, damit die von *DesignEditors* benötigten Units zur Verfügung stehen.

TPropertyEditor definiert eine Reihe von virtuellen Methoden, die Sie überschreiben können, um das von Ihnen gewünschte Verhalten des Editors zu erzielen. Da keine der Methoden als *abstract* deklariert ist, können Sie alle für Sie nicht interessanten Metho-

den einfach unverändert von *TPropertyEditor* erben. Eine Beschreibung aller virtuellen Methoden würde den Rahmen dieses Kapitels bei weitem sprengen, daher müssen hier die im Beispiel-Editor verwendeten Methoden genügen:

- Die Methode *GetAttributes* liefert allgemeine Informationen über den Editor, beispielsweise, ob das Property aus einer Werteliste besteht (die der Objektinspektor in einer aufklappbaren Liste darstellt), ob es Unter-Properties hat oder ob es über einen Dialog editiert werden kann (in diesem Fall bringt Kylix einen entsprechenden Schalter im Objektinspektor an).
- Die Methode *Edit* wird dann aufgerufen, wenn der zuletzt genannte Schalter gedrückt wird. Die Methode kann dann das Formular des Editors aufrufen.
- Die Methode *GetValue* teilt Kylix mit, mit welchem Text das Property im Objektinspektor dargestellt werden soll.

Die Verwendung der anderen Methoden ist in der Online-Hilfe sowie im CLX-Quelltext in *DesignIntf.pas* beschrieben und ist normalerweise sehr einfach bzw. ergibt sich in den konkreten Anwendungsfällen.

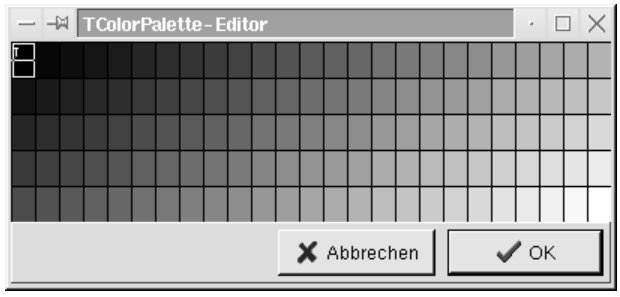


Abbildung 8.8: Der Editor der Palette besteht zum größten Teil selbst aus einer solchen.

Der Property-Editor für die Palette ist in der folgenden Klasse *TPaletteEditor* definiert, deren Methode *GetAttributes* den Editor als Dialog ausgibt und das Textfeld im Objektinspektor vor Schreibzugriffen schützt (Attribut *paReadOnly*). Da nicht alle Farben in das Textfeld passen, kann die Funktion *GetValue* entweder einen festen Hinweistext wie »(Farben)« anzeigen oder sich irgendeinen Aspekt der Palette aussuchen, den sie anzeigen möchte. *TPaletteEditor* wählt letztere Möglichkeit und zeigt zur Demonstration den Mittelwert der Blauwerte aller Farben an (zwischen 0 und 255):

```
type
  TPaletteEditor = class (TPropertyEditor)
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
  end;
```

```
function TPaletteEditor.GetAttributes: TPropertyAttributes;
begin  { teilt Kylix die Art des Property-Editors mit }
  GetAttributes := [paDialog, paReadOnly];
end;

function TPaletteEditor.GetValue: string;
var  { liefert den Text, der im Objektinspektor angezeigt wird }
  i, b: Word;
begin
  b := 0;
  with GetComponent(0) as TColorPalette do begin
    for i := 0 to Columns*Lines-1 do
      inc(b, GetBValue(FColors[i]));
    Result := 'Blau-Mittel = '+IntToStr(b div (Columns*Lines-1));
  end;
end;

procedure TPaletteEditor.Edit;
begin  { Aufrufen des Dialogs }
  EditPalette(GetComponent(0) as TColorPalette, Designer);
end;
```

Der eigentliche Editor wird in der Prozedur *EditPalette* aufgerufen, die dynamisch ein Editorformular erstellt, das als wichtigstes Element eine Farbpalette *Palette* enthält (dieses Formular ist hier nicht abgedruckt). *EditPalette* setzt die wesentlichen Properties des Paletteneditors auf die der Palette und lädt sie nach erfolgter Editierung wieder von dort zurück.

```
procedure EditPalette(EditedPal: TColorPalette; Designer: IDesigner);
var
  Dlg: TPalEditForm;
begin
  Dlg := TPalEditForm.Create(Application);
  with Dlg.Palette do begin
    { Die Editor-Palette soll aussehen wie die editierte Palette }
    NoSelection := True;
    FColors := EditedPal.FColors;
    Columns := EditedPal.Columns;
    Lines := EditedPal.Lines;
    Columns := EditedPal.Columns;
    SwapAlign := EditedPal.SwapAlign;
  end;
  { Aufruf des Editordialogs }
  if Dlg.ShowModal = idOK then begin
    { OK wurde betätigt -> Änderungen zurücklesen }
    EditedPal.FColors := Dlg.Palette.FColors;
    if Dlg.Palette.SaveColors then begin
      EditedPal.SaveColors := True;
      { Designer ist ein von TPropertyEditior geerbtes
        Element (und sollte eigentlich nie "nil" sein). }
      if Designer <> nil then
```

```
          Designer.Modified;
        end;
        EditedPal.Invalidate;
      end;
      Dlg.Free;
    end;
```

Die wesentliche Funktion des Editorformulars ist eine Farbverlaufsfunktion: Das Formular verwendet die Drag&Drop-Ereignisse der Palette, um beim Ziehen einer Farbe auf eine andere Farbe einen Farbverlauf zwischen diesen beiden Farben zu erzeugen (siehe CD-ROM). Außerdem gestattet es der Editor natürlich, jede einzelne Farbe mit einem Doppelklick zu editieren. Allerdings bedürfte es zu dieser Möglichkeit keines eigenen Property-Editors, denn die Farbpalette könnte den Doppelklick auch zur Entwurfszeit abfangen (in der Methode *WndProc*) und den Farbdialog aufrufen.

Der eigene Property-Editor wird im Fall der Farbpalette allerdings durch den Komponenteneditor unnötig gemacht, denn dieser bietet ebenfalls eine Möglichkeit, das Editorformular aufzurufen, und verwendet dabei die obige Methode *EditPalette* wieder.

Der Komponenteneditor *R138*

Die Komponenteneditoren sind den Property-Editoren sehr ähnlich und durch die Klasse *TComponentEditor* beschrieben. Um diese Klasse nutzen zu können, sind die gleichen Vorbereitungen notwendig wie bei *TPropertyEditor:* Einbinden der Unit *DesignEditors* in die Property-Editor-Unit und Einbinden des Packages *designide.dcp* in das Komponenten-Package.

Wie schon beim Property-Editor folgt hier nur eine Beschreibung speziell der in *TPaletteComponentEditor* überschriebenen virtuellen Methoden:

- In der Methode *GetVerbCount* teilen Sie Kylix mit, über wie viele verschiedene Verben/Befehle Ihr Editor verfügt.
- Sobald Kylix die Zahl der Verben kennt, kann es jeden einzelnen Befehl mit der Methode *GetVerb* erfragen. Überschreiben Sie diese Methode und geben Sie in Abhängigkeit vom als Parameter übergebenen Verb-Index den String zurück, der das Verb beschreibt. Kylix listet diese Strings im Popup-Menü der Komponente über den vordefinierten Punkten als Erstes auf.
- Nachdem Ihre Komponente die Verben mit dieser Methode aufgezählt hat, braucht sie nur noch darauf zu warten, das Kylix sie auffordert, eines dieser Verben auszuführen, weil der Benutzer den entsprechenden Menüpunkt ausgewählt hat. Hierfür ist die Methode *ExecuteVerb* zuständig, auch sie erhält wieder einen Verb-Index.
- Optional können Sie auch die Methode *Edit* überschreiben und darin definieren, wie die Komponente auf einen Doppelklick reagieren soll.

Der Komponenteneditor der Farbpalette bietet die Menüpunkte *Zurücksetzen* und *Farben editieren* an. Der erste Punkt setzt die Farben auf ihre Standardwerte zurück, die vom Property *RGBSteps* abhängig sind, während der zweite nur das eben definierte Property-Editorformular aufruft. Der Doppelklick ist eine alternative Möglichkeit, den Paletteneditor aufzurufen (Methode *Edit*):

```
type
  TPaletteComponentEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure Edit; override;
    procedure ExecuteVerb(Index: Integer); override;
  end;

implementation

function TPaletteComponentEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TPaletteComponentEditor.GetVerb(Index: Integer): string;
begin
  case index of
    0: Result := 'Zurücksetzen';
    1: Result := 'Farben editieren...';
  end;
end;

procedure TPaletteComponentEditor.Edit;
begin
  EditPalette(Component as TColorPalette, Designer);
end;

procedure TPaletteComponentEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: with (Component as TColorPalette) do begin
         InitColors;
         Invalidate;
       end;
    1: EditPalette(Component as TColorPalette, Designer);
  end;
end;
```

Registrieren der Editoren

Bei der Registrierung der Editoren kommt es wieder zu einem intensiven Einsatz von Typinformationen. Für einen Property-Editor übergeben Sie als letzten Parameter die

von *TPropertyEditor* abgeleitete Klasse des Property-Editors. In den ersten drei Parametern geben Sie an, bei welcher Gelegenheit der Editor eingesetzt werden soll:

- Im ersten Parameter geben Sie Informationen über den Typ der Properties an, für die der Editor verwendet werden soll. Sie erhalten diese Informationen von der Funktion *TypeInfo*, der Sie den Typenbezeichner übergeben (beispielsweise *TypeInfo(Boolean)*).
- Optional können Sie im zweiten Parameter eine Klasse angeben, auf die Kylix den Einsatz des Editors beschränken soll.
- Aus dieser Klasse können Sie im dritten Parameter ein einzelnes Property auswählen, für das der Editor als Einziges anzuwenden ist. Sie geben dieses Property als String an.

ür den Property-Editor der Farbpalette stellt sich die Frage, für welches Property er berhaupt gelten soll, denn das Farben-Array *FColors* ist im Property-Editor wie alle anderen Array-Properties nicht zugänglich. *TColorPalette* definiert dazu das boolesche Property *ColorDummy*, das nur dazu dient, ein Feld im Objektinspektor für den Aufruf des Editors zu reservieren. (Eleganter wäre es, die Farben in einer eigenen Klasse zu definieren, so dass *FColors* zu einem Objekt würde und auch im Objektinspektor angezeigt werden könnte). Die Registrierung des Property-Editors findet in der Prozedur *Register* statt, die bereits für die Registrierung der Komponente und der Property-Kategorien zuständig ist:

```
procedure Register;
begin
  RegisterComponents('Zum Buch', [TColorPalette]);
   { (in diesem Beispiel spielt der Typ des Properties keine Rolle) }
  RegisterPropertyEditor(TypeInfo(Boolean), TColorPalette,
                         'ColorDummy', TPaletteEditor);
  RegisterPropertiesInCategory(...); // siehe Kapitel 8.3.1
```

ür den Komponenteneditor ist der Registrierungsvorgang sehr einfach, da Sie nur die Komponenten- und die Editorklasse miteinander in Beziehung bringen müssen. Die Registrierungsprozedur der Farbpaletten-Unit endet damit wie folgt:

```
  RegisterComponentEditor(TColorPalette, TPaletteComponentEditor);
end;
```

Auslagern der Editoren in eigene Units

Während in der oben gezeigten *Register*-Prozedur die Komponente und ihre Editoren und Kategorien gemeinsam registriert werden, finden Sie auf der CD eine Trennung zwischen Komponenten- und Editoren-Unit. *TPaletteEditor* und *TPaletteComponentEditor* sind in einer eigenen Unit namens *ColorPalDesignTime* untergebracht. Während diese die Unit *ColorPal* natürlich in ihrer *uses*-Klausel aufführt, ist das umgekehrt nicht der

Fall, damit *ColorPal* völlig unabhängig von *ColorPalDesignTime* ist. Die oben gezeigten Aufrufe von *RegisterPropertyEditor*, *RegisterComponentEditor* und *RegisterPropertiesIn Category* finden in einer zweiten *Register*-Prozedur innerhalb von *ColorPalDesignTime* statt.

Diese Trennung hat zweierlei Vorteile:

- Es ist einfach möglich, getrennte Laufzeit- und Entwurfszeit-Packages von der Komponentenbibliothek des Buches herzustellen. Dazu muss im Laufzeit-Package gegenüber dem Entwurfszeitpackage lediglich die Unit *ColorPalDesignTime* entfernt werden. Da dies aber noch keine wesentliche Platzersparnis bringt und die Buchkomponenten möglicherweise nicht als Laufzeit-Package verwendet, sondern eher statisch zur ausführbaren Datei hinzugelinkt werden, ist dieser Fall nicht in die Praxis umgesetzt worden, das Package auf der CD ist also nur als Entwurfszeit-Package vorgesehen.
- Die Units des OpenTools API wie die schon erwähnte Unit *DesignEditors* können nicht in normale Anwendungen eingebunden werden und ihr Einsatz ist ja auch nur in Entwurfszeit-Packages sinnvoll. Wenn die *ColorPal*-Unit nun auch die Editorklassen enthalten würde, müsste sie solche ToolsApi-Units einbinden und könnte nun selbst nicht mehr in normalen Anwendungen genutzt werden, was ja ziemlich dem Zweck der Komponente widersprechen würde.

8.6.6 Speichern

Der Property-Editor für die Farben hat alleine jedoch noch nicht viel Nutzen, da sich die Farbwerte der Palette in einem Array-Property befinden, das die CLX grundsätzlich nicht automatisch speichern kann. Damit die geänderten Farben auch zur Laufzeit wieder sichtbar werden, benötigt *TColorPalette* weitere Methoden.

Zu einer an den Komponenten-Prinzipien orientierten Speicherungslogik gehören zwei Teile:

- Die Komponente muss eine Methode zur Verfügung stellen, mit der die CLX auch dieses nicht standardmäßige Property speichern kann. In diesem Fall muss die Methode 256 Farben, also theoretisch maximal ein KByte speichern.
- Vergleicht man den Platzbedarf von einem KByte mit den wenigen Bytes, die für gewöhnliche Properties genügen, erkennt man schnell, dass die Komponentenpalette die Farben nur speichern sollte, wenn diese auch geändert wurden. Die Komponente muss also eine Überprüfung durchführen, ob eine Speicherung notwendig ist.

Die Notwendigkeit des Speicherns

Wenn es sich bei den Farben um ein normales Property handelte, so könnten wir hinter diesem entweder einen *default*-Wert oder eine *stored*-Direktive angeben, anhand derer die CLX ebenfalls überprüfen könnte, ob die Speicherung notwendig ist,

```
property Colors: TImaginaryType read p1 write p2 stored AreColorsChanged;
```

wobei *AreColorsChanged* entweder eine Funktion oder eine Variable der Komponentenklasse ist. Zwar ist die oben gezeigte *stored*-Direktive bei den Farben der Palette nicht möglich, da die CLX nicht an der Speicherung beteiligt ist, trotzdem bleibt die Definition einer Variable oder Methode wie *AreColorsChanged* auch bei *TColorPalette* sinnvoll, wie sich gleich herausstellen wird (*TColorPalette* verwendet das Property *SaveColors*).

Nicht-Properties im Property-Tarnanzug

Grundsätzlich kann eine Komponente beliebige Daten in den Speicherungsprozess der CLX und der Kylix-IDE aufnehmen, indem sie die schon in *TPersistent* definierte virtuelle Methode *DefineProperties* überschreibt (siehe auch Kapitel 8.4.2). In dieser Methode kann eine Komponente ihre noch zu speichernden Daten nach Belieben aufteilen und gegenüber der CLX als Properties deklarieren. Eine solche Property-Deklaration besteht aus nahezu denselben Bestandteilen wie eine richtige Property-Deklaration (mit dem Unterschied, dass sie zur Laufzeit über einen Methoden-Aufruf stattfindet): Aus einem Property-Namen, aus einer Lese- und Schreibmethode und aus einem *Stored*-ähnlichen Flag.

Als Beispiel sehen wir uns zuerst die Schritte an, die im Fall der Farbpalette durchzuführen sind. Hierzu gehört zunächst die Definition der Lese- und Schreibmethode. Letztere erhält ein *TWriter*-Objekt als Parameter und tut nichts anderes, als die zu speichernden Farbwerte einzeln an dieses weiterzugeben:

```
procedure TColorPalette.WriteColors(Writer: TWriter);
var
  i: Integer;
begin
  for i := 0 to Columns*Lines-1 do
    Writer.WriteInteger(FColors[i]);
end;
```

Mit einer ähnlichen Methode liest die Komponente die Daten wieder aus einem *TReader*-Objekt:

```
procedure TColorPalette.ReadColors(Reader: TReader);
var
  i: Integer;
begin
  SaveColors := True; { siehe Abschnitt unten }
```

```
    for i := 0 to Columns*Lines-1 do
      FColors[i] := Reader.ReadInteger;
  end;
```

Als Nächstes wird die virtuelle Methode *DefineProperties* überschrieben. *DefineProperties* gehört zu den wenigen von *TPersistent* eingeführten Methoden und erhält ein abstraktes *TFiler*-Objekt als Parameter. Hinter diesem verbirgt sich ein konkretes *TWriter*- oder *TReader*-Objekt.

Für die Methode *DefineProperties* spielt es keine Rolle, mit welchem dieser beiden Objekte sie es zu tun hat, sie ruft in beiden Fällen die *TFiler*-Methode *DefineProperty* auf, um das Schein-Property zu deklarieren. Wie angekündigt, werden hier ein Name für das Property, die Lese- und die Schreibmethode und schließlich ein Flag angegeben, das besagt, ob die Daten gespeichert werden müssen:

```
procedure TColorPalette.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  Filer.DefineProperty('Colors', ReadColors, WriteColors, FSaveColors);
end;
```

Vielleicht wird Ihnen diese Vorgehensweise etwas merkwürdig vorkommen – warum können Sie z. B. nicht schon den Aufruf von *DefineProperty* davon abhängig machen, ob die Daten gespeichert werden müssen (*if FSaveColors then Filer.DefineProperties(...)*)? Diese Frage und die Notwendigkeit des Namensparameters lässt sich durch die Funktionsweise der Property-Speicherung erklären.

Funktionsweise

Alles in allem kommt es beim Speichern und Laden der als Properties verkleideten Daten zu einem beispielhaften Wechselspiel zwischen einem polymorphen *TFiler*-Objekt und einem polymorphen *TPersistent*-Objekt (hinter dem sich Ihre Komponente verbirgt). Dieses Wechselspiel wird hier am Beispiel des Schreibens einer erdachten Komponente *TNewComponent* erläutert:

- Nachdem *TWriter.WriteProperties* die Properties geschrieben hat, die die CLX automatisch speichern kann, sollen die anderen Daten gespeichert werden. Dazu ruft *TWriter* nun die Methode *DefineProperties* des zu speichernden Objekts auf (*TWriter* weiß an dieser Stelle nur, dass das zu speichernde Objekt eine von *TPersistent* abgeleitete Klasse hat). *TWriter* übergibt sich dieser Methode selbst als Parameter.
- In der Methode *TNewComponent.DefineProperties* wird das aufrufende *TWriter*-Objekt nun als abstraktes *TFiler*-Objekt angesehen. Ohne weiter darüber nachzudenken, zählt *TNewComponent* mit einer Reihe von *DefineProperty*-Aufrufen die zusätzlichen Properties auf.

- *DefineProperty* ist wieder an das aufrufende *TWriter*-Objekt gerichtet, das den Namen des Properties (angegeben im ersten Parameter) in den Stream schreibt und danach sofort wieder die Komponente aufruft, und zwar über die im dritten *DefineProperty* aufgerufene Schreibmethode (dies alles findet natürlich nur statt, falls der vierte Parameter *True* ist).
- So kommt es nun endlich zum Aufruf der *WriteData*-Methode von *NewComponent*. Diese schreibt die Daten, indem sie wieder Methoden von *TWriter* aufruft.

Beim Lesen der Komponente kann die CLX nun feststellen, ob das mit *DefineProperty* definierte Property gespeichert wurde, indem sie prüft, ob der im ersten Parameter angegebene Name im Stream steht. Falls dies nicht der Fall ist, ruft *TReader.DefineProperty,* die ebenfalls von *TNewComponent.DefineProperties* aufgerufen wird, die Methode *TNewComponent.ReadData* nicht auf.

Hinweis: Der vierte *DefineProperty*-Parameter hat beim Lesen also keinen Einfluss mehr. Die oben erwähnte Abfrage *if FSaveColors then Filer.DefineProperties(...)* in *TColorPalette.DefineProperties* würde also nicht funktionieren, da *DefineProperties* auch beim Lesen des Properties aufgerufen wird und erst die Methode *Filer.DefineProperties* erkennen kann, ob das Property gespeichert wurde oder nicht.

Das Property SaveColors

Es bleibt also noch die Aufgabe, die Variable zu definieren, die darüber entscheidet, ob gespeichert wird. *TColorPalette* initialisiert dazu im Konstruktor die Variable *FSaveColors* und ändert diese bei jeder Änderung einer Farbe.

Insbesondere dann, wenn die Farbwerte mit *ReadColors* gelesen werden, muss *FSaveColors* auf *True* gesetzt werden, damit die Daten bei der nächsten Speicherung wieder in den Stream geschrieben werden (*ReadColors* wird schließlich nicht nur beim Wiederherstellen des Formulars zur Laufzeit, sondern auch beim Laden des Formulars zur Entwurfszeit aufgerufen). Insofern ist das Flag *FSaveColors* besonders von normalen Veränderungsflags zu unterscheiden, die angeben, ob die Datei seit dem letzten Speichern geändert wurde. (Daher heißt *FSaveColors* auch nicht *FColorsChanged*.)

ommunikation mit der Kylix-IDE

Da *FSaveColors* über das Property *SaveColors* veröffentlicht ist (zum Zwecke der Demonstration), können Sie das Umschalten des Flags bei Änderung einer Farbe auch im Objektinspektor sehen. Voraussetzung hierfür ist, dass *TColorPalette* Kylix darüber informiert, dass sich der Inhalt der Komponente und möglicherweise die Properties geändert haben. Dies geschieht mit der bereits in Kapitel 8.4.4 erwähnten Methode

IDesignerHook.Modified, wobei *TColorPalette* vorher feststellen muss, ob ein solcher Designer überhaupt vorhanden ist. Es folgt der Teil der Methode *TColorPalette.ChangeColorClick*, der ausgeführt wird, wenn der Benutzer eine Farbe geändert hat:

```
FColors[RightClicked] := ColorDialog.Color; { Farbwert wird geändert }
InvalidateCell(RightClicked); { Farbzelle für ungültig erklären}
SaveColors := True; { Farben für das Speichern vormerken }
{ "Ungültig-Erklären des aktuellen Objektinspektor-Inhalts" }
if GetParentForm(self).DesignerHook <> nil then
  GetParentForm(self).DesignerHook.Modified;
```

Alle anderen Property-Änderungen finden im Objektinspektor selbst statt und müssen Kylix daher nicht extra über einen *Modified*-Aufruf mitgeteilt werden.

Hinweis: Wenn Sie ein Property im Objektinspektor ändern, liest Kylix auch die anderen Properties neu ein. Wenn ein Property andere Properties beeinflusst, werden diese Nebenwirkungen daher sofort sichtbar. Bei *TColorPalette* werden beispielsweise *Columns* und *Lines* durch *RGBSteps* beeinflusst, und in vielen Komponenten der CLX, die Text anzeigen, ändern Sie mit dem Property *Font.Height* auch das Property *Font.Size* und umgekehrt.

Binäre Properties

Ein besonderer Nachteil der oben abgedruckten Methoden zum Schreiben des Farb-Arrays ist ihre sehr geringe Effizienz. *TWriter* schreibt für jeden geschriebenen Integer-Wert zuerst eine Kennung (ob der Wert *Byte*, *Word* oder gar *LongInt*-Größe hat) und dann den eigentlichen Zahlenwert.

Um ähnliche Arrays effizienter schreiben und lesen zu können, sollten Sie ein datenintensives Property statt mit *TFiler.DefineProperty* mit der Methode *DefineBinaryProperty* definieren. Der Unterschied liegt in den Schreib- und Lese-Methoden, die Sie an diese Methode übergeben: Diese erhalten kein *TReader*- bzw. *TWriter*-Objekt als Parameter, sondern ein *TStream*-Objekt, das über effiziente Methoden zum Speichern von beliebig langen Speicherbereichen verfügt (siehe Kapitel 4.3). Da die Editiermöglichkeit der Palette zur Entwurfszeit sowieso eher demonstrativen Charakter hat, kann eine Optimierung hier jedoch ausbleiben.

8.6.7 Vordefinierte Aktionen

Zu guter Letzt eignet sich *TColorPalette* auch noch dazu, die Registrierung von selbst definierten Standardaktionen zu demonstrieren. Wenn Sie das Package mit *TColorPalette* in der IDE installiert haben, sollen Sie im Komponenteneditor von *TActionList* (siehe Kapitel 4.6.4) die Möglichkeit haben, mit NEUE STANDARD-AKTION... zwei Aktionen der Kategorie »Farbpalette« zu erzeugen, und zwar die beiden Aktionen, die Sie

schon aus dem Popup-Menü der Palette aufrufen können: FARBE EDITIEREN... und ARBPALETTE ZURÜCKSETZEN. Im Dialogfenster *Standardaktionen* werden diese Aktionen unter den Namen *TColorPalChangeColor* und *TColorPalReset* auftauchen.

Die Erben der Klasse TAction *R140*

Beide Aktionen müssen von der Klasse *TAction* abgeleitet werden, da sie aber über eine wichtige Gemeinsamkeit verfügen, definiert die Unit *ColorPal* zuerst eine gemeinsame Basisklasse:

```
TColorPalAction = class(TAction)
public
  function HandlesTarget(Target: TObject): Boolean; override;
end;
```

Wie in Kapitel 4.6.4 beschrieben, testet die CLX auf der Suche nach einer Zielkomponente (*Target*) für eine Aktion verschiedene Komponenten daraufhin, ob sie vom Aktionsobjekt verarbeitet werden können. Dies geschieht durch einen Aufruf der *HandlesTarget*-Methode. *TColorPalAction* antwortet genau dann mit einem *True*, wenn es sich bei der getesteten Komponente um eine *TColorPalette* handelt:

```
function TColorPalAction.HandlesTarget(Target: TObject): Boolean;
begin
  Result:=Target is TColorPalette;
end;
```

Die beiden speziellen Klassen, in denen nun die beiden Standardaktionen definiert werden, können sich die Methode *HandlesTarget* schon sparen, denn sie erben sie direkt von *TColorPalAction*. Beide benötigen noch eine Methode *ExecuteTarget*, um die Aktion auszuführen. Außerdem erhalten sie noch einen Konstruktor, der das *Caption*-Property mit einer Vorbelegung versieht, so dass man, wenn man diese Standardaktion verwendet, den Menüpunkt-Text nicht mehr extra anzugeben braucht.

Da beide Klassen gleich aufgebaut sind, soll hier der Abdruck einer der Klassen zur Erläuterung genügen:

```
TColorPalChangeColor = class(TColorPalAction)
public
  constructor Create(AOwner: TComponent); override;
  procedure ExecuteTarget(Target: TObject); override;
end;

implementation

constructor TColorPalChangeColor.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption:='Farbe editieren...';
end;
```

```
procedure TColorPalChangeColor.ExecuteTarget(Target: TObject);
begin
  (Target as TColorPalette).ChangeColorClick(Target);
end;
```

Die Methode *ExecuteTarget* erhält als Parameter immer ein Objekt, das vorher schon mit *HandlesTarget* als *TColorPalette* identifiziert wurde, und muss lediglich die schon bestehende Methode für den FARBE ÄNDERN...-Menüpunkt des Popup-Menüs aufrufen.

Schließlich muss die Kylix-IDE noch über die neuen Standardaktionen informiert werden, denn sie soll sie ja in einem ihrer Dialoge auflisten. Auch hier gibt es wieder eine Registrierungsfunktion, *RegisterActions*, welche in der schon bekannten Prozedur *Register* aufgerufen wird:

```
procedure Register; // Register-Prozedur der Komponenten-Unit
begin
  RegisterComponents('Zum Buch', [TColorPalette]);
  RegisterActions('Farbpalette', // << Aktions-Kategorie
                  [TColorPalChangeColor, TColorPalReset], nil);
end;
```

Da die Aktionen eine Laufzeitfunktion darstellen, werden sie in der Unit der Komponente selbst definiert, und nicht in einer eventuellen Laufzeit-Unit, die z. B. Property-Editoren und -Kategorien enthält (konkret in diesem Fall also in der Unit *ColorPal*, nicht in *ColorPalDesignTime*).

Dies ist quasi ein Minimalablauf zur Definition eigener Standardereignisse. Zu den zusätzlichen Möglichkeiten gehören:

- Die Anpassung des *Enabled*- oder *Checked*-Properties im Rahmen der *OnIdle*-Behandlung (entsprechend dem *OnUpdate*-Event eines *TAction*-Objekts wird in Standardaktions-Klassen die virtuelle Methode *UpdateTarget* verwendet).
- Die Erweiterung von *TAction* um neue Properties, die z. B. eine feste Zielkomponente vorgeben – Beispiele hierfür geben die von Kylix vordefinierten Datenbank-Standardaktionen, in denen Sie im *DataSource*-Property eine feste Datenquelle angeben können.

Interessante Beispiele für die Programmierung von Standardaktionen geben die CLX-Quelltexte *QDBActns.pas* und *QStdActns.pas*.

8.7 Formulare als Dialogkomponenten

Dieses Kapitel geht noch einen Schritt weiter als die Standard-Dialogkomponenten der CLX. Schon diese zeigen einige Vorteile gegenüber normalen Formularen wie z. B. die leichte Aufrufbarkeit über die Methode *Execute* und die bei allen Komponenten zu findenden Vorteile der Property-Einstellung im Objektinspektor. Zwar ist die *Execute*-Methode ein Vorteil gegenüber normalen Formularen, gegenüber anderen Komponenten ist sie jedoch eher ein Nachteil: die üblichen Steuerelementkomponenten funktionieren bereits, ohne dass sie zuerst durch eine Extramethode des Formulars aufgerufen werden müssen.

Die in diesem Kapitel vorgestellte Komponente ist eine Dialogkomponente, die diesen Aufruf von *Execute* unnötig macht und selbst für ihre Aktivierung sorgt, wenn Sie die entsprechende Option im Objektinspektor anschalten – ansonsten können Sie sie auch wie die Standarddialoge mit der *Execute*-Methode aufrufen. Das Ergebnis wird die Komponente *THotkeyManager* sein, die durch ihr alleiniges Vorhandensein in einem Formular mit Hauptmenü dafür sorgt, dass der Anwender die Tastenkürzel für die Menüpunkte beliebig anpassen kann, wobei die Änderung *persistent* sind, also zwischen zwei Programmläufen automatisch gespeichert werden.

8.7.1 Eine Hülle für das Formular

Wie jeder andere selbst erzeugte Dialog auch kann der Dialog für eine Komponente ein normales Formular sein. Dieses *ist* zwar eine Komponente (da von *TComponent* abgeleitet), kann aber nicht als solche in die Komponentenpalette eingebunden werden. Dafür bedarf es einer zusätzlichen nicht-visuellen Komponente als Verpackung, die dann im Formular so erscheint wie die Icons der Standarddialog-Komponenten.

THotkeyManager *R136*

THotkeyManager ist eine solche Hüllkomponente, die dazu dient, das eigentliche Dialogformular aufzurufen. Sie können ein solches Dialogformular als eigenständiges Projekt entwickeln, das Sie auch anderweitig verwenden können. *THotkeyManager* beispielsweise verwendet das Formular *THotkeyEditor*, das in Kapitel 4.6.3 beschrieben wurde (Abbildung 4.7) und bereits als normales Formular in den TreeDesigner eingebunden ist. Dieses Formular bleibt weiterhin in seiner eigenen Unit und kann dort innerhalb eines anderen Projekts visuell weiter entwickelt und verändert werden. Von der Komponenten-Unit aus lässt sich das Formular nur nicht-visuell über Programmcode ansprechen.

ür unsere Komponente stellt sich als Erstes die Frage, welche Klasse als Basisklasse dienen soll. Da es sich um eine nicht-visuelle Komponente handelt, kommen weder

TWidgetControl noch *TControl* in Frage. Letztendlich bleibt nur die Klasse *TComponent* übrig, von der *THotkeyManager* dann das Verhalten, sich zur Entwurfszeit als Icon darzustellen, sowie die veröffentlichten Eigenschaften *Name* und *Tag* erbt.

Das Verpacken des Formulars in der nicht-visuellen Komponente besteht nun lediglich darin, in *THotkeyManager* eine Instanz des Formulars zu deklarieren (Variable *Hotkey Editor*). Wie alle anderen Komponenten, die Sie nicht zur Entwurfszeit einfügen, müssen Sie auch solche eingepackten Formulare manuell initialisieren, indem Sie ihren Konstruktor aufrufen. Die Initialisierung, der Aufruf des Formulars und seine abschließende Freigabe finden allesamt in der *Execute*-Methode der Hüllkomponente statt:

```
function THotkeyManager.Execute: Boolean;
var
  HotkeyEditor: THotkeyEditor;
begin
  Result := False;
  HotkeyEditor := THotkeyEditor.Create(Owner);
  if Assigned(ActionList) then
    HotkeyEditor.ActionList := ActionList
  else HotkeyEditor.Menu := Menu;
  if HotkeyEditor.ShowModal = mrOK then begin
    if FileName <> '' then
      SaveShortCutsToFile(FMenu, FileName);
    Result := True;
  end;
  HotkeyEditor.Free;
end;
```

Falls der Benutzer das Formular mit *OK* schließt, nimmt die Methode an, dass sich Tastenkürzeleinstellungen geändert haben, und speichert sie mit der hier nicht weiter interessanten Prozedur *SaveShortCutsToFile*, die in der Unit von *THotkeyEditor* enthalten ist.

8.7.2 Die automatische Aktivierung

Die automatische Aktivierung von *THotkeyManager* besteht darin, dass diese Komponente von selbst einen Menüpunkt an das Hauptmenü des Formulars anhängt und diesen mit einer eigenen Methode verknüpft, die den Tastenkürzel-Dialog aufruft. Die Erweiterung des Menüs findet mit Hilfe von einigen aus Kapitel 4.6.1 bekannten Methoden wie folgt statt:

```
Menu.Items[AutoMenuIndex].Add(
  NewItem(AutoMenuText, 0, False, True, AutoActivate, 0, ''));
```

Um den neuen Punkt gleich mit dem Aufruf des Tastenkürzel-Dialogs zu verknüpfen, wird in den obigen Zeilen die folgende Methode *AutoActivate* als *OnClick*-Methode für den neuen Menüpunkt angegeben:

```
procedure THotkeyManager.AutoActivate(Sender: TObject);
begin
  Execute;
end;
```

erwendung der Komponente *R66*

Die folgenden Properties von *THotkeyManager* erlauben Ihnen, auf die automatische Aktivierung Einfluss zu nehmen:

- *AutoMenuIndex*: Hier geben Sie den Index des Hauptmenüpunkts an, hinter dessen Popup-Menü der Menüpunkt für den Tastenkürzeldialog eingefügt werden soll. Wenn Ihr fünfter Hauptmenüpunkt beispielsweise *Optionen* heißt, erreichen Sie mit einem *AutoMenuIndex* von vier, dass der automatische Menüeintrag am Ende des Optionsmenüs angehängt wird. Wenn Sie *Execute* manuell aufrufen wollen, müssen Sie *AutoMenuIndex* bei seinem voreingestellten Wert -1 belassen.
- In *AutoMenuText* können Sie den vorgegebenen Menütext »Tastenkürzel-Editor...« ändern.
- Wenn Sie in *FileName* einen Dateinamen angeben, lädt die Komponente die Tastenkürzel beim Programmstart automatisch und speichert sie nach jedem neuen Aufruf des Dialogs (in der bereits gezeigten Methode *Execute*).
- Auch das Property *Menu* ist optional, es gibt die Hauptmenükomponente an, in deren Popup-Menü sich die Komponente eintragen und dessen Tastenkürzel editiert werden sollen. Sie können es setzen, indem Sie aus der aufklappbaren Liste eines der im Formular vorhandenen *TMainMenu*-Objekte auswählen. Der letzte Dienst, den *THotkeyManager* anbietet, ist, das zu editierende Menü automatisch festzustellen. Wenn Sie im Property *Menu* keine Angabe machen, verwendet er stattdessen einfach das Hauptmenü des Formulars.
- Das Property *ActionList* wurde eingeführt, um auch die Tastenkürzel von Aktionslisten editieren zu können. Auch das Formular *THotkeyEditor* bietet dafür ein entsprechendes Property. In *THotkeyManager* ist es (anders als in *THotkeyEditor*) möglich, *ActionList* und *Menu* gleichzeitig zu setzen. In diesem Fall werden die Tastenkürzel aus *ActionList* editiert, und im *Menu* wird nur der automatisch erzeugte Menüpunkt zum Aufruf des Editors eingetragen. Wenn Sie die Tastenkürzel des Menüs editieren wollen, müssen Sie *ActionList* auf *nil* setzen (bzw. das Feld im Objektinspektor leer lassen).

Die automatische Menü-Erweiterung, die Auswahl des Menüs aus einer Dropdown-Liste und die automatische Feststellung des Menüs sind nur aus einer Komponente heraus möglich, die sich als Element in dem Formular befindet, dessen Menü verändert werden soll. Ohne die Komponente *THotkeyManager* als Verpackung könnte das Formular *THotkeyEditor* diese Funktionen nicht in dieser angenehmen Form anbieten.

8.7.3 Implementierung

Wir untersuchen zum Schluss zwei Aspekte von *THotkeyManager*, bei denen der Status der Komponente (Property *TComponent.ComponentState*) eine wichtige Rolle spielt und bei denen sich eine Gelegenheit ergibt, die virtuelle Methode *Loaded* zu überschreiben.

Das folgende Listing zeigt zunächst die Property-Schreibmethoden für *Menu* und *FileName*:

```
procedure THotkeyManager.SetMenu(NewMenu: TMainMenu);
begin
  if NewMenu <> FMenu then begin
    FMenu := NewMenu;
    if Assigned(FMenu) then begin
      ConnectToMenu;
      if FFileName <> '' then ReloadShortcuts;
    end;
  end;
end;

procedure THotkeyManager.SetFileName(NewName: string);
begin
  if FFileName <> NewName then  begin
    FFileName := NewName;
    if Assigned(FMenu) and (FFileName <> '')
       then ReloadShortCuts;
  end;
end;
```

SetMenu trägt den neuen Menüpunkt mit der Methode *ConnectToMenu* gleich in das Hauptmenü ein und lädt auch noch die Tastenkürzel, falls das Property *FileName* bereits gesetzt ist. Falls dies nicht der Fall ist, holt die Prozedur *SetFileName* dies nach, sobald *FileName* gesetzt wird. Beide Methoden sind also so geschrieben, dass Sie diese beiden Properties in beliebiger Reihenfolge setzen können und dass danach in jedem Fall die Initialisierung mit *ConnectToMenu* und *ReloadShortCuts* stattgefunden hat.

Hinweis: Allerdings setzt *THotkeyManager* voraus, dass die anderen beiden Properties, *AutoMenuIndex* und *AutoMenuText* bereits vorher gesetzt wurden. Um auch diese Bedingung zu beseitigen, müssten weitere Property-Schreibmethoden hinzugefügt und die oben gezeigten Methoden erweitert werden. Wenn Sie eine *THotkeyManager*-Komponente zur Entwurfszeit in das Formular einfügen, müssen Sie sich sowieso nicht um die Reihenfolge der Property-Initialisierung kümmern.

Der Unterschied zwischen Entwurfs- und Laufzeit

Das automatische Einbinden eines Menüpunkts zum Aufruf des Tastenkürzel-Editors ist zur Laufzeit sehr nützlich; da die Komponente jedoch auch von der Kylix-IDE ganz

normal geladen wird, findet die Menüpunkt-Einbindung auch zur Entwurfszeit statt, falls keine zusätzlichen Vorkehrungen getroffen werden. Der neue Menüpunkt wäre im Entwurfszeit-Formular nicht nur sichtbar, sondern er würde auch noch in der Formulardatei als Teil des Menüs gespeichert werden. Beim nächsten Laden dieser Datei in die IDE würde die CLX diesen Punkt als Teil des Menüs laden, und *THotkeyManager* würde ihn schließlich ein zweites Mal hinzufügen. Bei jedem Laden des Formulars würde so ein weiteres Exemplar des Menüpunkts zum Menü hinzukommen.

Die einfachste Lösung, um dies zu vermeiden, besteht in der Abfrage, ob die Komponente sich im Entwurfsmodus der Kylix-IDE befindet (*csDesigning in ComponentState*), und das Menü nur dann zu erweitern, wenn das nicht der Fall ist. Diese Abfrage ist Teil der Methode *ConnectToMenu*, so dass die oben gezeigten Property-Schreibmethoden nicht geändert werden müssen:

```
procedure THotkeyManager.ConnectToMenu;
begin
  if not (csDesigning in ComponentState)
     and (AutoMenuIndex <> -1) then
  begin
    if not Assigned(FMenu) then
      FMenu := (Owner as TForm).Menu;
    if Assigned(FMenu) then FMenu.Items[AutoMenuIndex].
      Add(NewItem(AutoMenuText, 0, False, True, AutoActivate, 0, ''));
  end;
end;
```

erwendung der Methode Loaded

Als Zweites soll nun der Ladevorgang der Komponente so angepasst werden, dass *ConnectToMenu* und *ReloadShortCuts* erst dann aufgerufen werden, wenn die Komponente fertig initialisiert ist und alle Properties gesetzt sind. Diese weitere Einschränkung ist hier zwar nicht zwingend erforderlich, stellt aber in der derzeitigen Version von *THotkeyManager* sicher, dass auch die Properties *AutoMenuIndex* und *AutoMenu xt* richtig gesetzt sind, wenn der neue Menüpunkt zum Menü hinzugefügt wird.

Wie bei der ausführlichen Beschreibung des Ladeprozesses in Kapitel 8.4.2 erwähnt, uft die CLX die virtuelle Methode *Loaded* auf, wenn sie die Komponente fertig aus der Formular- bzw. kompilierten Anwendungs-Datei geladen hat. Neben dem obligatorischen Aufruf der geerbten Methode holt *THotkeyManager* die Anpassung des Menüs in der überschriebenen *Loaded*-Methode nach:

```
procedure THotkeyManager.Loaded;
begin
  inherited Loaded;
  ConnectToMenu;
  ReloadShortCuts;
end;
```

Um doppelte Aktionen zu vermeiden, darf nun während des Ladens jedoch keine Menüveränderung mehr stattfinden, wenn die CLX die Properties aus der Formulardatei lädt und die Methoden *SetFileName* und *SetMenu* aufruft. Daher fragen *ConnectToMenu* und *ReloadShortCuts* zusätzlich zum oben beschriebenen *ComponentState*-Flag *csDesigning* das Flag *csLoading* ab und führen ihre Aktion nur aus, wenn keines der Flags gesetzt ist. So wird beispielsweise die *if*-Bedingung der oben gezeigten Methode *ConnectToMenu* wie folgt erweitert:

```
if not (csDesigning in ComponentState)
   and not (csLoading in ComponentState)
   and (AutoMenuIndex <> -1) then
```

Es gibt also drei Möglichkeiten, wann *THotkeyManager* das Hauptmenü des Formulars anpasst:

- gar nicht, wenn sich das Formular im Entwurfsmodus befindet,
- in der Methode *Loaded*, wenn die Komponente zur Laufzeit aus den Formular-Ressourcen der ausführbaren Datei geladen wird, und
- sofort in den Schreibmethoden der Properties *FileName* und *Menu*, wenn Sie diese direkt im Quelltext setzen, beispielsweise nach einer dynamischen Erzeugung einer *THotkeyManager*-Komponente zur Laufzeit.

Die Klassendeklaration

Das Listing der Klassendeklaration fasst die besprochenen Teile von *THotkeyManager* zum Bilden der Hülle um das Formular, zum Definieren der Entwurfszeitschnittstelle, für die automatische Menüerweiterung und die automatischen Dateioperationen zusammen:

```
type
  THotkeyManager = class(TComponent)
  protected
    FMenu: TMainMenu;
    FActionList: TActionList;
    FAutoMenuIndex: Integer;
    FAutoMenuText: string;
    FFileName: string;
    procedure SetFileName(NewName: string);
    procedure SetMenu(NewMenu: TMainMenu);
    procedure SetActionList(NewList: TActionList);
    procedure ReloadShortCuts;
    procedure ConnectToMenu;
    procedure Loaded; override;
    procedure AutoActivate(Sender: TObject); { verknüpft mit OnClick }
    function MenuTextChanged: Boolean; { siehe stored von AutoMenuText }
  public
    constructor Create(AnOwner: TComponent); override;
```

```
    function Execute: Boolean;
  published
    property AutoMenuIndex: Integer read FAutoMenuIndex
               write FAutoMenuIndex default -1;
    property AutoMenuText: string read FAutoMenuText
               write FAutoMenuText stored MenuTextChanged;
    property Menu: TMainMenu read FMenu write SetMenu;
    property ActionList: TActionList read FActionList write SetActionList;
    property FileName: string read FFileName write SetFileName;
  end;
```

Einige Teile davon sind nicht besprochen worden, enthalten aber keine besonderen Neuigkeiten mehr, wie Sie im vollständigen Quelltext der CD überprüfen können.

8.8 Weitere Komponenten zum Buch

Dieses Kapitel gibt einen kurzen Überblick über die zusätzlichen Komponenten, die Sie auf der CD dieses Buchs finden und die auch in einigen Beispielprogrammen verwendet werden können. Die Verwendung der Komponenten ist jeweils ziemlich einfach und in der Implementierung der Klassen werden im Wesentlichen die in diesem Kapitel besprochenen Prinzipien wiederholt. Zu beidem – Verwendung und Implementierung – gibt es im Folgenden kurze Hinweise.

8.8.1 Anzeige von Verzeichnisstrukturen und Dateilisten

Sehr nützlich für das Browsen in Verzeichnisstrukturen, ohne dafür einen Dialog öffnen zu müssen, sind die Komponenten *TDirectoryTreeView* und *TFileTable* sowie die etwas spartanischere *TFileListBox*, die hauptsächlich der Kompatibilität mit Delphi dient.

TDirectoryTreeView *R53*

Diese in mehreren Beispielprogrammen dieses Buchs verwendete Komponente zeigt die Verzeichnisstruktur des Systems an, per Voreinstellung beginnend beim Wurzelverzeichnis, was aber über das Property *RootDirectory* geändert werden kann. Sie wurde unter dem Gesichtspunkt der Kompatibilität mit *TDirectoryListBox* von Delphi ür Windows entworfen und verfügt daher auch über ein Property namens *Directory*, ber welches zur Laufzeit das gerade gewählte Verzeichnis abgefragt werden kann. Um die Kompatibilität mit Delphi noch zu erhöhen, wurde auch eine Komponente mit dem Namen *TDirectoryListBox* definiert, die jedoch eine leere Ableitung von *TDirectoryTreeView* ist, also nur einen Alias-Namen für diese liefert.

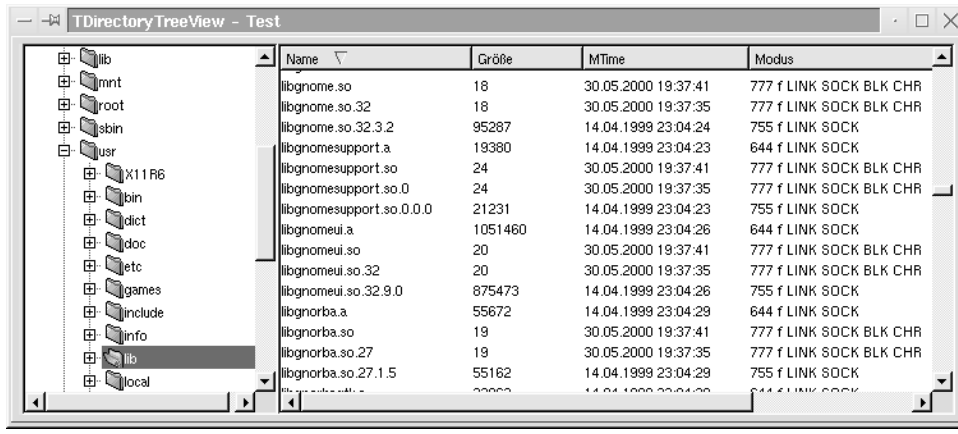


Abbildung 8.9: Ein DirectoryTreeView mit angeschlossener FileTable

Das Beispielprogramm *FileSystemBrowsersTest* auf der CD (siehe Abbildung 8.9) demonstriert die Verwendung dieser Komponente im Zusammenhang mit der unten beschriebenen *TFileTable*. Die im TreeView verwendete Bilderliste enthält Bilder, die von KDE übernommen werden. Diese Liste ist jedoch nicht fest in die Komponente *TDirectoryListBox* integriert, so dass Sie die Bilderliste aus dem Beispielprogramm kopieren müssen, falls Sie die gleichen Bilder verwenden wollen.

Die Implementierung dieser Komponente arbeitet hauptsächlich mit den von *TTreeView* geerbten Properties und Methoden, um den Verzeichnisbaum dynamisch aufzubauen. Dabei werden zwei TreeView-Ereignisse durch das Überschreiben dynamischer Methoden bearbeitet. Diese Methoden sind *CanExpand* und *Change*, welche den Ereignissen *OnExpanding* und *OnChange* entsprechen, die für *TTreeView* im Objektinspektor angezeigt werden. In *CanExpanding* werden die Unterverzeichnisse des geöffneten Verzeichnisses eingelesen und zur TreeNodes-Struktur eingefügt, in *Change* wird ein neu gewähltes Verzeichnis an eine eventuell verknüpfte *FileTable* oder *FileList* weitergegeben. Den Quelltext der *CanExpanding*-Methode sowie weitere Listing-Auszüge von *TDirectoryTreeView* finden Sie in Kapitel 3.6.3).

TFileTable

Da häufig auch noch der Inhalt eines Verzeichnisses angezeigt werden soll, enthält *TDirectoryTreeView* auch noch ein Property *FileTable*, das Sie mit einer *TFileTable* komponente verknüpfen können. Diese Komponente gehört ebenfalls dem Buchkomponenten-Package an und listet die Dateien eines bestimmten Verzeichnisses auf (siehe Abbildung 8.9). Neben dem Namen stellt Sie in weiteren Spalten Informationen über die Größe, den Zeitstempel, den Modus, die Benutzer-ID des Dateibesitzers sowie die Gruppen-ID des Besitzers dar. Letztere drei Informationen sind Linux-spezifisch und

dienen eher der Demonstration, wie Sie diese Informationen in Kylix erhalten (mit Hilfe der Funktion *fstat*), als dass die Darstellung dieser drei Spalten als perfekt einzustufen wäre. Für den Modus werden lediglich einige Kürzel angezeigt, zu denen Sie sich etwa aus der *man*-Dokumentation zu *fstat* weitere Informationen holen können.

Die Komponente führt drei neue Properties ein:

- Das wichtigste Property ist *FileName* – hier können Sie erwartungsgemäß den Namen der gerade selektierten Datei abrufen.
- In *Directory* wird das Verzeichnis angegeben, das aufgelistet werden soll, es wird automatisch von einer *DirectoryTreeView*-Komponente gesetzt, wenn Sie deren *FileTable*-Property entsprechend eingestellt haben.
- Im *Mask*-Property können Sie eine Suchmaske für die Dateien angeben und so die Auflistung einschränken. Voreinstellung ist unter Linux die Maske »*« für »alle Dateien«.

Darüber hinaus versieht *TFileTable* noch einige von *TListView* geerbte Properties mit neuen Vorgabewerten:

- *ShowColumnSortIndicators* wurde auf *True* gesetzt, so dass der Benutzer anhand einer Markierung im Spaltentitel ersehen kann, nach welcher Spalte die Liste sortiert ist.
- *Sorted* wurde auf *True* gestellt, um die von *TListView* freundlicherweise zur Verfügung gestellte Sortierfunktion zu nutzen. Sie können zur Laufzeit auf Spaltentitel klicken, um nach einem anderen Kriterium zu sortieren.
- *ViewStyle* wurde auf *vsReport* gestellt, damit die zusätzlichen Spalten neben dem Dateinamen auch sichtbar sind.
- Schließlich wurde auch *ReadOnly* eingeschaltet, damit der Benutzer die Dateinamen nicht editieren kann, denn eine Umbenennungs-Funktion ist von dieser Komponente nicht vorgesehen (lässt sich aber wohl nachträglich implementieren, auch durch das Formular, welches die Komponente verwendet).

Hinweis: Als Alternative zu *TFileTable* finden Sie auf der CD auch noch eine *TFileListBox*, die als Ersatz für die gleichnamige Komponente von Delphi für Windows verwendet werden kann. Sie ist nicht von *TListView*, sondern von *TListBox* abgeleitet und zeigt nur die Dateinamen ohne weitere Informationen an. Das Property *FileList* in *TDirectoryTreeView* dient der Verknüpfung mit einer solchen Liste, es entpricht daher dem gleichnamigen Property der *TDirectoryListBox*-Komponente von Delphi für Windows.

8.8.2 Abschalten von ausgewählten Meldungsfenstern

In Kapitel 4.2.3 wurde bereits die Komponente *TStateSaver* vorgestellt, die dem automatischen Speichern der Eingaben und Einstellungen dient, welche der Benutzer zur Laufzeit in einem Formular gemacht hat, und die auch für die automatische Wiederherstellung beim nächsten Programmstart sorgt. Auch Hinweise zur Implementierung dieser Komponente sind in dem genannten Kapitel gegeben. An dieser Stelle soll eine mit *TStateSaver* zusammenarbeitende Komponente vorgestellt werden.

TPersistentConfirmation *R74*

Eine Anwendung, die vor der Ausführung bestimmter Aktionen eine Bestätigung des Benutzers erfragt oder die Meldungsfenster über bestimmte Ergebnisse ihrer Aktionen anzeigt, weist eine gewisse Freundlichkeit gegenüber dem Benutzer auf und ist auch leicht zu programmieren, da hierfür nur einige *ShowMessage*- und *MessageDlg*-Aufrufe erforderlich sind.

Noch freundlicher zum Benutzer ist allerdings eine Anwendung, die diesem auch ermöglicht, die wiederholte Anzeige desselben Meldungsfensters für die Zukunft abzuschalten. Dies vervielfältigt jedoch den Programmieraufwand, weil Steuerelemente für das Abschalten der Meldung eingebaut und abgefragt werden müssen (was in *ShowMessage* und *MessageDlg* schon einmal nicht möglich ist) und möglicherweise die Wahl des Benutzers noch in einer Konfigurationsdatei gespeichert werden müssen.

All dies können Sie automatisieren, wenn Sie die Komponenten *TPersistentConfirmation* und *TStateSaver* kombinieren. Für jede Bestätigungsmeldung fügen Sie eine *TPersistentConfirmation* in Ihr Formular ein, deren Properties wie folgt eingestellt werden:

- *Caption* erhält den Titel des anzuzeigenden Meldungsfensters
- *ConfirmationText* den Textinhalt,
- *HasCancelButton* ist *True*, wenn ein Abbruch-Schalter im Fenster angezeigt werden soll.
- *HasOkButton* gilt entsprechend für einen *Ok*-Schalter.

Statt nun *ShowMessage* oder *MessageDlg* aufzurufen, rufen Sie die *Execute*-Methode von *TPersistentConfirmation* auf:

```
if PersistentConfirmation.Execute = True then ...
```

Diese Methode zeigt ein Fenster an, wie in Abbildung 8.10 gezeigt. Mit dem Ergebnis von *True* sagt die Methode, dass der Benutzer den OK-Schalter gedrückt hat. Schaltet der Benutzer die Anzeige des Meldungsfensters ab, so liefern weitere Aufrufe von *PeristentConfirmation* gleich den Wert *True* zurück, ohne das Fenster anzuzeigen.

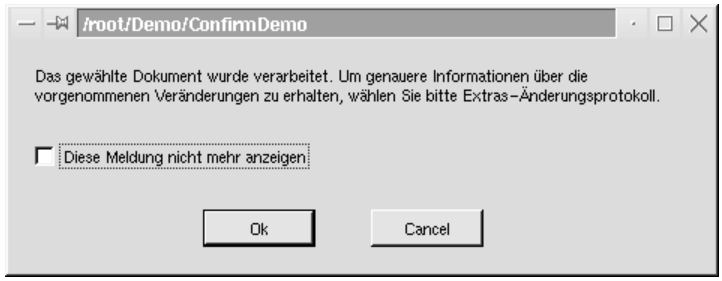


Abbildung 8.10: Ein Meldungsfenster von TPersistentConfirmation

Diese Verhaltensweise können Sie durch das Property *DefaultResult* ändern: Schalten Sie dieses auf *False*, damit *Execute* immer *False* zurückliefert, falls der Benutzer die Anzeige des Fensters abbestellt hat.

Schließlich gibt es noch das Property *DontShowAgain*, in dem der aktuelle Status (Meldungen anzeigen oder nicht) gespeichert wird. Es wird normalerweise automatisch verwaltet, kann von Ihnen jedoch dazu verwendet werden, beispielsweise alle Meldungen wieder einzuschalten (siehe Beispielprogramm *ConfirmDemo* auf der CD).

Selbstverständlich können Sie diese Komponente leicht gemäß Ihren Ansprüchen erweitern, so dass Sie z.B. auch bestimmte von *MessageDlg* gewohnten Optionen darin einstellen können.

Zur Implementierung der Komponente: *TPersistentConfirmation* ist im Wesentlichen nur ein weiteres Beispiel für eine nicht-visuelle Komponente, die ein Dialogformular anzeigt (siehe Kapitel 8.7). Die Methode *Execute* ist sehr simpel aufgebaut. Mit der Speicherung der Einstellung hat die Komponente nichts zu tun, sie wird alleine von *TStateSaver* übernommen.

8.8.3 Delphi-Kompatibilitäts-Komponenten

Drei weitere Komponenten sind zur leichteren Portierung von Delphi für Windows-Anwendungen vorgesehen, sie bieten einen teilweise sehr provisorischen Ersatz von Delphi-Komponenten, die unter Kylix nicht verfügbar sind: *TApplicationEvents*, *TUpDown* und *TPrinterSetupDialog*. Die Funktionsweise dieser Komponenten unter Delphi sei hier als bekannt vorausgesetzt, so dass im Folgenden nur die Einschränkungen erwähnt werden:

- *TApplicationEvents* darf im Gegensatz zum Vorbild von Delphi nur ein einziges Mal pro Anwendung verwendet werden und stellt nur ein einziges Event zur Verfügung (*OnIdle*), mit dem Sie eine Bearbeitung für das *OnIdle*-Ereignis von *TApplication* verknüpfen können. Für die Portierung einer Anwendung, die von *TApplication* nur dieses eine Ereignis bearbeitet und dies auch nur mit einer Methode, genügt

das jedoch. *TApplicationEvent* wurde bei der Portierung des TreeDesigners verwendet und kann natürlich auch erweitert werden, so dass es bei Bedarf auch andere Ereignisse von *TApplication* unterstützt.

- *TUpDown* ist in Wirklichkeit ein unter falschem Namen getarntes *SpinEdit*-Widget. Es stellt die von *TUpDown* bekannten Properties *Associate* und *Position* zur Verfügung (alle anderen *TUpDown*-Elemente sind nicht vorhanden) und lässt sich als Übergangslösung verwenden, bevor ganz auf *SpinEdit* umgestellt wird.

  Für diese Umstellung besitzt die Klasse ein *UseSpinEdit*-Property. Wenn dieses *False* ist, funktioniert das *Position*-Property wie unter Windows und gibt den Zahlenwert an, der in dem über das *Associate*-Property verknüpften Editierfeld steht. Durch Zuweisung an *Position* ändern Sie somit auch das *Text*-Property des Editierfelds. Ist *UseSpinEdit* dagegen *True*, stimmt *Position* immer mit dem Wert des SpinEdit-Erbes der Komponente überein.

- *TPrinterSetupDialog* enthält nur eine einzige Zeile Code in der Methode *Execute*. Darin wird die Methode *Printer.ExecuteSetup* aufgerufen, die wiederum den in Qt eingebauten Druckereinstellungs-Dialog startet (siehe Kapitel 6.6.3). Die praktisch wahrscheinlich weniger relevante Einschränkung besteht darin, dass *TPrinterSetupDialog* nur von *TComponent* abgeleitet ist und nicht von *TCommonDialog* wie die Delphi-Komponente. Dadurch fehlen *TPrinterSetupDialog* einige Elemente, jedoch wird normalerweise sowieso nur die *Execute*-Methode genutzt.

Der wesentliche Vorteil dieser Komponenten liegt darin, dass sich, wenn sie in der Komponentenpalette von Kylix installiert sind, auch Delphi-Formulare mit diesen drei Komponenten in Kylix laden lassen, ohne dass die drei Komponenten mit allen ihren Properties verloren gehen. In der Sprache der Software-Technologie handelt es sich bei den drei Komponenten quasi um *Adapter*, die die in der CLX vorhandenen Klassen *TApplication* und *TSpinEdit* sowie das *Printer*-Objekt an die bekannten Delphi-Schnittstellen *TApplicationEvents*, *TUpDown* und *TPrinterSetupDialog* anpassen, so dass der Programm-Code, der diese Delphi-Schnittstellen verwendet, nicht geändert werden braucht (wobei die oben genannten Einschränkungen zu beachten sind).

8.8.4 Sonstige Komponenten

Schließlich finden Sie auf der Palettenseite dieses Buchs noch die beiden Komponenten *TMiniHistoryGraph* und *TMenuToolBar*.

TMiniHistoryGraph

Die Funktion und Verwendung dieser Komponente wird in Kapitel 5.1.5 an einem Beispiel demonstriert.

Da die Komponente sich selbst zeichnet, besteht die wichtigste Ereignisbearbeitung in der Überschreibung der *Paint*-Methode, die von *TGraphicControl* erbt. Zur Entwurfszeit zeigt die Komponente, um ihre Funktion zu symbolisieren, vier Balken an. Um zu erkennen, ob es sich um »Entwurfszeit« oder »Laufzeit« handelt, fragt sie *csDesigning in ComponentState* ab.

TMenuToolBar *R I I*

Das Menü der Kylix-IDE hebt sich von den Menüs einer normalen Kylix-Anwendung dadurch ab, dass es sich wie eine Toolbar innerhalb der *TControlBar*-Komponente verschieben lässt (Abbildung 8.11). Mit der Komponente *TMenuToolBar* können Sie diese Funktion auch in eine Kylix-Anwendung einbauen:

- Die MenuToolbar-Komponente können Sie zur Entwurfszeit ganz normal in Ihr Formular einsetzen, sie bleibt zur Entwurfszeit leer.
- Die eigentlichen Menüpunkte definieren Sie wie gewohnt in einer eigenen *TMainMenu*-Komponente. Damit diese nicht vom Formular als Menü angezeigt wird, löschen Sie im Objektinspektor das *Menu*-Property des Formulars.
- Um die Punkte der MainMenu-Komponente in die MenuToolbar zu übertragen, setzen Sie *zur Laufzeit* das Property *TMenuToolbar.MainMenu* auf Ihre *MainMenu*-Komponente (z. B. beim Ereignis *OnCreate*).

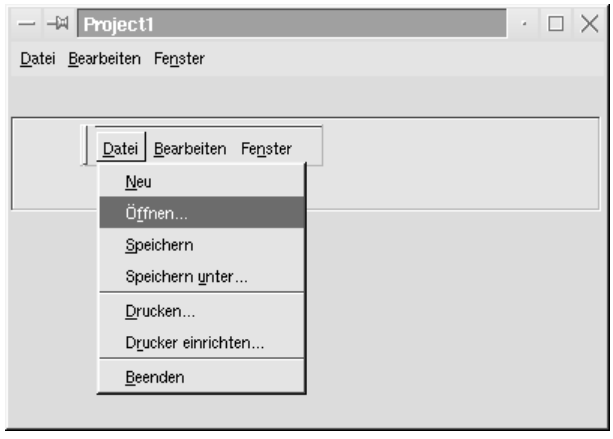


Abbildung 8.11: TMenuToolbar im Vergleich zu einem normalen Hauptmenü

Der TreeDesigner gibt ein Beispiel zu dieser Verwendung von *TMenuToolbar*. Er verwendet auch noch das Property *ChildMenu*, in dem das Menü eines MDI-Kindfensters angegeben wird, das von *TMenuToolbar* mit dem Hauptmenü verschmolzen wird. Dies funktioniert analog zur automatischen Menüverschmelzung der CLX in MDI-Fens-

tern, jedoch nicht vollkommen automatisch: Das Property *TMenuToolbar.ChildMenu* muss manuell gesetzt werden, immer wenn ein anderes MDI-Kindfenster aktiviert wird.

Die Implementierung dieser Komponente gestaltet sich nicht ganz so einfach, da die *TToolButton*-Komponente der CLX im Gegensatz zu derjenigen aus der VCL kein *MenuItem*-Property besitzt und somit auch keine Unterstützung für die Verbindung mit einem Hauptmenüpunkt mitbringt. Stattdessen unterstützt *TToolButton* lediglich ein Popup-Menü und ein DropDown-Menü, was beides nicht mit der erwünschten Funktionsweise zu vereinbaren ist: Schon wenn die Maus über einen anderen Hauptmenüpunkt bewegt wird, soll dessen Menü automatisch geöffnet werden. Die übliche Funktionsweise eines Popup- oder DropDown-Menüs sieht aber so aus, dass dieses Menü erst durch einen Mausklick geschlossen wird.

TMenuToolBar muss also, während eines der Menüs angezeigt wird, die Mausbewegung überwachen und feststellen, wann die Maus sich über einem anderen ToolButton befindet. Sie fängt die *OnClick*-Ereignisse der eigenen Schalter ab und überwacht daraufhin auf Anwendungsebene alle Mausbewegungs-Ereignisse von Qt. Eine genauere Erläuterung würde den Rahmen dieses Kapitels sprengen, jedoch sei darauf hingewiesen, dass *TMenuToolBar* durch einen (von der CLX abgeschauten) Trick die geschützte Methode *TControl.MouseLeave* aufruft. Bei Verwendung solcher Tricks kann jedoch nicht sichergestellt werden, dass die Komponente ohne Änderung auch mit zukünftigen Versionen der CLX harmonisch zusammenarbeitet.

Anhang A: Rezeptverzeichnis

1 Benutzerschnittstelle: Entwurf

1 Steuerelemente in Gruppen zusammenfassen 349
2 Toolbars und Symbolleisten entwerfen am Beispiel des TreeDesigners 604
3 Fenster mit frei verschiebbaren Trennbalken in Bereiche einteilen 325
4 Eine scrollbare Zeichenfläche bereitstellen 674
5 Property-Einstellungen aus Beispielprogrammen rekonstruieren 606
6 Resourcen in die ausführbare Datei einbinden 59
7 Klassen von Komponenten austauschen 363
8 Ungewöhnliche Formularänderungen im Textmodus durchführen 357
9 Eine einfache Komponentenschablone erstellen 126
10 Bestehende Formulare auf Vererbung umstellen 402
11 Ein Hauptmenü als verschiebbare Toolbar anzeigen 972
12 Gleiche Aktionen für Menüpunkte und Schalter gemeinsam verwalten 485
13 Mehrere Ereignisquellen in einer Methode bearbeiten 352

2 Formulare zur Laufzeit

14 Mauszeigerformen dynamisch anpassen 332
15 Eigene Mauszeigerformen verwenden 330
16 Koordinatensysteme von Komponenten umrechnen 323
17 Die Formularaufteilung bei Größenänderung automatisch anpassen 325
18 Größenveränderliche Formulare mit fester Raumaufteilung programmieren 341
19 Größenveränderung eines Formulars verhindern 342
20 Steuerelemente in Arrays verwalten 355
21 Vorgegebene Steuerelement-Arrays nutzen 356
22 Z-Reihenfolge von Komponenten zur Laufzeit anpassen 321
23 Menüs und Aktionen zur Laufzeit aktivieren und deaktivieren / 1 615
24 Menüs und Aktionen zur Laufzeit aktivieren und deaktivieren / 2 486
25 Menüpunkte zur Laufzeit aktivieren und deaktivieren / 3 492
26 Menüs dynamisch aufbauen 468
27 Mehrseitige Dialoge erstellen 359
28 Alternative Seitenauswahlverfahren in mehrseitigen Dialogen bereitstellen 361
29 Dialogeingaben als Datenstrukturen zurückgeben 148

30 Dialogeingaben überprüfen 127
31 Verwenden einer Steuerelementgruppe in verschiedenen Fenstern....728
32 Icon/ListView-Kombination mit vier Darstellungsmodi aufbauen 383

3 Formulare: Management mehrerer Formulare

33 Formulare dynamisch erzeugen....346
34 Formulare freigeben....197
35 Modale Dialogfenster erzeugen....124
36 MDI-Kindfenster erzeugen....710
37 Hauptmenüpunkte im Kindfenster bearbeiten....715
38 Ereignisse an MDI-Kindfenster weitergeben....727
39 Nicht-modale Fenster schließen....348
40 Den Wechsel zwischen MDI-Kindfenstern bearbeiten....720
41 MDI-Kindfenster beim Start öffnen und ein Dokument laden....711
42 Symbolleisten in ein anderes Formular einblenden....724
43 Unabhängige Formulare durch Interfaces aufeinander abstimmen....718
44 Komponenten anderer Formulare direkt verwenden....725
45 Das Dokument-View-Konzept realisieren....618
46 Fenster-Menü: anhängen und bearbeiten....713
47 Andocken von Hilfsfenstern an ein Hauptfenster....739

4 Benutzerschnittstelle: Komponenten

48 Bilderlisten zur Laufzeit aufbauen....388
49 Die THistoryList-Beispielkomponente verwenden....476
50 Listbox-Elemente per Drag&Drop umsortieren lassen....140
51 Datenelemente in einer Listbox speichern....148
52 Eine Listenkomponte für Schriftauswahl erzeugen....919
53 Verzeichnisbaum und Dateiliste anzeigen (fertige Komponenten)....966
54 Icons im IconView per Drag&Drop verschieben....380
55 ListViews als Eingabekomponenten....160
56 Große TreeView-Strukturen effizient aufbauen....393
57 Datenobjekte im IconView speichern....382
58 Listbox-Einträge editieren und löschen (mit Mehrfachauswahl)....135

5 Features

59 Hinweistexte zu Komponenten in der Statuszeile anzeigen....155
60 Popup-Menüs für Grafikbestandteile bereitstellen....745
61 Drag&Drop-Funktionen programmieren....750
62 Sicherheitsabfragen zum Dateispeichern durchführen....613
63 MDI-Anwendungen erzeugen....702

64 History-Listen verwalten und speichern 414
65 History-Listen in Komboboxen anbieten (Beispielkomponente) 922
66 Einen Tastenkürzeleditor in die Anwendungen einbauen 962
67 Dateilisten zur Laufzeit an Menüs anhängen 473
68 Sekundärfenster schließen, verstecken und wiederanzeigen 346
69 Freien Festplatten- und Hauptspeicher ermitteln 542
70 Die Klassen in Quelltextdateien per Shell-Skript finden 559
71 Die Ausgabe von externen Konsolenanwendungen abfangen 553
72 Externe Programme ausführen (RunProgram-Funktion) 557
73 Zwischenablageoprationen mit Standardaktionen implementieren 489
74 Bestätigungsmeldungen durch den Benutzer unterdrücken lassen 969
75 Anwendungsspezifische Daten über die Zwischenablage kopieren 493
76 Laden und Verknüpfen von Bibliotheksfunktionen zur Laufzeit 586
77 Anwendungen in das KDE-Klassen-Framework einbinden 583

6 Benutzereingabe mit Tastatur und Maus

Tastatur und Maus

78 Den Fokus per Mausklick steuern 735
79 Benutzereingaben simulieren 736

Tastatur

80 Tastatureingaben für Komponenten bearbeiten (Beispiel: Listboxen) 137
81 Den Tastaturfokus steuern 336
82 Tastatureingaben an andere Komponenten weiterleiten 736
83 Eine ScrollBox per Tastatur steuern 735

Maus

84 Mauseingaben in virtuelle Koordinatensysteme akzeptieren 689
85 Die grundlegenden Tastaturereignisse bearbeiten 327
86 Tastaturereignisse global für das Formular abfangen 738
87 Mauszeichnen-Aktionen programmieren 651
88 Angeklicktes Element einer Zeichnung herausfinden 656
89 Selbst definierte Markierungspunkte zur Größenänderung behandeln 657

7 Grafikausgabe

90 Speicherbitmaps erzeugen 452
91 Eine Liste von kleinen Vorschau-Bitmaps erzeugen 461
92 Polygone zeichnen 663

93 Temporäre Umrisse zeichnen 653
94 Tabellenkomponenten (TDrawGrid) selbst zeichnen 447
95 Listbox-Einträge selbst zeichnen 458
96 Text bündig oder zentriert ausrichten 664
97 Unnötiges Zeichnen vermeiden 669
98 Das Neuzeichnen von Komponenten veranlassen 334
99 Bildschirmaktualisierung per Ungültig-Erklärung 660
100 Grafik an selbst definierten Begrenzungen abschneiden (Clipping) 676
101 Die zoombare Grafikausgabe des TreeDesigners. 686
102 Grafikausgabe in einen Hilfs-Thread auslagern 509
103 Folgen von Grafikoperationen speichern (Metadateien) 462
104 Grafikausgabe mit der Qt-Bibliothek. 572
105 Die Beispielklasse TVirtualCanvas verwenden 681
106 Virtuelle Koordinatensysteme verwenden 682
107 Grafikausgabe in selbst gezeichneten Komponenten 939

8 Drucken

108 Text drucken 691
109 Einfache Grafiken drucken 692
110 In WYSIWYG-Qualität drucken 693
111 Einen Druckvorschaudialog realisieren 694
112 Drucken in Millimetern 700

9 Ein- und Ausgabe: Dateien

Dateien allgemein

113 Dateien öffnen und beschreiben (Object-Pascal-Ebene) 283
114 Objekte und Dokumente in Dateien speichern (mit Streams) 640
115 Polymorphe Objekte speichern 427
116 Verbindungen zwischen Objekten (Zeiger) speichern 649
117 XML-Dateien erzeugen 771
118 Das Heimatverzeichnis ermitteln. 563
119 Verzeichnisse nach Dateien durchsuchen 281
Memory-Streams verwenden, siehe R75. 493

onfigurationsdateien

120 Die Benutzereinstellungen in Dialogen automatisch speichern 421
121 Fenstergröße und -Position zwischen Programmläufen speichern 424
122 Ini-Sektionen verarbeiten 419
123 Speichern von Docking-Einstellungen 742
124 ListBox-Inhalte speichern und wiederherstellen 138

10 Datenbanken – ausgewählte Themen

125 Lookup-Felder definieren . . . 824
126 Haupt-Detailformulare gestalten . . . 851
127 Datenbank-Updates zentral durchführen . . . 869
128 Automatische Nummerierung von Datensätzen . . . 856
129 Datensatzfelder initialisieren und formatiert anzeigen . . . 863
130 Die Details eines Haupt-Detailformulars sortieren . . . 852
131 Die Tabellenstruktur zur Laufzeit abfragen . . . 828
132 Datensätze im Programm erzeugen . . . 834
133 Langwierige Datenbankoperationen abbrechen lassen . . . 837
134 Einen SQL-Monitor in die Anwendung einbauen . . . 882

11 Komponenten – ausgewählte Themen

135 Komponenten in einer Testumgebung testen . . . 931
136 Formulare in Komponenten kapseln . . . 960
137 Erweiterbare Popup-Menüs bereitstellen . . . 944
138 Entwurfszeit-Editoren für Komponenten bereitstellen . . . 950
139 Property-Editoren erzeugen . . . 947
140 Neue Standardaktionen definieren . . . 958
141 Mehrere Komponenten zu einer Komponente kombinieren . . . 928
142 Property-Kategorien verwenden . . . 901
143 Komponenten für extern definierte Widgets erzeugen . . . 581

12 Programmiertechnik

Datenstrukturen

144 Properties selbst definieren . . . 187
145 Typumwandlungen sparen mit Typumwandlungs-Properties . . . 629
146 Einfache Methoden durch Nur-Lesen-Properties ersetzen . . . 628
147 TStrings mit Objekten assoziieren . . . 412
148 Dynamische Listen, Teil 1: eine Bitmap-Liste aufbauen . . . 456
149 Dynamische Listen, Teil 2: die Bitmap-Liste freigeben . . . 460
150 Gleichzeitigen Datenzugriff aus mehreren Threads synchronisieren . . . 514

OOP

151 Schreiben eines Object-Pascal-Interfaces für eine C++-Klasse . . . 578
152 Klassenvariablen simulieren . . . 208
153 Die Klasse eines Objekts zur Laufzeit festlegen . . . 205
154 Abstrakte Klassen erzeugen . . . 204

155 Beispiel zum dynamischen Aufbau von Objekten 638
156 Interfaces in neuen Klassen implementieren 273
157 Bestehende Klassen um Interfaces erweitern 277

Erweitertes Pascal

158 Typen und Variablen umwandeln 238
159 Schleifen über Aufzählungstypen schreiben 218
160 Funktionen mit variablen Parameterlisten schreiben 250

Debugging

161 Mit dem CLX-Quelltext debuggen.....119
162 Diagnosemeldungen ausgeben.....110

Programmfluss

163 Eine dynamische Anzahl von Threads verwalten.....523
164 Unterbrechbare Threads mit einer sicheren Suspend-Funktion implementieren.....520
165 Auf das Ende von Threads reagieren 525
166 Rechnerleerläufe für Hintergrundaufgaben benutzen mit OnIdle.....499
Simulation von Ereignissen: siehe R79 736

Anhang B: CLX-Hierarchiegrafiken des TreeDesigners

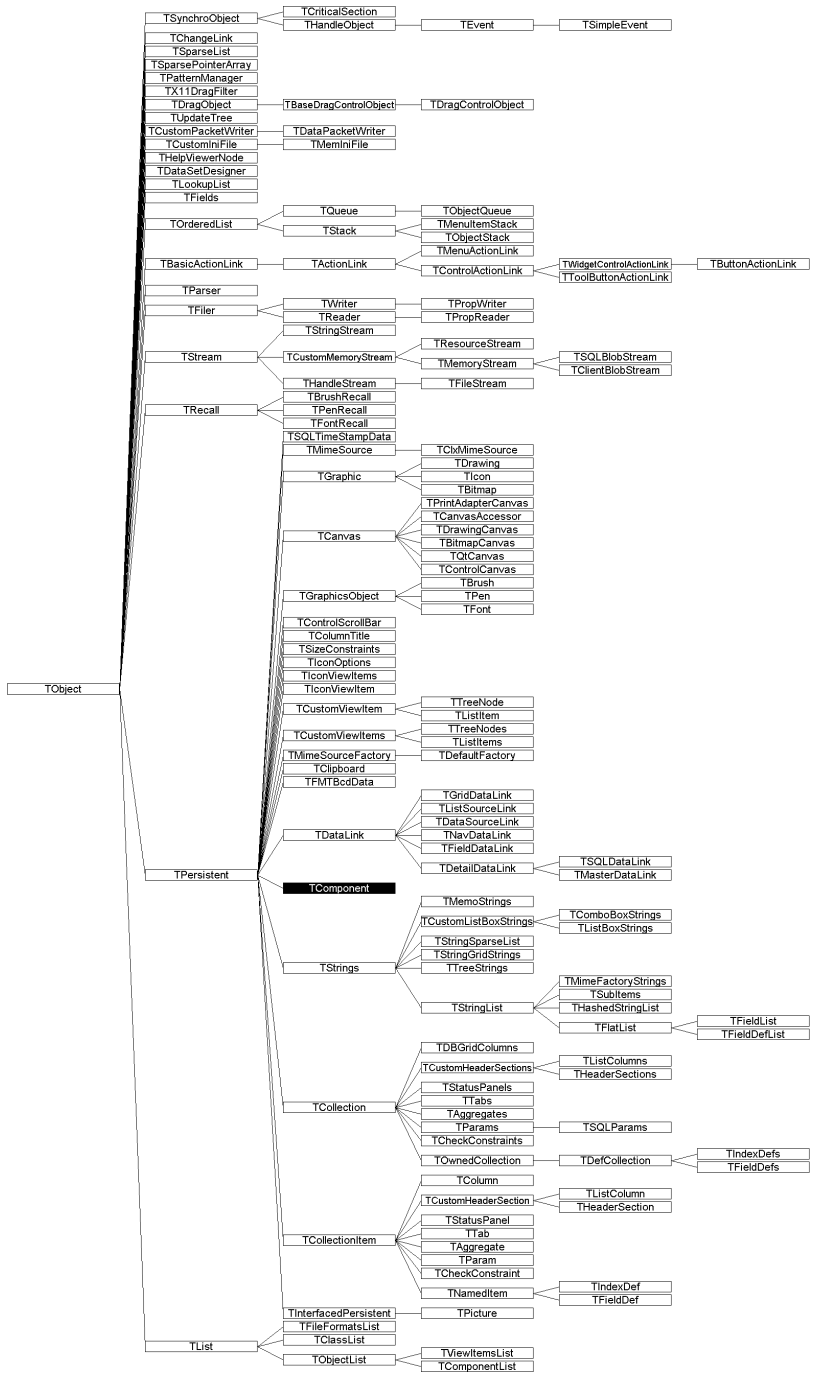


Abbildung B.1: Die »Wurzelregion« der CLX-Klassenhierarchie

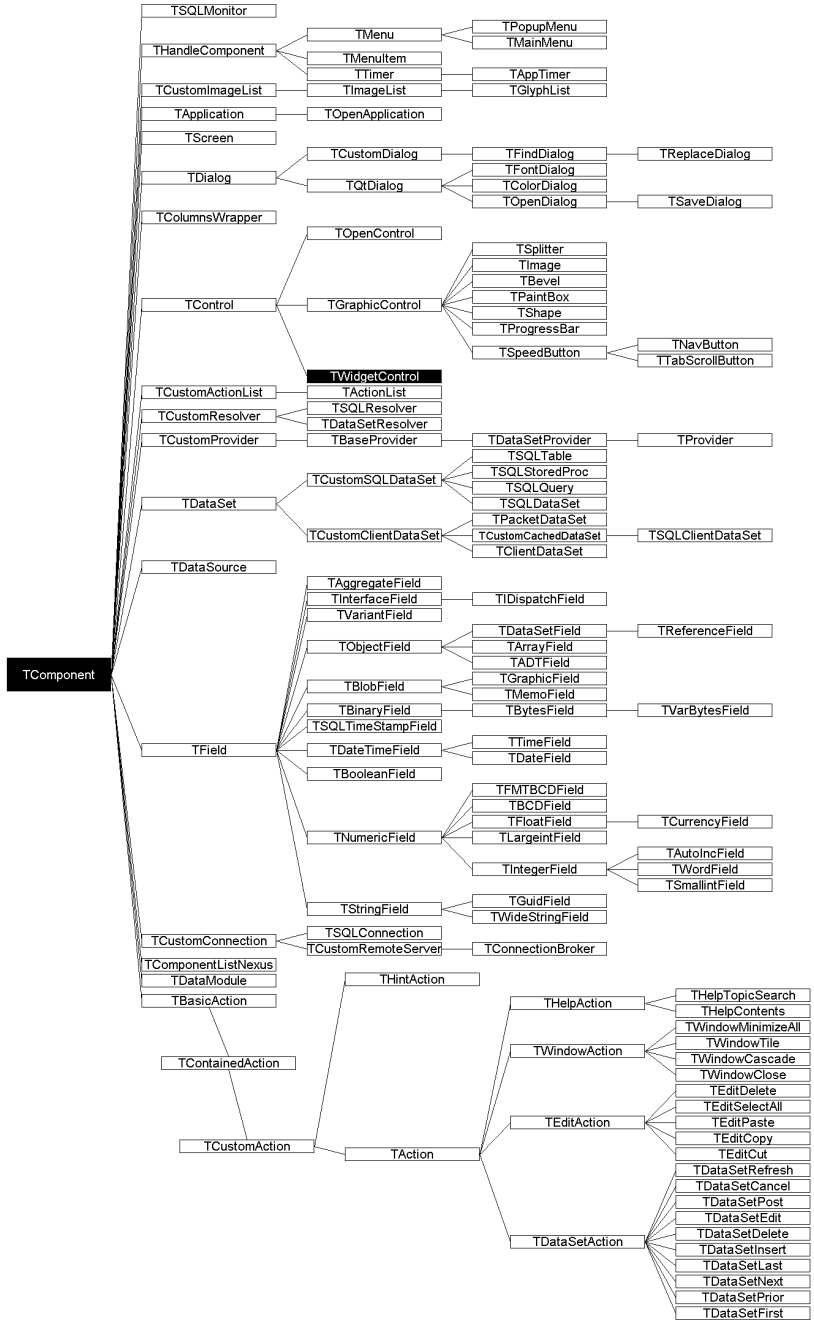


Abbildung B.2: TComponent bildet die Grundlage für alle Komponenten, sowohl visuelle als auch nicht-visuelle.

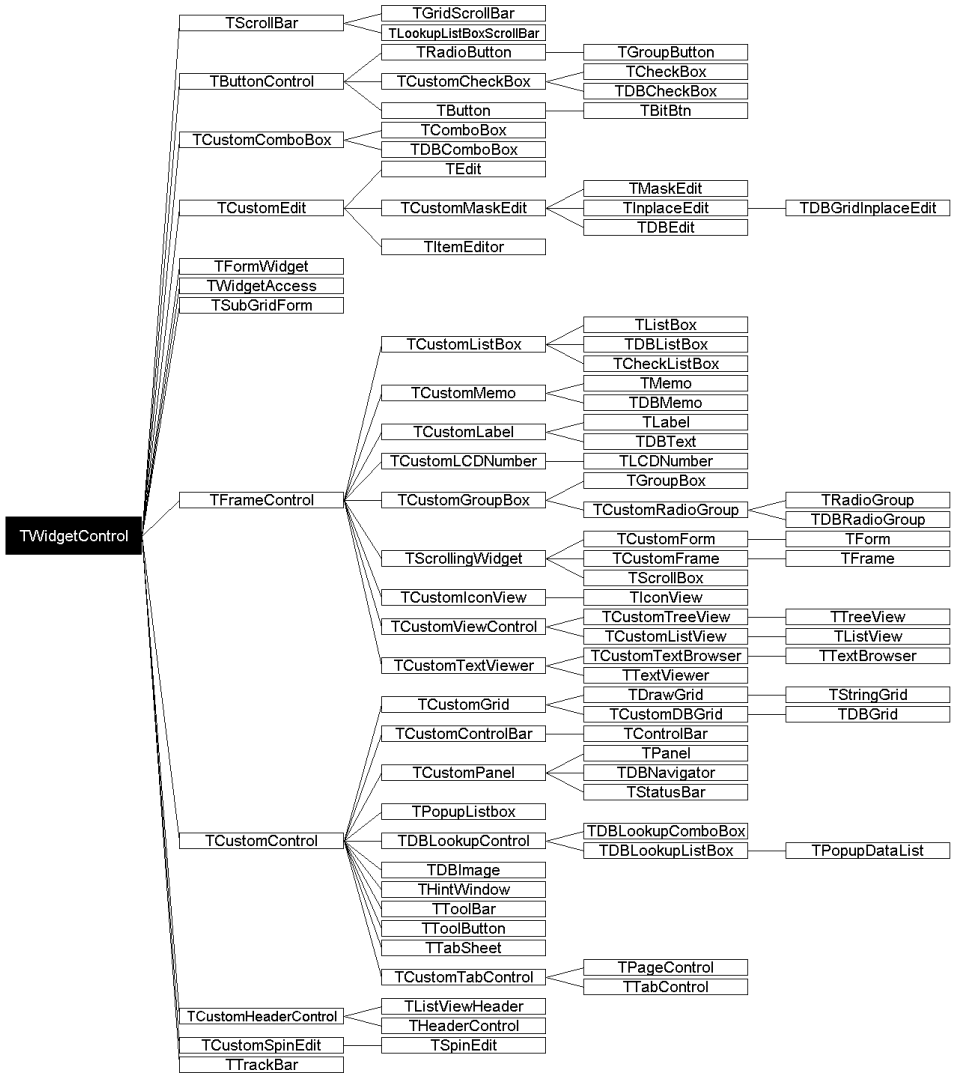


Abbildung B.3: Der in Abbildung B.2 ausgeblendete Teilbaum der TWidgetControl-Komponenten

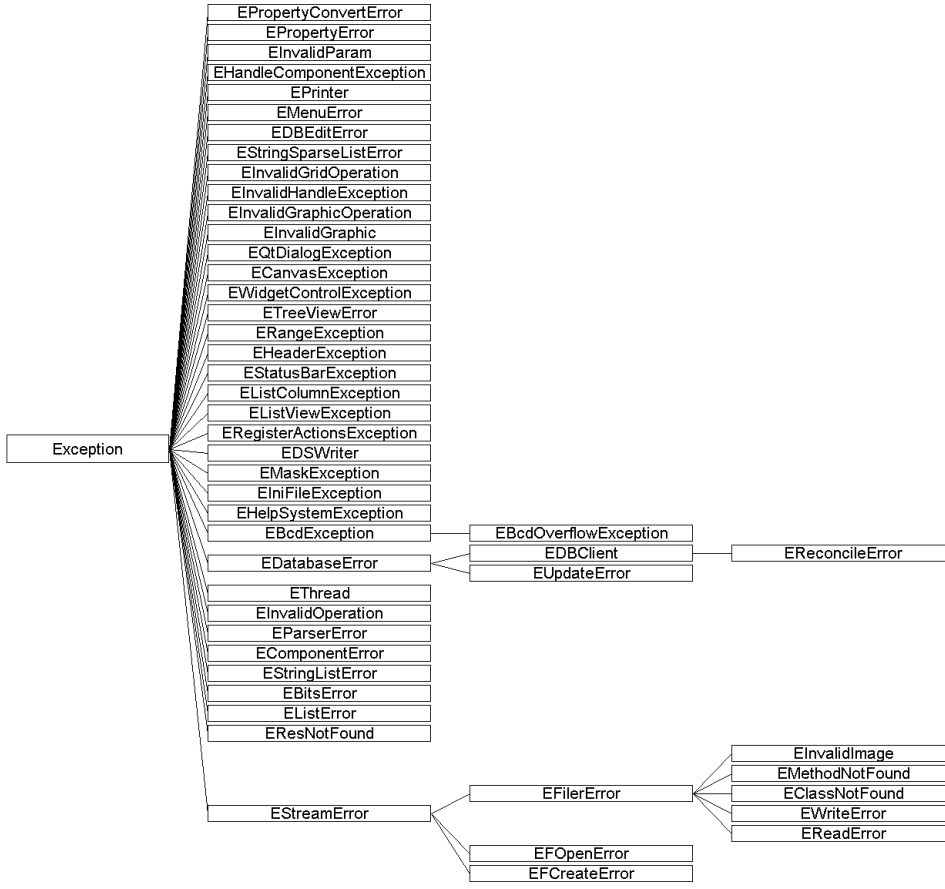


Abbildung B.4: Die Exceptions-Klassen sind in einer eigenen kleinen Hierarchie angeordnet (Exception selbst ist von TObject abgeleitet).

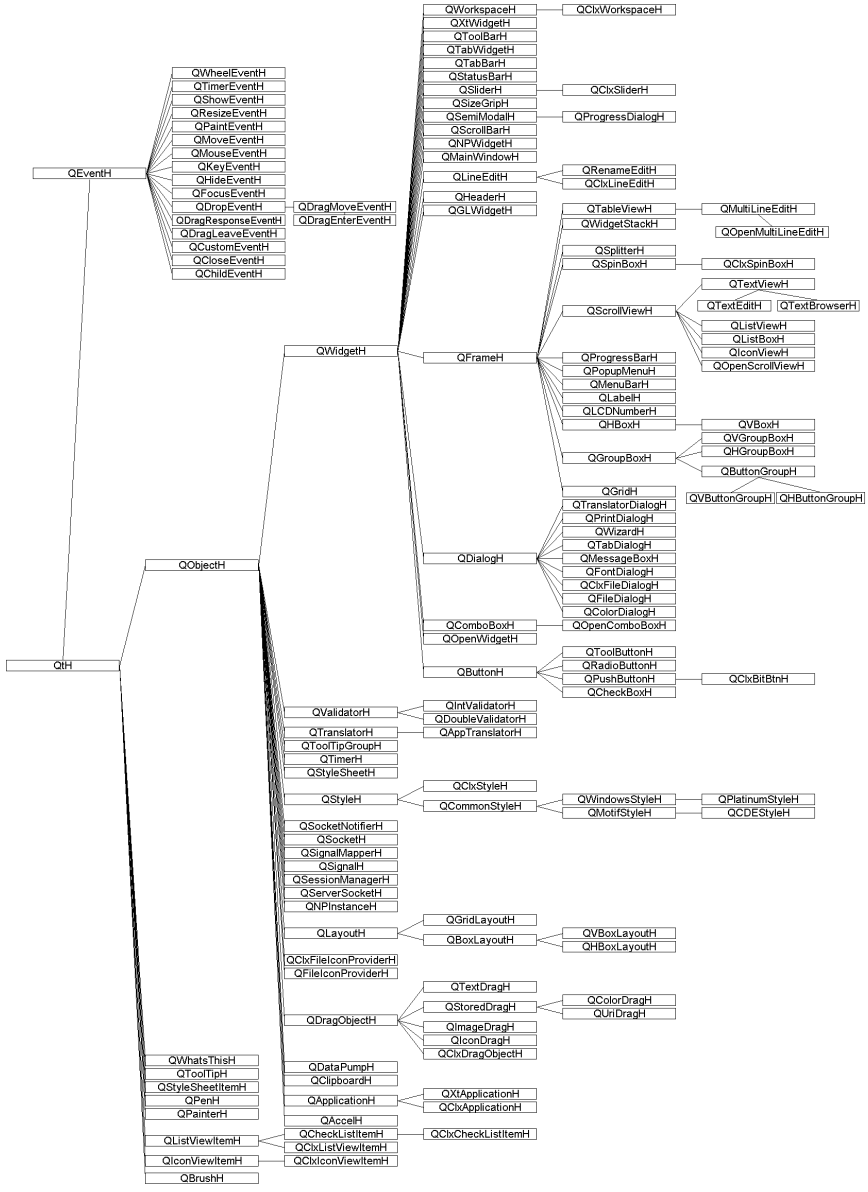


Abbildung B.5: Die Qt-Klassenhierarchie spiegelt sich in der Hierarchie von QtH-Klassen wider, die keine Elemente haben, sondern nur der Typüberprüfung durch den Compiler dienen (siehe Kapitel 5.4.2).

Index

Das Stichwortverzeichnis enthält einige spezielle Punkte, auf die hier besonders hingewiesen werden soll:

- Klassenelemente sind grundsätzlich als Unterpunkt der Klassen aufgeführt, durch die sie definiert werden. Da jedoch die Elemente der grundlegenden Klassen wie beispielsweise *TControl* und *TWidgetControl* beim Einstieg in Kylix schwer zu unterscheiden sind und weitere Elemente in verschiedenen Klassen unabhängig eingeführt werden, wurde das Stichwort *CLX – allgemeine Klassenelemente* gewählt, um diesen Elementen ein gemeinsames Dach zu geben. Sollten Sie weder dort noch bei einer speziellen Klasse fündig werden, empfiehlt sich das Nachschlagen unter *TControl* oder *TWidgetControl* oder der Aufruf der Online-Hilfe von Kylix, die meistens die Klassen auflistet, in denen das gesuchte Klassenelement vorkommt.
- Drei Stichwörter fassen andere Stichwörter zusammen, die auch als eigenständige Punkte des Verzeichnisses aufgeführt werden: *CLX* fasst die Klassen zusammen, die an einer bestimmten Stelle ausführlich beschrieben sind, während Sie die Klassen, die eher kurz erwähnt werden, im Verzeichnis nur als eigenständige Stichwörter finden. In *Beispielprogramme* finden Sie schließlich eine Übersicht über die Stellen, an denen namentlich genannte Beispielprogramme besprochen sind.
- Alle Stichwörter, die speziell dem Begriff *Steuerelemente* zuzuordnen wären, wurden der Eindeutigkeit halber dem Punkt *Komponenten* untergeordnet. Weitere zusammenfassende Stichwörter sind *IDE* und *Datenbanken*.
- Falls Sie sich intensiver mit der Beispielanwendung *TreeDesigner* beschäftigen und Informationen zu einer der darin definierten Methoden suchen, sollten Sie unter den Punkten *TGraphicDoc*, *TGraphicElement* und *TDocumentForm* nachschlagen.

Symbols

642
$ALIGN 538
$B-Anweisung 221
$D-Anweisung 173
$define-Anweisung 174
$EXTERNALSYM 534
$H-Anweisung 226, 227
$I-Anweisung 262, 285
$if und $ifend 174
ifdef-Anweisung 174
$J-Anweisung 177
$L-Anweisung 173
$MINENUMSIZE 537
$P-Anweisung 247
$Q-Anweisung 261
$R-Anweisung 59, 66, 173, 217, 261
$SONAME u.$%$a. 547
$X-Anweisung 244
$Y-Anweisung 173

Numerics

64-Bit-Architektur 234

Abbildungsmodi
von TVirtualCanvas 683
ABI 564
Ablageobjekt 430
Ableiten 191
abstract 204
Addr 235, 289
Adressoperator 220, 235
nderungsflag 611
nderungsprotokoll 871
Aktenkoffermodell 780, 817, 874
Aktionslisten 483
Fenster-Standardaktionen 713
Standardaktionen 486
Tastenkürzel 478
Tastenkürzeleditor 962
TreeDesigner-Beispiel 615
Aktionsziele 487
AlarmClockX
siehe Beispielprogramme 124
AlignmentFlags 665
alNone und andere 74, 324
and 219, 220
AnsiChar 216, 225
ANSI-Code 170
AnsiCompareStr 287
AnsiCompareText 287
AnsiLowerCase 287
AnsiString 225, 227
AnsiUpperCase 287
Anwendungen
außerhalb der IDE starten 530
Datenbank~ 797
dialogorientierte 590
dokumentorientierte 590
Entwicklungszyklus 27
neu kompilieren 93
AppBrowser 83
Append 284
Application 312
ApplicationAndBorderStyles 355
ApplyUpdates 816, 836, 857, 870
Arbeitsbereich 25
Array-Properties 188
Arrays 222
Array-Konstruktor 248
dynamische 223
offene 247
von C übersetzen 539
von Steuerelementen 355
ASCII-Code 170
as-Operator 211, 353
bei Interfaces 271
Assertions 119
Assigned 195, 236, 289
AssignFile 284
AssignPrn 691
Aufrufkonventionen 245
Aufruf-Stack (Debugger-Fenster) 118
Aufzählungstypen 217
im Objektinspektor 50
von C übersetzen 536
Ausdrücke
Auswertungsreihenfolge 221
boolesche 221
reguläre 561
Ausgabeumleitung 553

B

Bäume 645
speichern 649
bash 561
BashRCEditor 29
Ereignisbearbeitung 67
Formularentwurf 42
BashRCEditor2 74
Basic-Sprachen
Unterschiede zu Object Pascal 168
Basisklasse 191
Bedingungen 240
Befehlszeilenparameter 712
begin 66, 179
Beispielprogramme
AlarmClock(1-5) 121

AlarmClock1 124
AlarmClock2 131
AlarmClock3 134
AlarmClock4 145
AlarmClock5 154, 382, 386
ApplicationAndBorderStyles 351, 355
BashRCEditor 29, 42, 67
BashRCEditor2 74
CanvasMagnetism 442
cdsBrowser 809
ClipboardViewer 490, 492
ConfirmDemo 970
CoordinateSystemTestForm 679
DBBrowse 827
DragDropDemo 749
DrawGrid 447
FileDB 832, 848
FileSystemBrowsersTest 967
FixedSizeFrm 342
gdbBrowser 811
GrDoc (Unit) 626
HistoryList-Demo (HlTest) 468
HList 470
HlTest 468, 927
InterfaceDemo 267, 273, 276
khtml_wrapper.cpp 578
ListViewIconView 375, 377
LookupFieldDemo 826
MessageDemo 308
Metafile 462
MiniSysMon 540
OwnerDraw1 455
OwnerDraw2 461
PageControlNotebook 359, 738
PolyStrm 428
ResTestMax 347
RunPrograms 555, 559, 593
StreamUtil (Unit) 429
SuspendableThread 518
SyncDemo 511, 520
TColorPalette 931
TEdge 645
TerminDB 853
TextPrinter 691
TFontComboBox 919
TGraphicDoc 626
TGraphicElement 633
THistoryCombo 922
THistoryList 414, 470, 471, 923
THotkeyManager 960
ThreadDemo1 341, 497, 509, 518
TLabelEditCombi 928
TNumEdit 898
TPaletteComponentEditor 950
TPaletteEditor 948
TreeDesigner 600, 686
TreeDesignerSDI 600, 704
TScrollBoxEx 675, 734, 920
TSound 523
VCanvas 680
WaveProcessingThreads 521
WorldXForm 573
XWindowCall 545
Benutzername
SYSDBA 793
Bereichsüberprüfung 261
Besitzhierarchie 308
Bezeichner 64
Aufbau 170
Bibliotheken
finden 529
Suchpfad 36
Verzeichnisse 529
zur Laufzeit laden 586
Bildeditor
Ersatz 57
Bildkomponenten
Namenstabelle 454
Bildschirm
Auflösung 318
Höhe und Breite 318
Bildschirmkoordinaten
errechnen 323
biSystemMenu und andere 343
Bitmaps 451
Beispielprogramm 455
editieren 58
Formate 58
siehe auch TBitmap 452
verkleinern 461
verwenden 58
vordefinierte 158
BLOBs 802, 803
blockread 283, 284
blockwrite 284
Boolean 218
Borland Pascal
Objektmodell 213
bpg-Dateiendung 95
break 243
Breakpoints 110
Aktionen 114
Durchlaufzähler 113
Gruppen 113
Gruppen temporär deaktivieren 114
Optionen 112
setzen 112
temporäre 111
briefcase model
siehe Aktenkoffermodell 780

Browser
der Kylix-IDE 100
bs...-Konstanten 352
bsDialog und andere 340
Build 93
Byte 214
ByteBool 218

Aufruf von Object Pascal-Funktionen 547
Typen in Object Pascal 535
C++ 166
Klassendeklarationen in Object Pascal 565
Methoden über C-Funktionen aufrufen 566
Methodenaufrufe aus Object Pascal 564
caHide und andere 348
CancelUpdates 808, 816
CanvasMagnetism (Beispielprogramm) 442
Cardinal 214
case 242
bei Records 233
cbGrayed 143, 354
C-Bibliothek
von Linux 528
Cdecl 245
cdecl 532
C-Deklarationen
nach Object Pascal übersetzen 532
CD-ROM
zum Buch installieren 885
Char 216
char (C-Typ) 536
Chr 289
class 183
class of 207
Classes 426
Client-Bereich
eines Fensters 25
Client-Datenmengen
siehe TClientDataSet 780
Client-Koordinaten
umrechnen 323
Clipboard 488
Clipping 436, 676
Close 284, 691
CLX 291
DataSet 798
Hierarchiebaum 982, 983, 984, 985
IDesignerHook 916
Kompatibilität mit der VCL 293
Mime-Typen 492
Multithreading 505
Printer 699
Qt-Signals 305
Quelltext 35
TActionList 484
TApplication 312
TBevel 162
TBitmap 452
TButton 129
TCanvas 434
TCheckBox 143
TClipboard 493
TColorDialogs 370
TComboBox 132, 142
TControl 302, 320
TControlBar 607
TControlScrollBar 338
TCustomEdit 372
TCustomForm 339
TDataSource 798
TDBGrid 802
TDBNavigator 804
TDrawGrid 447
TDrawing 462
TField 817
TFiler 430
TFileStream 424
TFindDialog 371
TForm 344
TGraphic 449
TGraphicControl 302, 303
THandleStream 424
THeaderControl 152
TIcon 450
TIconView 380
TImage 151, 453
TImageList 376, 388
TIniFile 418
TList 416
TListBox 134
TListView 157
TMainMenu 465
TMemIniFile 418
TMemo 374
TMemoryStream 433
TMenu 465
TObject 209, 297
TOpenDialog 705
TPageControl 359
TPen 437
TPersistent 299
TPicture 452
TPopupMenu 465
TPrinter 692, 699
TProgressBar 156
TRadioButton 143
TRadioGroup 143
TReader 430

TReplaceDialog 371
TSaveDialog 369
TScreen 317
TScrollBar 163
TShape 151
TSplitter 325
TStatusBar 155
TStream 424
TStringList 410
TStrings 410
TTabSheet 359
TThread 507
TThreadList 521
TToolBar 604
TTreeView 390
TWidgetControl 302, 320, 336
TWriter 430
Überblick 296
CLX – allgemeine Klassenelemente
Action 485
ActiveControl 48
ActivePage 360
ActivePageIndex 360
Align 74, 324, 603
Alignment 48
AllowAllUp 607
Anchors 335
ApplyUpdates 816, 870
Assign 449, 454
AutoSize 48, 335
BorderIcons 51
BorderStyle 352
BringToFront 64, 321
Brush (TWidgetControl) 333
Canvas 435, 609
Caption 48, 54, 324
Close 197
Color 61, 333
ConnectionName 794
Constraints 334
Controls 357
CopyToClipboard 489
CreateDataSet 783
CreateWidget 294
Cursor 51
CutToClipboard 489
DataField 802
DataSource 802
DblClick 908
DragMode 381, 747
DrawItem 920
Enabled 320, 838
EventFilter 306, 908
Execute 368, 960
FieldByName 818
FieldDefs 783
Font 51, 333
Free 195, 297
Glyph 58
Hide 321
Hint 156, 333
HookEvents 305
Invalidate(TControl) 687
Koordinaten 322
Koordinatenumrechnung 323
Loaded 423
LoadFromFile 67
ModalResult 127, 340
OnChange 69, 146, 637
OnClick 56, 60, 62, 68, 131
OnClose 712
OnCloseQuery 76, 712
OnCreate 67, 193
OnDblClick 137
OnDblClk 62
OnDrawItem 446
OnEnter 327
OnExit 327
OnKeyDown 70, 138
OnMeasureItem 457
OnMouse... 651
OnMouseDown 62, 70
OnPaint 660
Pages 360
Painting 920
Parent 711, 717
ParentFont 50
ParentShowHint 156
PasteFromClipboard 489
ReadOnly 807
SaveToFile 68
Scrollbars 48
SelText 373
SendToBack 322
SetFocus 735
Show 321
ShowHint 156
Showing 321
TabOrder 42
TabStop 49
Tag 300, 352, 355
Text 324, 372
Tracking 675
Value (von Feldern) 830
Visible 321
CLX-Quelltext
im Debugger 119
Code Insight 69, 86
CodeExplorer 83
CoDomDocument 767

ColorPalDesignTime (Beispielunit) 902, 952
COM 264, 270
  Referenzzählung 272
Common Controls 37
Comp 216
CompareStr 227, 288
CompareText 288
Compiler
  aufrufen 61
  Aufrufoptionen 62
  Optimierung 105
  vordefinierte Symbole 174
Compileroptionen
  allgemein 172
  Assertions 120
  Debugger 105
  Exceptions 261
Compilierung
  bedingte 173
const 247
Constraints 334
constructor 194
contains-Klausel 896
continue 243
Contnrs (Unit) 417
Copy 288
crCross und andere 51
creat (Libc-Funktion) 532, 534
CreateTreeDesignerTree 768
Critical Sections
  siehe TCriticalSection 514
csDesigning 916, 964
csLoading 965
CSS 757
Currency (Typ) 215
Cursor
  in Datenmengen 805
Cursors-Array 330

Data Pump-Utility 788, 849
DataSet 798
Dateiattribute
  in TFileTable 967
Dateidatenbank
  Indizes 848
Dateien
  binäre (Beispiel) 150
  herkömmliche Funktionen 284
  INI- und Konfigurations~ 417
  typisierte 283
  untypisierte 283
Dateitypen 283
Dateiverwaltung 281
Dateiwahldialoge
  Einstellungen 369, 370
Daten und Code
  in Objekten 182
Datenbanken
  Änderungsprotokoll 815
  Aggregat-Felder 818
  Aktenkoffermodell 780
  aktualisieren 870
  Anwendungen installieren 794
  Anzeigeformatierung 821
  automatische Navigation 802
  BeforeUpdateRecord 877, 878
  Bereiche filtern 846
  blättern zur Entwurfszeit 824
  Cursor 805
  DataSource und DataField 802
  Daten-Pipeline 798
  Desktop~ konvertieren 788
  Editieren 806
  eindeutige Nummern vergeben 856
  Felder 817
  Felder-Editor 822, 824
  Feldzugriff 830
  FieldDefs 826
  Gerüst 798
  Grundbegriffe 778
  Grundkonzepte der Programmierung 805
  Haupt-/Detail-Formulare 851
  in Client-Datenmengen suchen 850
  Indizes 843
  Joins 861
  Lesezeichen 843
  Login 793
  Lookup-Felder 824
  manuelle Updates 878
  mehrschichtige Anwendungen 796
  Metadaten 785
  mit isql erzeugen 785
  modale Dialoge 866
  Modus 805
  modusabhängige Suchmethoden 845
  Navigation 806
  Normalisierung 779
  OnReconcileError 875
  Operationen im Hauptspeicher 840
  Parameter für Anfragen 862
  persistente Felder 822
  physikalische Struktur 779
  Positionszeiger 805
  Post 808, 814, 866
  Provider 796
  Reconcile-Dialog 875
  referentielle Integrität 790

Schreibschutz 807
Sortieren einer Detailtabelle 852
Sortierreihenfolge 844
speichern 815
Suchmethoden 841
Suchmethoden von TTable 845
Tabellen erzeugen 787
Tabellenformate 820
Tabellenliste ermitteln 813
TDBImage 829
Update-Konflikte 875
Verbindung wechseln 811
Verbindungen definieren 792
Zugriffsvariationen 795
Datenbankoberfläche 781
Datenformate
in XML 753
Datenmengen 778
unidirektionale 791, 796, 841
Datenmodule 800
initialisieren 864
Datenquellen
siehe TDataSource 794
Datensätze
numerieren 857
nummerieren 856
OnNewRecord 864
Datensatz
aktueller 805, 829
dBase 788
DBBrowse 827
dbExpress 777
dbxconnections 792
DCR-Dateien 914
DCU-Dateien 90
Debug-Desktop 34
Debugger 104
Adress- und Datenhaltepunkte 113
Assertions 119
Ausdrücke 116
Auswertung durch Kurzhinweis 87, 115
Compileroptionen 105
CPU-Fenster 107
Drag&Drop 115
Einzelschritte 117
Ereignisprotokoll 108, 114
Formatanweisungen 116
FPU-Fenster 107
Funktionsaufrufe 116
Inspektorfenster 116
Modul-Fenster 106
Prozessorregister überwachen 116
Variablen ändern 117
Variablen untersuchen 114
Variablenfenster 115
Dec 231, 261, 289
Declared 175
default 912, 917, 918, 954
Defined 175
Deklaration
von Formularen 65
Deklarationsblöcke 175
Delegation
von Interfaces 275
Dereferenzierung 236
DesignEditors (Unit) 947, 950
DesignerHook 916
designide.dcp (Package) 947, 950
DesignIntf
Kategorienamen 902
desk-Dateien 34, 418
destructor 196
Destruktoren 195
automatischer Aufruf 196
DFM-Dateien 90
Diagnosemeldungen
ausgeben 110
Dialoge
Gliederung 358
Initialisieren mit Records 148
Klassen für ihre Daten 146
mehrseitige 358
modale 124
Seitenauswahlverfahren 361
seitenübergreifende Elemente 360
Standarddialoge 366
Tastaturbedienung 48
Tastatursprünge 49
Dialogelemente
Eingaben abfragen 145
Dialogelemente (siehe Steuerelemente) 26
div 219
dlopen (Libc-Funktion) 586
dlsym (Libc-Funktion) 586
dmAutomatic 747
dmManual 747
DMTs 203
DockFormBase (TreeDesigner-Unit) 739
Docking 738
ausschalten 33
Einstellungen speichern 742
Implementierung 743
DocListForm (TreeDesigner-Formular) 601
do-Klausel 259
Dokumente
im TreeDesigner 617
Dokumentverwaltung 590, 611
im Dok-View-Konzept 620
Speichern 610

Dokument-View-Konzept 617
Dokumentfreigabe 621
Fenstertitel 624
Unabhängigkeit der Fenster 668
View-Verwaltung 623
DOM 757
Attribute abfragen 770
Baumstruktur durchlaufen 769
Dokumentelement 760
Knotennamen 760
Knotentypen 759
DOM-Klassen
siehe IDom... 758
Double 216
double (C-Typ) 536
DPR-Dateien 90
DPtoLP 689
Drag&Drop 746
auf Komponententeile 752
Eigenschaften verändern 748
Ereignisse 746
manuelles 751
mit rechter Maustaste 752
Objekte aufnehmen 748
Starten 747
von bestimmten Quellen 751
von Dateien 748
zur Nachrichtenübermittlung 748
DrawGrid (Beispielprogramm) 447
Drucken 690
Grafik 692
in Millimetereinheiten 700
in PostScript 692
Seitengröße 700
Seitenwechsel 690
Text 691
WYSIWYG 693
Drucker
Beschränkungen 691
Druckereinrichtung 697
Druckvorschau 694
dsInactive und andere 848
dsMotif und andere 315
dst-Dateien 34
DTD (XML) 756
dup2 (Libc-Funktion) 553
dynamic 203

EAccessViolation 192, 228
EAssertionFailed 121
EConvertError 161, 262, 288
Editierfelder 372
Editor 83
CodeExplorer-Optionen 85
Hyperlinks 84
Navigation 84
Programmierhilfen 69, 86
Quelldateien öffnen 36
Quelltextschablonen 88
Suchfunktion 36
Tastenkürzel 89
EFCreateError 615
EFOpenError 73
Eingabefokus
Tastatur 336
Einheiten
Größenverhältnis virtueller ~ 685
virtuelle 678
EInOutError 262
EIntOverflow 261
EInvalidCast 212
Einzelschritte 117
EListError 483
Elternklasse 191
end 66, 179
Endlosschleifen 120, 180
Entwicklungszyklus 27
Entwurfszeit 38
enum (C-Typen) 536
ERangeError 217, 261
EReadError 425
Ereignisprotokoll 108
Ereignisse
Arten 20
auf Betriebssystem-Ebene 23
Auslöser 20
bearbeiten 59
Drag&Drop 746
dynamisch verknüpfen 475
Ereignismethoden
leere 79
für die Grafikausgabe 659
geschützte 734
in Komponenten 906
in Kylix 22
mehrere mit einer Methode verknüpfen 72
Methoden 78
simulieren 736
Standardbearbeitung 22
Standardereignis 77
System 21
Tastatur 733
Tastatureingaben weiterleiten 736
verknüpfen 60
Verknüpfung ändern 78
Verknüpfung löschen 78
zu Kindfenstern weiterleiten 727
errno (Libc-Funktion) 535
Erstellungsreihenfolge 42, 357

Erweiterbarkeit 182
Event-Handler 905
Events 20, 902
  auslösen 905
  deklarieren 903
  Konventionen 904
  mit Rückgabewerten 944
EWriteError 77, 425
except 258
Exception (Klasse) 255
Exception-Handler 258
Exception-Klassen 255
Exceptions 254
  auflösen 260
  bei Datenbank-Updates 876
  EFCreateError 615
  erneuern 260
  EWriteError 77
  im Beispielprogramm 262
  in OnCloseQuery 614
  Konvertierungsfehler 161
  LoadFromFile 73
  OS-Fehlercodes 256
  schachteln 260
  temporär deaktivieren 114
  Umgebungsoptionen für 262
exec... (Libc-Funktionen) 552
  Fehler behandeln 559
execlp (Libc-Funktion) 551
Execute-Methoden selber schreiben 961
execvp (Libc-Funktion) 557
Exit 242
exit (Libc-Funktion) 552
ExitCode
  einer Kylix-Anwendung 559
exports 546
Expose (X Window) 660
Extended 216
external-Deklaration 533
ExtractFilePath 281

F

faAnyFile und andere 282
faDirectory 837
False 218
Farben
  als Zahlenwerte 438
fbsToolWindow 341
Fehlerbehandlung
  in C-Bibliotheken 535
Felder
  abfragen 828
  Datensätze numerieren 857
  Datensätze nummerieren 856
  in Datenbanken 778
  leere 786
  Lookup~ 858
  OnGetText 864
  persistente 822, 858
  persistente Komponenten 865
  statische 822
  Wertangaben im Update-Prozess 878
Felder-Editor 822
Fenster
  anordnen (MDI) 713
  Definition 24
  Instanz 26
  Klasse 26
  Koordinaten umrechnen 323
  teilen 325
  Typen 25
Fensterhandle 302
Fensterhierarchie 24, 310
Fensterlisten
  an Menüs anhängen 713
Fenstermanager 323
  Formulargrößen-Änderung 342
  Geschwindigkeit 347
  Rahmen-Icons 343
FieldByName 818
FieldDefs 783
FileDB 832, 848
FileExists 139
FileSearch 282
FillChar 289
finalization 181, 243
Finalize 223
finally 257
FindClass 426, 911
  für beliebige Klassen 429
FindFirst 281, 282, 837
  Beispiel 456
FindNext 837
FixedSizeFrm (Beispielformular) 342
Flackern
  von dynamischen Werkzeugleisten 718
Fließkommatypen 215
Fließkommazahlen 170
fmOpenRead und andere 425
Fokus
  in einer Listbox 136
for 240
fork (Libc-Funktion) 551, 554, 557
Format 879
  Beispiel 71
Formatstrings 286
FormClass 208
Formular
  als Schablone 44
  Deklaration 80

Entwurfszeitverhalten 39
Fehlermeldungen beim Laden 897
Formulardateien 910
übergroße 799
Formular-Designer 39
Formulare 24
Abhängigkeit 718
aktives ermitteln 317
als Komponenten 960
als Textdatei editieren 350, 357, 363, 364, 606
andocken 738
Arbeit mit mehreren ~ 346
Arten 339
automatische Anzeige 94, 314
automatische Erstellung 92
automatische Größe 335
Beschränkung der Größenveränderung 334
Dateien 89
Definition 26
dynamisch erzeugen 346
Entwurf 38, 40
flexibel entwerfen 74
für Datenbanken 797
Funktion als Schablone 26
gleichmäßig aufteilen 341
globale Variablen 709
Größe fest einstellen 342
Hauptformular 94
Hintergrund-Panels nachträglich einfügen 75
im Textformat speichern 364
in Objektablage ablegen 98
Liste 32
Liste zur Laufzeit 318
Menüs 131
Methoden selbst erzeugen 71
mit kleiner Titelleiste 341
modale 339, 347
modale Dialoge 124
ohne Rahmen 341
Rahmenstile 340
Schließen 346
Sicherheitsabfrage 613
Unabhängigkeit 730
und OOP 182
verbinden 800
visuell vererben 97
Wiederverwendung 95
zentrieren 344
Formularklasse
für Kylix reservierte Bereiche 81
Formularvererbung 397
nachträglich einführen 402
Vergleich mit Frames 405
Formularverknüpfung 801
visuelle 376
FrameGlobalToolbar (Beispiel-Unit) 729
Frames 24, 403
Anwendungsbeispiel 576
für Werkzeugleisten 728
im TreeDesigner 406
TGlobalToolbar (Beispiel) 729
Vorteile 405
Free (Methode) 195, 197
friend (C++) 190
fsMDIChild 702, 704
fsMDIForm 702
fsMdiForm 705
fsNormal und andere 343
ftString und andere 820
Funktionen 244
Ergebnisse 244
Standardparameter 251
überladen 252

G

Gerätekoordinaten 690
Geräteunabhängigkeit 678, 696
GetEnvironmentVariable 138, 563
GetMem 235
Gimp 57
GlobalProcs (Beispielunit) 498
goto 243
GPL 577
Grafik
geräteunabhängige 678
Grafikausgabe 441
Clipping 436
effiziente 666
ereignisgesteuerte 659
in Weltkoordinaten 571, 574
Koordinatentransformation 569
Grafiken
Entwurf 57
Formate 57
Grafikkomponenten
Namenstabelle 454
Grafikobjekte 449
kopieren 454
Graphen 645
GrDoc (Beispielprogramm-Unit) 626
grep 560
Gruppenfenster 349
Gruppierung
von Steuerelementen 349
Gültigkeitsbereich
Modul 179
Gültigkeitsbereiche 178
GUID
erzeugen 271
für Interfaces 271

H

Halt 243, 552
Haltepunkte
  Adress- und Datenhaltepunkte 113
  siehe Breakpoints 111
Haltepunkteliste 112
Haupt-/Detail-Formulare 851
Haupt-/Detail-Verbindung
  lösen 852
Hauptformular 94, 346
Hauptmodul 179
Hauptprogramm 179
Heimatverzeichnis 34
  ermitteln 563
Hexadezimalzahlen 170
High 218, 249, 289, 356
Hilfecompiler 888
Hilfedateien 888
Hilfefunktion 35
Hinweise 318, 332
  in Statuszeile 155
  kurze und lange 156
Hinweisfenster 332
  Hint-Properties in TApplication 313
  mit der CLX 156
History-Listen 922
HKeyMan 478
HList 470
Home-Verzeichnis 563
HResult (Typ) 278

I

I/O-Prüfung 262
Icons 450
IconViews
  siehe TIconView 375
IDE 30
  Anwendungen außerhalb starten 530
  Aufbau 30
  Ausgabeverzeichnis 90
  Ausschneiden und einfügen 75
  automatische Quelltextänderung 80, 81
  Browser 100
  Compiler aufrufen 62, 93
  Dateiverwaltung 43
  Desktop-Toolbar 34
  Docking 32
  Docking ausschalten 33
  Editor 83
  Ereignisse verknüpfen 60
  Fenstermanagement 32
  Formulare editieren 39, 40
  Formularschablonen 98
  Frames entwerfen 404
  Gruppen bilden 349
  in Datenmengen navigieren 824
  Klassenvervollständigung 84
  Komponenten ausrichten 41
  Komponenten auswählen 40, 46
  Komponenten gleichzeitig verändern 50
  Komponenten installieren 894
  Komponenten löschen 41
  Komponenten umordnen 41
  Komponentenpalette 36
  Komponentenschablonen 99
  Konfiguration speichern 33
  Kopieren von Komponenten 725
  Listviews entwerfen 158
  lokale Menüs 31
  Mausfunktionen 40
  Menüs editieren 53
  neu starten 31
  Neue Formulare und Projekte erzeugen 98
  Objektablage 95
  Objektinspektor 44
  Programmausführung 111, 112, 117
  Programme starten 61
  Projektverwaltung 89
  Property-Editoren 50
  Quelltextdateien 32
  Schnittstelle für Programmierer 914
  Speichern unbenannter Projekte 44
  Tabellen erzeugen 781
  To-Do-Listen 102
  Umbenennen von Komponenten 81
  Umbenennen von Methoden 81
  Verwaltung von Ereignismethoden 78, 79
  Visuelle Formularvererbung 97
IDesignerHook 916
  IsDesignEvent 917
  Modified 917, 957
IDomDocument 760
  createElement 773
IDomElement 761
IDomNamedNodeMap 770
IDomNode 758, 763
if-Anweisung 241
IID (Interface Identifier) 271, 278
IInteface
  _Release 272
IInterface 270
  _AddRef 271, 278
  _Release 278
  QueryInterface 271, 278
IInterface-Übersicht 271
image (Mime-Typ) 492
implementation 66, 71, 180
implements 275
in 219

Inc 231, 261, 289
Include-Dateien 173
IncludeTrailingPathDelimiter 139, 289, 836
index -Direktive 189
IndexDefs 783
Indizes
  eindeutige 856
  in Datenbanken 843
InfoForm (TreeDesigner-Formular) 601
inherited 194, 202
  Unterschied zu Borland Pascal 194
INI-Dateien 417, 478
  automatisches Management 925
  Sektionen 419
initialization 181
Installation der CD 885
Instanzen 183
int (C-Typ) 536
Int64 214
  Überlauf 215
Integer 214
Integrität
  referentielle 790
Interbase
  Beispieldatenbanken 784
  Dateien erzeugen 785
  Datenbank-Browser 811
  Generatoren 856
  siehe auch isql 854
  siehe auch SQL 856
  Sprachreferenz 785
  Trigger 856
  Versionen 783
interface 66, 180, 264
Interfaces 264
  Deklaration 266
  delegierte Implementation 275
  im TreeDesigner 718
  Methoden zuordnen 279
  Vererbung 270
  Verwendung 267
Interface-Variablen 270, 274
InterlockedIncrement 278
Interlocked-Routinen 514
Interpunktionszeichen 169
IntersectRect 670
IntToHex 288
IntToStr 288
Invalidate 660
InvalidateRect 667
Invertieren
  eines Rechtecks 447
IoResult 262, 285
is-Operator 211, 212
isql 784, 785
  Metadaten extrahieren 854
  Skripte ausführen 786
IToolbarHost (Beispiel) 718, 722
IUnknown 270
  siehe IInterface 271

J

Jahreszahlen 286
Join 861
  innerer 861

K

Kanten (in Bäumen) 645
KApplication 583, 588
Kategorien
  für Properties definieren 901
  Namenskonstanten 902
KDE
  Anwendungsobjekte 583
  Icons für Verzeichnisbäume 967
  Konqueror 157, 577
  Widgets 577
Key_Delete 734
Key_Next 734
khtml 577
KHTMLPart 579
KHTMLView 581
Kindfenster 25
Klassen 183
  ableiten 191
  abstrakte 204, 265
  Basisklassen 191
  Deklaration 183
  Deklarationen per Skript finden 559
  Gültigkeit des Zugriffsschutzes 190
  Registrieren 429
  statische Elemente in (C++) 208
  Vorwärtsdeklaration 191
  Zugriffsschutz 190
Klassenhierarchie 191
Klassenmethoden 208
  und Klassenvariablen 208
  und Objektinstanzen 210
Klassenreferenzen 205
  in TFieldDef 827
  Kompatibilitätsregeln 207
  konstante 207
  Verwendungsbeispiel 206
Klassenreferenz-Typen 206
Klassenvervollständigung
  Beispiel 72
Knoten (in Bäumen) 645
Knoten von XML
  siehe DOM 759

kof-Dateien 418
Kollision
  bei gleichzeitigen Datenzugriffen 514
Kommentare 63, 170
Komponenten 24, 26
  ~Benutzer 886
  Adapter 971
  aktive ermitteln 317
  aktivieren 320
  als Gruppenfenster 349
  anordnen 324
  anzeigen 321
  Anzeigesteuerung 333
  Arrays 355
  besitzergezeichnete 446, 455, 458
  Besitzhierarchie 308
  Bitmaps 914
  datenbanksensitive 801
  DefineProperties 954
  Definition 26
  dynamisch erzeugen 929
  Editieren 40
  Editoren 915, 947, 950
  Eingaben abfragen 145
  Entwickler 886
  Erstellungsreihenfolge 42
  Erweiterung 898
  EventFilter 908
  Fensterhierarchie 310
  Freigabe 308
  für KDE-Widgets 581
  für Menüs 53
  gleichzeitig behandeln 351
  Größenänderung behandeln 930
  gruppieren 42, 349
  hinzufügen 39
  Hüllkomponenten für Formulare 960
  in Komponenten 942
  installieren 885, 894
  kombinieren 927
  Konzept von Kylix 886
  Koordinaten 322
  kopieren 725
  Laden aus einem Stream 911
  Liste 38
  Liste des Formulars 310
  Liste im Objektinspektor 46
  neu übersetzen 897
  neu zeichnen 334
  nicht visuelle 301
  Properties 900
  Property-Kategorien 901
  Referenzen im Objektinspektor 51
  Reihenfolge 42
  Schablonen 99, 126, 403
  Schein-Properties 954
  Speichern 910
  speichern 953
  Tabulatorreihenfolge 42, 49
  TComponent 300
  testen 931
  Verändern 917
  verstecken 321
  virtuelle Methoden 907
  visuelle 302
  Wesen von 886
  zeichnen 939
  Z-Reihenfolge 321
  zur Entwurfszeit 916
  zwischen Formularen teilen 800
Komponentenbibliothek 889, 932
Komponenten-Experte 891
Komponentenpalette
  Seiten 36
Komponentenschablonen 725
Konfigurationsdateien 417
Konqueror
  siehe KDE 157
Konsolenanwendungen 548
  Ausgabeumleitung 553
Konstanten 64, 169, 176
  Arrays 239
  initialisierte 239
  lokale 178
  Records 239
  typisierte 177
  zuweisbare typisierte 177
Konstruktor
  für Arrays 248
Konstruktoraufruf
  intern 195
  ohne Objekterzeugung 193
Konstruktoren 192, 638
  aufrufen 193
  virtuelle 207
Kontextmenüs 31
Kontrollstrukturen 63
Koordinatensysteme
  Achsenrichtungen 686
  Ursprung 684
  virtuelle 678, 686
  Zoomen 687
Koordinatentransformation 569, 678
Koordinatenumrechnung 323, 689
Kundendatenbank 779
Kylix
  Online-Referenz 35

label 243
Laufzeit 39
Laufzeitbibliothek 281
Laufzeitfehler
  Fehlende Komponentenklasse 82
Laufzeitfehler 210 204
Laufzeit-Typinformationen 204, 209, 212
lbOwnerDrawVariable 455, 458
ld.so 530
LD_LIBRARY_PATH 529
ldconfig 530
ldd 530
Length 230, 288
Lesemethode
  eines Properties 187
Lexikalische Elemente
  von Object Pascal 169
LGPL 577
libc 528
Libc (Unit) 532
libqtxml_wrapper (Beispiel) 762
library 546
Linken
  bei Programmstart 530
  Bibliotheksverzeichnisse 529
Linux
  C-Bibliothek 528
  Kernel 528
Linux (Compilersymbol) 174
Listenkomponenten
  Bedienung programmieren 134
ListViews
  entwerfen 158
  siehe TListView 375
Lizenzbedingungen
  der (L)GPL 577
LoadCursor (Beispielfunktion) 330
LoadThemeHook 583
loCaseSensitive und loPartialKey 841
LongBool 218
LongInt 214
LongWord 214
Lookup-Felder 824
  in Listen und Komboboxen 826
Low 218, 249, 289, 356
LowerCase 288
LPtoDP 690

  GUI-Front-End 555

Markierungsschalter
  grauer Zustand 354
Markup 754, 756
  in Formulardateien 755
Master-/Detail-Formulare
  siehe Haupt-/Detail-Formulare 851
Mauseingaben 328
Mauspaletten (siehe Werkzeugleisten) 603
Mausprogrammierung 650, 655
Mauszeigerformen 329
  definieren 329
  eigene 330
Mauszeigerressource 59
MDI-Anwendungen 25, 702
  AutoMerge 708
  Dokumentfenster 702
  Dokumentfenster in SDI-Fenster umwandeln 707
  Fensterliste 713
  Fenstermenü 713
  GroupIndex (TMenuItem) 708
  Hauptfenster 702
  Hauptmenüpunkte wiederholen 715
  Icons für Kindfenster 703
  Kindfenster ansprechen 709
  Kindfenster automatisch öffnen 711
  Kindfenster erzeugen 710
  Kindfenster schließen 621, 712
  Kindfensteraktivierung 720
  lokale Werkzeugleisten 716
  Mauspaletten 703
  Menüs verschmelzen 707
  Startparameter 712
  Unabhängigkeit der Fenster 703
Mehrfachvererbung 280
Meldungen
  im Debugger 108
Meldungsfenster
  mit Abschalt-Option 969
Memofelder 372
Memos
  Dateien laden und speichern 68
Mengen 233
  Konstanten 170
  Properties 354
  von Flags 51
Menü
  Komponenten 53
Menü-Designer 53
  Bedienung 54
  Untermenüs 54
Menüpunkte
  Properties in Aktionslisten 483
Menüs 131, 464
  Aufklapp-Ereignis 493
  Automatisches einbinden von Menüpunkten 963

Befehle ausführen 56
dynamisch aktivieren 493
dynamisch erweitern 470, 714
markieren 615
Methoden verknüpfen 56
Popup-Menüs 616
Properties der Menüpunkte 54
Punkte markieren 132
Tastenkürzel 469
verschmelzen 707
Menus (Unit) 465
MessageDemo 110, 308
MessageDlg 77
Metadateien 462
Metadaten 785
Metafile (Beispielprogramm) 462
Metaklassen 205
Metasprache 756
Methoden
aufrufen 63
dynamische 203, 307
statische 199
überschreiben 201
virtuelle 197, 200
virtuelle Ereignis~ 907
VMTs 202
zur Ereignisbearbeitung 60
Methodenkopf 60
Methodenzeiger 198, 254, 902, 903
und virtuelle Methoden 198, 202
zur Laufzeit verknüpfen 715
midas.so 781
Mime-Typen der CLX 492
mmAnIsoTropic (und andere) 683
mod 219
Modalität 340
Modulkonzept 179
Modultypen 179
MouseCapture 651
mrOK und andere 127, 340
Multitasking
gleichzeitige Datenzugriffe 511
kooperatives 496
preemptives 502
Mutexe
in TEvent 517
MyBase 780
Tabellen erzeugen 781
MySQL 780

N

Nachrichten
abfangen 307
mit Qt veschicken 736
Übersicht 303
nbFirst und andere 806, 808, 809
Neuzeichnen
von Komponenten 334
New 235
NewItem 468, 961
NewLine 468
NewMenu 468
NewPopupMenu 468
NewSubMenu 468
nil 195, 236
nodefault 912
Normalformen
von Datenbanken 779
NormalizeRect (TreeDesigner) 668
not 219, 220
NOT NULL 786

O

object 183, 213
object (Schlüsselwort) 499
Object Pascal 165
lexikalische Elemente 169
Schlüsselwörter 171
und C++ 166
Objektablage 95
Inhalt 98
Registerdialog 359
Objekte
Aufbau 182
DMTs 203
Initialisierung 192
Kompatibilität 198
Lebenslauf 192
polymorphe 203
polymorphe Speicherung 427
self-Parameter 185
VMTs 202
Wesen von 181
Zugriff auf die Klassenelemente 184
Objektinspektor 44
bearbeitetes Objekt 46
Ereignisse 22
Ereignisse verknüpfen 60
Komponentenliste 46
Property-Editoren 50
Seiten 45
Objekttyp
siehe Klasse 183
odSelected und andere 458
of
bei Arrays 222
bei Klassenreferenzen 206
ofPathMustExist und andere 370
OnClose 712

OnCloseQuery 712
bei Dokument-View-Anwendungen 625
OnDestroyDocWin (selbstdef. Event) 622
Online-Referenz
Lücken 35
OnNewRecord 864
on-Schlüsselwort 259
OOP 181
open (Libc-Funktion) 554
Operatoren 169, 219
bitweise 220
boolesche 220
für Mengen 234
Rangfolge 219
relationale 221
Optimierungsoptionen 105
219, 220
Ord 289
ord 218
Order-By-Klausel 844
OutputDebugString(Windows) 110
overload 252
override 200, 201
OverviewForm (TreeDesigner-Formular) 601
OwnerDraw1 (Beispielprogramm) 455
OwnerDraw2 (Beispielprogramm) 461
OwnerDraw-Elemente 455

Packages 896
Editor 894, 897, 932
Entwurfszeit~ 890, 947, 952, 953
installieren 894, 897
Laufzeit 890
Quelltexte 896
packed 538
Paradox 788
Primärindex 849
ParamCount 289
Parameter 246
aktuelle 246
formale 246
Standardparameter 251
untypisierte 247
variable Listen 249
Variablen-Parameter 246
von Methoden 60
Wertparameter 246
ParamStr 289
Pascal
Ur-Pascal 175
Pascal (Aufrufkonvention) 245
Passwort
masterkey 793
PChar 229, 231
Array von PChars 557
Erzeugen aus einem String 226
PenCapStyle (Qt) 445
pfInUpdate/pfInWhere 881
Pinsel 439
Pipes 553
pmCopy und andere 438
pmNotXor 653
poDesigned und andere 344
Pointer 234, 236, 539
poLandscape 699
Polymorphie 197, 203, 275
in Streams 426
PolyStrm 428
poPortrait 699
Popup-Menüs 616, 744
Portieren
Autoinkrement-Felder 791
BDE-Anwendungen 790, 791, 815
Bilderlisten für Common Controls 375
Docking-Prinzip 739
InvertRect 448
LoadCursor 332
mit VCanvas 680
MM_LOMETRIC 700
nicht-transparente Textausgabe 459
Property-Kategorien 902
SetTextAlign (WinAPI) 664
SetViewportOrg/Ext 680, 682
SetWindowOrg/Ext 680
Standard-Dialoge 367, 371
TApplicationEvents 970
TFileListBox 968
THotkey durch TComboBox 481
Threads 508
TIniFile-Verwendung 418
TListView mit vier Ansichtsarten 383
TMetafile 462
TNotebook 366
TPrinterSetupDialog 697, 971
TTabbedNotebook 363
TUpDown 971
Unterschiede bei OnIdle 500, 502
WindowMenu-Property 713
Windows-Clipping-Funktionen 677
WM_MdiActivate 720
Portierung
OutputDebugString 110
Pos 288
PostScript
Drucken 693
Einheiten 693
Printer 692, 699

private 190
private-Bereich
  einer Formularklasse 71
program 179
Programmabschnitte
  kritische 514
Programme
  starten 552, 787
Programmierung
  ereignisorientierte 886
  objektorientierte 886
Programmlader 530
Projekt
  aktives 94
  anlegen 30
  erzeugen 62, 93
  Quelltext 92
Projektdatei 179, 711
Projektdateien 44, 90
Projekte
  Dateien 89
  und Packages 897
Projektgruppen 94
  Beispiel 91
  Quelltext 95
Projektmanager 91
Projektoptionen
  Debug-DCUs 119
  Suchpfad 119
Projektverwaltung 89
  Fenster 91
  nicht-visuelle Units 90
  und Projektdatei 91
Properties 44, 637
  Array-Properties 188
  Bezeichnung 45
  binäre Speicherung 957
  deklarieren 187
  Editoren 50, 915, 947
  für richtiges Timing 682
  geänderte feststellen 606
  geschachtelte 51
  in Komponenten 900
  Indizierte 189
  Kategorien in der IDE 45
  Konzept 186
  Mengen 354
  Reihenfolge der Initialisierung 963
  selbsteingeteilte Kategorien 901
  setzen 64
  speichern 912
  Standard-Properties 188
  Typen 50
  überschreiben 900, 913
  veröffentlichen 917
  verwenden 186
  zur Implementation von Interfaces 275
protected 190
Provider 796
Prozeduren 244
Prozesse 550
  mit Ausgabeumleitung 554
  starten 551
  Umgebungsvariablen 563
PtInRect 657
public 190
published 190
PWideChar 231
PWideString 568

Q

QApplication 583
  postEvent 737
  sendEvent 736
QClipbrd (Unit) 490
QCursor 331
QDomDocument 767
  load 768
  toString 774
QEvent 305
  type 909
QImageIO 58
QKeyEvent 737
QMouseEvent 909
QObjectH
  Deklaration 567
QPaintDevice 700
QPaintDeviceMetrics 700
QPainter 445, 570
  Clipping-Methoden 676
  drawText 665
  drawtext 574
  Methoden 571
  setWindow/Viewport 680
  Zugriff über die CLX 572
QPaintEvent 660
QPen
  setCapStyle 446
QPicture 454, 462
QPixmap
  createHeuristicMask 331
QPrinters 697
QPrinters (CLX-Unit) 691
QRegion
  verwenden 676
QScrollView 581
QString
  und WideString 568
Qt
  Widget-Handle 302

QTextBrowser 337, 577
qtintf.so 564
Qt-Toolkit
  als CLX-Grundlage 292
  Anbindung an Object Pascal 564
  Aufzählungstypen 569
  besondere Zeichenstile 445
  direkt aufrufen 337
  Dokumentation 566
  DOM-Modul 762
  Ereigniskonzept 23
  in der CLX verwendete Klassen 294
  Koordinatentransformation 569
  Metadateien 462
  Nachrichtenverarbeitung 305
  Signals in der CLX 305
  Standarddialoge 366
  Stile für GUI-Elemente 314
  und WideStrings 232
Qt-Unit 138
Qt-Widgets
  und TWidgetControl 302
QWidget
  winid 546
QWidgetH 302
QWMatrix 566, 570
QXMLBridge
  Implementation 763
QXMLBridge (Beispiel-Unit) 762

raise 255
Random 289
Randomize 290
read 283
read-Direktive 187
readln 283
  von Konsole 549
Real
  Kompatibilität 216
Rechtecke
  vertauschte Koordinaten 668
Reconcile-Dialog 875
record 232
Records 147, 183, 232
  in Datenbanken 778
  packed 538
  Varianten 233
  von C übersetzen 538, 541
Rectangle
  invertieren 448
reentrant 511
Referenzzählung bei IInterface 272
Register 959
  für Property-Editoren 952
Register (Aufrufkonvention) 245
RegisterActions 959
RegisterClass 426
  für beliebige Klassen 429
RegisterComponentEditor 952
RegisterComponents 914
RegisterPropertiesInCategory 901
RegisterPropertyEditor 952
Registrierung
  von Editoren 951
Registry
  automatisches Management 925
  Speichern von History-Listen 419
reguläre Ausdrücke 561
Reintroduce 201
Relationen 778
repeat 241
requires-Klausel 896
Reset 284
Ressourcen
  einbinden 59, 173
  schützen 258
  verwenden 330
Ressourcendateien 58
Ressourcenverzeichnis
  von Kylix 34
Rewrite 284, 691
RGB-Funktion 438
Root-Fenster 25
Round 215, 290
Route
  große 19
  kleine 19
  kleine, Startpunkt 27
  kleine, Stationen 28
RTLVersion 174
RTTI 209, 212, 426
RunProgram
  Fehlerbehandlung 787
RunProgram (Beispielfunktion) 554, 786

S

SafeCall 245
SbTest (Beispielprogramm) 735
ScaleDialog (TreeDesigner-Formular) 602
Schalter
  Arrays 355
  flache 606
  Gruppen 607
Schleifen
  endlose 120, 241
  for 240
  repeat 241
  while 242
Schlüsselwörter 169, 171

Schnittstellen (siehe Interfaces) 264
Schreibmethode
  eines Properties 187
Schriftarten 439
  in einer Kombobox 919
Schutzverletzung 192
Scrolling 671, 672
  fließendes 675
SDelphiBitmap 492
sed (Beispiel) 561
Seek 284
self 185, 208
Semikolons
  in Object Pascal 172
Sender 353
SetLength 228
  bei Arrays 223
SetViewportExt 685
SetViewportOrg 684
SetWindowExt 685
SetWindowOrg 684
Shared Libraries
  erzeugen 546
Shell
  Skripte 559
Shift 656
shl 219, 220
  Beispiel 438
ShortCut 470
ShortCutToKey 470
ShortCutToText 470, 481
ShortInt 214
ShortString 225, 226
ShowMessage 68
  verbessern 969
shr 219, 220
Sicherheitsabfrage 613
  vor Schließen eines Fensters 76
Signals 23
Single 216
SizeOf 249, 290
Skripte
  aus Kylix-Anwendungen starten 559
Slots 23
SmallInt 214
so-Bibliotheken
  zur Laufzeit laden 586
so-Dateien
  in C für Kylix erzeugen 580
  mit Kylix erzeugen 546
Software-Entwurf 28
Speicheradressen
  ausrichten 538
Speicherreservierung 193
SQL
  Anweisungen ausführen 878
  Beispielabfragen 859
  CREATE DATABASE 785
  CREATE GENERATOR 857
  CREATE INDEX 785
  CREATE TABLE 785
  CREATE TRIGGER 857
  DDL 785
  eindeutige Indizes 856
  Filtern von Datensätzen 862
  Joins 861
  Monitor 881
  Order By 844, 862
  Parameter selber einsetzen 879
  parametrisierte Anfragen 862
  SELECT 860
  UPDATE 878
SQLClientDataSet 796
ssAutoBoth 48
ssLeft und andere 328
ssShift und andere 656
Stack-Prüfung 262
Standardaktionen
  selbst definieren 957
  Verwendungsbeispiel 489
Standarddialoge 366
Standarddirektiven
  von Object Pascal 171
Standardereignis 77
Standardialoge
  Aufruf 367
Standardkomponenten 36
Standard-Properties 188
startkylix 31
Startparameter 712
statfs (Libc-Funktion) 541, 542
Statuszeile
  Hinweise 155
Stdcall 245
Step 117
Steuerelemente 26
Steuerzeichen 170
Stifte 437
  438
  Stil der Linienenden 445
Stile
  Application.Style 314
Stored 912
stored 917
StrCat 288
Streams 424, 640
  polymorphe 426
String
  automatische Initialisierung 228

Stringlisten 410
Strings
  Aufbau der langen Strings 228
  Copy On Write 229
  Konstanten 170
  nullterminierte 230
  offene 247
  Pascal-Format 226
  Referenzzähler 229
  umwandeln in DelimitedText 411
  Zugriff auf einzelne Zeichen 228
 rLen 230, 288
 rLower 288
 rPCopy 230, 288
 rToDate 286
StrToInt 288
 rToTime 124
Strukturierung 182
 rUpper 288
 yle-Sheets 757
SuspendableThread (Beispielunit) 518
Symbolleisten (siehe Werkzeugleisten) 603
Synchronisation
  mit TCriticalSection 514
Synchronize-Methode 505
sysinfo (Libc-Funktion) 543
syslog 110
system (Libc-Funktion) 786
Systeminformationen
  Drucker 699

Tabellen
  erzeugen 781
  in Datenbanken 778
  Strukturinfo mit FieldDefs 826
  verknüpfen 859
Tabulatorreihenfolge 42, 49, 350, 357
TAction 484, 958
  Enabled 616
  ExecuteTarget 959
  HandlesTarget 958
  OnHint 486
  OnUpdate 486
TActionLink 484
TActionList 484, 713
  Verwendungsbeispiel 489
Tags
  in XML 756
tail 110
TAlarm (Beispielklasse) 147
TApplication 312
  Active 313
  Cursors 330
  Ereignis-Übersicht 316
  ExeName 313
  HandleException 614
  Hint 155
  MainForm 314
  Minimize 315
  NormalizeTopMosts 315
  On... 316
  OnActionExecute 487
  OnEvent 308
  OnHint 155
  OnIdle 496
  ProcessMessages 500, 501, 837
  Property-Tabelle 313
  Restore 315
  RestoreTopMosts 316
  ShowException 316
  Style 314
  Terminate 316
TApplicationEvents
  Ersatzkomponente 970
Tastaturbedienung 48
Tastatureingaben 327, 733
  im Formular abfangen 738
Tastaturfokus 302, 326, 336, 733, 735
  weiterleiten 736
Tastencodes 138
Tastenkürzel 469
Tastenkürzeleditor 478
  Bedienung 479
TBatchMove 788
TBevel 162
TBitBtn 129
TBitmap 452
  Create 452
  Dateien laden 456
  erzeugen 452
  für selbstgezeichnete Bilderlisten 390
  zeichnen 457
TBlobField 820
TBookmark 843
TBooleanField 820
TBorderIcons 353
TBorderStyle 352
TBrush 439, 639
  OnChange 637
tbsButton und andere 604
TButton 129
  Cancel 125
  Default 125
  ModalResult 125, 340
TCanvas 434
  Arc 442
  beim Drucken 690
  Canvas-Objekte 435
  Chord 442

Draw 457
Handle 572
Koordinatenangaben 442
LineTo 441, 443
MoveTo 441
OnChange 637
Pie 442
PolyBezier 443
Polygon 443, 663
PolyLine 441, 443
Start/Stop 463, 572, 661
StretchDraw 457, 461, 464
TextHeight 444
TextOut 444, 664
TextRect 444, 666
TextWidth 444
Übersicht 445
UpdateRect 671
TCanvas.StretchDraw 390
TCheckBox 143
TCheckListBox 133
TClass 205
TClassList 417
TClientDataSet 795
Active 799
Änderungsprotokoll 815, 871
Änderungsprotokoll anzeigen 872
ApplyRange 846
ApplyUpdates 816, 836, 857, 870
BeforeUpdateRecord 877, 878
CancelRange 846
CancelUpdates 808, 816
ChangeCount 873
Close 811
CreateDataSet 783
Data 817, 873
Datenpaket 780
Delta 815, 873
FileName 782, 783
Goto-Methoden 847
IndexDefs 845
IndexFieldCount 845
IndexFieldNames 844, 852
IndexName 844
Indizes 849
KeyExclusive 846
komplette Tabellen laden 841
Locate 850
MasterFields 852
MasterSource 851
MergeChangeLog 815, 817
MyBase-Tabellen erzeugen 781
OnGetTableName 881
OnNewRecord 864
OnReconcileError 875
PacketRecords 852
Params 862
ReadOnly 880
Refresh 871
SetKey 847
SetRange 846
siehe auch TDataSet 806
Sortierreihenfolge 844
StatusFilter 873
UndoLastChange 808
Verbindungswechsel 811
Wirkungsbereich von Datenbank-Operationen 840
TClipboard 488, 493
Assign 491
AsText 491
Supports 492, 493
TColor 438
TColorDialog
CustomColors 943
TColorDialogs 370
TColorPalette 931
Color-Properties 937
Columns 936
Deklaration 933
FColors 936
Lines 936
NoSelection 938
OnDefineColor 945
OnExpandPopup 944
Property-Kategorien 901
RGBSteps 936
SelColors 937
Standardaktionen 957
SwapAlign 936
TCombBox
DroppedDown 813
TComboBox 132, 142
als History-Liste 922
besitzergezeichnet 919
OnChange 69
OnSelect 813
TComponent 26, 300, 887, 910
Besitzhierarchie 308
ComponentCount 301, 309
ComponentIndex 301, 309, 310
Components 301, 309, 911
ComponentState 301, 911, 916, 964
Create 309
FindComponent 301
InsertComponent 309
InsertComponnet 301
Loaded 911, 919, 964
Name 300
Owner 301, 309

RemoveComponent 301, 309
Tag 300
TComponentEditor 915
Methoden 950
TComponentList 417
TControl 302, 311, 320
Align 324
Anchors 335
AutoSize 335
Bounds 322
BringToFront 321
Caption 324
Click 907
ClientHeight 322
ClientRect 322
ClientToScreen 324, 745
ClientWidth 322
Color 333
Constraints 334
Controls 357
Cursor 329
DblClick 907
Drag&Drop 746
Drag&Drop-Ereignisse 746
DragDrop 907
DragOver 907
Font 333
Hide 321
Invalidate 334, 687
Koordinaten 322
Left/Top/Width/Height 322
MouseDown 907, 941
Parent 711, 717
ParentColor 333
ParentFont 333
Repaint 334
Resize-Methode 930
ScreenToClient 324
SendToBack 322
SetBounds 322
Show 321
Showing 321
Text 324
Update 334
WndProc 950
TControlBar 130, 607, 706
TControlScrollBar 338
Range 687
TCriticalSection 514
Beispiel 515
TCurrencyField 820
TCustomClientDataSet
siehe TClientDataSet 862
TCustomConnection 779, 792
TCustomControl 294, 303, 928
Ereignismethoden 907
Paint 939
TCustomEdit 372
TCustomForm 339
TCustomFrame 404
TDataSet 779
Active 799
AfterPost 869
BOF 806
Cancel 808
Delete 809
DisableControls 832, 839
Edit 807, 850
EnableControls 839
EOF 806, 843
FieldByName 818, 831
FieldCount 818
FieldDefs 783
Fields 818, 826
FieldValues 831
FindField 831
First 806
FreeBookmark 843
GetBookMark 843
GoToBookmark 843
Insert 809, 850
Last 806
Locate 841, 850
Lookup 842
Next 806
OnFilterRecord 842
Open 864
Post 808, 814, 848, 866
Prior 806
ReadOnly 807
Refresh 808
State 848
Zustände 847
TDataSetProvider 796
TDataSource 795, 798, 799
AutoEdit 807
TDateField 820
TDateTime 122, 285, 286
TDateTimeField 820
TDBCheckBox 804
TDBComboBox 804
TDBEdit 803
TDBGrid 799, 802, 806
TDBImage 802, 827
TDBLabel 803
TDBListBox 804
TDBLookupComboBox 868
TDBLookupCombobox 826

TDBLookupListBox 804, 826
TDBLoopkupComboBox 804
TDBMemo 802, 804, 827
TDBNavigator 804, 807, 808
  im Felder-Editor 824
  VisibleButtons 880
TDefaultStyle 315
TDialog 366
TDirectoryListBox
  Verwendungsbeispiel 455
TDirectoryTreeView 966
  Beispiel 810
  Listing-Auszüge 392, 394
TDockableForm (Beispiel) 739
  Implementation 743
TDocumentForm 603
  BrushColorChange 728
  Changed, siehe TGraphicDoc 629
  CmChangeRulerFontClick 617
  CmGlobalOpenClick 715
  CmLoadClick 612
  CmPrinterSetupClick 697
  CmSaveAsClick 611
  CmSaveClick 611
  Create 622, 719, 721, 727, 729
  DeleteCommand 734
  DragAction 652
  DragMode 652
  DragStartX/Y 651
  FormActivate 720
  FormClose 713
  FormCloseQuery 614
  FormCreate 622, 715, 731
  FZoomFactor 687
  LastX/Y 651
  LoadFromFile 612, 712
  LoadFromStream 640
  MapMousePos 690
  MdiActivate 721
  MdiDeactivate 722
  MouseObject 652
  NoMapModeClick 615
  OnMouseDown 658
  PaintBoxDragDrop 753
  PaintBoxDragOver 752
  PaintBoxMouseDown 652, 658, 735, 745
  PaintBoxMouseMove 654
  PaintBoxMouseUp 655
  PaintBoxPaint 661, 670, 682
  PrintClick 698
  ResizePaintbox 687
  ScrollboxKeyDown 734
  ScrollBoxKeyPress 737
  ScrollBoxScroll 676
  SDI und MDI 726
  SetAttributes 726, 730
  SetDocument 623
  SetMapMode 682, 689
  SetZoomFactor 687
  ShiftOrderCommand 734
  ToolbarHost 719
  UpdateCaption 624
  ZoomFactor 687
  ZoomFactorBarChange 688
TDrawGrid 447
TDrawing 462
TDXML (Beispiel-Unit) 766
TEdge 645
  StandardConnection 647, 648
TEdit 144
  AutoSelect 737
  in neuer Komponente 927
Teilbereichstypen 217
TEmptyPanel (Beispiel) 918
TEvent 517
  Beispiel 518
Text
  ausrichten 48
  rotieren 573
  transparenter 444
  zentrieren 574
text (Mime-Typ) 492
Textausgabe 444
Textausrichtung 664
Textdateien 283
  für die Druckerausgabe 691
TextToShortCut 470
TField 817
  allgemeine Properties 818
  As... 830
  DataType 819
  FieldName 822
  Lookup 825
  NewValue 879
  OldValue 879
  OnGetText 864
  OnSetText 864
  persistente/statische Felder 822
  Properties zur Anzeigesteuerung 821
  ProviderFlags 881
  ReadOnly 880
  Value 830
TFieldDef
  Required 786
TFieldDefs 783, 826
TField-Hierarchie 819
TFields 818
TFileListBox
  Ersatzkomponente 968
TFileListBox (Ersatzkomponente) 966

TFiler 430
Arbeitsweise 432
DefineBinaryProperty 957
DefineProperty 955
Polymorphie 955
TFileStream 424
Beispiel 150
TFileTable 966, 967
Beispiel 810
TFindDialog 371
TFloatField 820
TFont 371, 439, 639
Pitch 440
Übersicht 439
TFontComboBox (Beispielkomponente) 919
TFontDialog 68
TForm 26, 344
ActiveChild 727
ActiveControl 734
ActiveMDIChild 345, 709, 710
BorderIcons 51, 340, 343
BorderStyle 125, 340
Close 197
Constraints 342
DesignerHook 916
Ereignisse 345
FormStyle 343, 702, 704
Icon 344, 703
KeyPreview 344, 734, 738
MDIChildCount 345, 709
MDIChildren 345
MDIChilds 709
Menu 340, 707
ModalResult 127, 340, 344
MouseCapture 651
OnActivate 720
OnClose 348, 613
OnCloseQuery 76, 613
OnConstrainedResize 342
OnCreate 67, 345
OnPaint 660
OnQueryClose 346
OnResize 342
PixelsPerInch 344
Position 125, 344
Show 346
ShowModal 126
WindowState 344
TFrame 404
TFrameBasicObjectAttrs (TreeDesigner) 407
TFrameControl 581
TGradientObject (TreeDesigner) 633
TGraphic 449
Dateioperationen 449
Übersicht 450
TGraphicControl 294, 302, 303, 361
TGraphicDoc 626
Änderungs-Flag 629
ChangeCount 629
Dateiverwaltung 612
Deklaration 626
Dokument-View-Eigenschaften 618
FileFormatVersion 643
FileName 612
GetGraphicElement 630
GetObjectByPoint 656, 657, 745
Height/Width 631
Items 629
LoadFromFile 643
MarkedElementCount 629
Methodentabelle 631
PaintAll 662, 670, 696
Properties 628
SaveToFile 642, 650
SaveToXML 775
SaveToXMLFile 773
SelectedSetBrushColor 629
SelectedSetFontName 630
SetFileName 624
View-Verwaltung 623
TGraphicDocument
WriteToStream 494
TGraphicElement 632, 633, 646
Brush, Pen, Font 637
Content 634
Create 638
Deklaration 633, 646
GetGripRect 657
GraphicsObjectChanged 637
InvalidateAllViews 624
InvalidateRect 667
LoadFromXML 776
LocalExpand 648
Methoden-Übersicht 636
NormalizeRect (globale Prozedur) 668
Paint 663
PaintHexagon 664
PaintText 664
PointInShape 657
SaveEdgesToStream 650
SaveToStream 640, 650
SaveToXML 774
SetNewRect 666
SetPreviousNode 647
SetShape 637
SetTextClipRegion 677
XorPaint 653
TGraphicField 821
TGraphicView (TreeDesigner) 619, 622, 669
TGroupBox 161, 356

TGUID 278
THandleStream 424
THeaderControl 152, 157
THistoryCombo 922
THistoryList 414, 470, 471, 923
  und Menüs 472
  Verwendung 476
THotkeyEditor (Formular) 960
THotkeyManager 960
  Deklaration 965
  Übersicht 962
Threads 495
  anhalten 508, 517
  gleichzeitige Datenzugriffe 511
  primärer 503
  Prioritäten 508
  sicher anhalten 517
  Synchronisation 508
TIcon 58, 450
TIconView 133, 375, 380
  Drag & Drop 380
  FindItemByPoint 381
  Freigabe der Items 383
  mit TListView 383
  neue Klassen für Items 382
TIconViewItem
  eigene Klassen ableiten 382
TImage 151, 453
TImageList 388, 706
  Add-Methoden 388
  Bilder laden 606
  entwerfen 376
  Insert-Methoden 388
  Replace-Methoden 388
  transparente Bilder 388
Time 122
Timer 122
  Ereignisse 21
TimeToStr 122
TIniFile 418
TIntegerField 820
TInterfacedObject 273, 280, 722
TKdeHtmlView
  Implementierung 581, 586
  Verwendung 585
TLabel 130
  FocusControl 49
  in neuer Komponente 927
TLabelEditCombi (Beispielkomponente) 928
TLargeIntField 820
TLCDNumber 153
TList 416, 626
  abgeleitete Klassen 417
  Funktionsweise 416
  Methoden und Properties 416
TListBox 134
  besitzergezeichnete 455
  Drag&Drop 140
  Inhalt speichern 138
  ItemAtPos 141
  Items 135
  Mehrfachauswahl 136
  OnDrawItem 458
  OnMeasureItem 460
  zur Steuerung eines PageControls 362
TListItem 377
  SubItems 380
TListItemClass 160
TListItems 377
  Add 379
TListView 133, 157, 375
  dynamisch aufbauen 379
  Einträge Löschen 378
  entwerfen 158
  fokussierter Eintrag 379
  für Dateilisten 968
  Items 158
  mit TIconView 383
  MultiSelect 377
  OnEdited 161
  OnSelectItem 810
  Properties 378
  ReadOnly 160
  selbst definierte Item-Klassen 160
  transparente Bilder 388
  Vergleichstabelle zu TListBox 142
  ViewStyle 157
  vsReport-Modus 376
  zur Entwurfszeit 376
  zur Laufzeit 377
TMainForm
  BeforeRun 712
  CheckDockPanelVisibility 741
  ChildWindowActivate 720
  CMFileOpen 621
  Dokumentverwaltung 620
  FindDoc 620
  FormClose 743
  GetDoc 620
  MDINewClick 711
  MDIOpenClick 710
  NewWindow 710, 719, 721
  PaletteBrushColorChange 728
  Tile1Click 713
TMainForm.ChildWindowDestroy 622
TMainMenu 128, 465, 467
  AutoMerge 708
TMaskEdit 144
TMemIniFile 418
  Beispiel 419

TMemo 374
Alignment 374
Lines 374
Modified 68, 76
OnKeyDown 70
ReadOnly 69
ScrollBars 48, 374
WantReturns 374
WantTabs 375
TMemoField 821
TMemoryStream 433
Beispiel 494
TMenu 465
Items 53
TMenuItem 129, 465
Checked 615
Einfüge- und Löschoperationen 467
GroupIndex 708
Items 466
OnClick 475
Property-Liste 54
TMenuToolBar 727, 972
TMiniHistoryGraph
Verwendung 544
TModalResult 344
TMouseEvent 903
TMultiLineElement (TreeDesigner-Klasse) 764
TNotifyEvent 903
TNumEdit 898
TObject 209, 297, 887
ClassInfo 209, 212
ClassName 210
ClassName(Is) 209
ClassParent 209, 211
ClassType 209, 210
Destroy 196, 298
FieldAddress 310
Free 297
FreeInstance 298
GetInterface 210, 278
GetInterfaceEntry/Table 209
InheritsFrom 209, 211
InstanceSize 210
NewInstance 298
TObjectAttrBaseForm (TreeDesigner) 400
TObjectAttrDlg (TreeDesigner) 400
TObjectList 417
To-Do-Listen 102
ToolbarHost (TreeDesigner) 719
Toolbars
dynamische 724
TOpenDialog 612, 705
siehe TSaveDialog 369
TOrderedList 417
TPageControl 359
Ereignisse 359
Erzeugen aus Windows-Dialogen 363
Programmierung 360
Seitenauswahlverfahren 361
TPaintBox 303, 435, 609
TPaletteComponentEditor 950
TPaletteEditor 948
TPanel 162
Beispiel 74
Caption 918
TParam
ParamType 863
TPen 437, 639
Color 438
Mode 438, 653
OnChange 637
Übersicht 437
TPersistent 299, 887
Assign 299
AssignTo 299
DefineProperties 299, 910, 954
Polymorphie 955
TPersistentConfirmation 969
TPicture 452
verwenden 453
TPoint 323
TPopupMenu 129, 465, 467, 616
AutoPopup 617
OnPopup 493
TPosition 344
TPrinter 692, 699
Abort 692
BeginDoc 692
Canvas 692, 699
EndDoc 692
ExecuteSetup 697
NewPage 692
PageWidth/Height 700
Property-Tabelle 699
TPrinterSetupDialog
Ersatzkomponente 971
TProgressBar 156
TPropertyEditor
Methoden 947
TQtDialog 366
TQuery 790, 791, 795
TQueue 417
Trace 117
TrackBar 144
Tracking (Scrollbar-Einstellung) 675
TRadioButton 143
TRadioGroup 143
Transaktionen
bei ApplyUpdates 870

TReader 430, 483
  Methoden 431
  ReadListBegin 483
TRect 323
TreeArrangeOptionsDialog (TreeDesigner-Formular) 602
TreeDesigner 590
  Aktionsliste 615
  Attributdialoge 401
  Baumstrukturen 644
  Bedienung 593, 644
  Dateien 600
  Dateiformat-Versionen 644
  Dateiliste 471
  Docking 740
  Drucken 598, 693
  Einblenden der Symbolleisten 721
  Farbpalette 725
  Formular-Übersicht 601
  Gerätekoordinaten 672
  GlobalToolbar 729
  Größenänderung von Grafikelementen 657
  Größenbegrenzung der Zeichenfläche 687
  Klassenhierarchien anordnen 644
  Klassenhierarchien anzeigen 593
  Komponenten der Leisten 605
  Koordinatensystem 686
  ldd-Listing 530
  MDI-Version 704
  Objektinhalte 597
  PaintHexagon 664
  Popup-Menü der Farbpalette 944
  Projektquelltext 711
  SDI-Version 704, 728, 730
  Skalieren der Grafik (Scale) 632
  spezielle Funktionen 592
  Symbolleisten 605
  TreeExplorer-Bedienung 599
  Übersichtsfenster 599, 618
  Verbindungsaufbau 647
  Version 2.5 594
  Version 3.0 594
  Version 3.5 594
  WYSIWYG-Darstellung 693
  XML-Dateien 754
  XML-Dateien anzeigen 764
  XML-Speicherung 771
  Zwischenablageoperationen 494
TreeViews
  siehe TTreeView 390
TReplaceDialog 371
TResizeMode (TreeDesigner) 657
Trolltech 566
True 218
TrueType-Schriften 439
TRuler
  Verwendung im TreeDesigner 675
Trunc 215, 290
try 257, 258
TSaveDialog 369, 611
  Optionen 369
TScreen 317
  ActiveControl 326
  ActiveCustomForm 710
  Cursors 329
  Ereignisse 318
  OnActiveControlChange 318
  OnActiveFormChange 318
  Property-Tabelle 317
TScrollBar 163
  OnChange 688
TScrollBox 162, 339, 672, 674, 734
  HorzScrollBar 687
  Tastatursteuerung 735
  VertScrollBar 687
TScrollBoxEx 675, 734, 920
  OnScroll 675
  UpdateRect 671
TScrollingWidget 337
  AutoScroll 338
  HorzScrollBar 338
  ScrollInView 338
  VertScrollBar 338
TSearchRec 281
TShape 151, 437, 438
TShiftState 328, 656
TSmallIntField 820
TSpeedButton 130
  Glyph 607
  Gruppen 607
TSpinEdit 144
TSplitter 163, 325
  beim Docking 741
TSQLClientDataSet 795, 812
  CommandText 814, 862
  CommandType 814
  DBConnection 794
  siehe auch TClientDataSet 806
  SQL-Abfragen 859
TSQLConnection 792, 812
  Connect 864
  Connected 812
  ConnectionName 814
  Execute 878
  GetTableNames 813
  LoadParamsOnConnect 794
  Params 813
  TableScope 813
TSQLDataSet 796, 878
TSQLMonitor 881

TSQLQuery 859
TSQLTable/Query 796
tsTabs u.$%$a. Konstanten 361
TStack 417
TStateSaver 421
  Beispiel 810
  erweitern 422
  mit TPersistentConfirmation 969
TStatusBar 155
TStoredProc 791
TStream 424, 482
  Beispiel 150
  Position 425
  Read 425
  Read/WriteComponent 426
  ReadBuffer 425
  Seek 425
  Size 425
  Write 425
  WriteBuffer 425
TStringField 782, 820
TStringGrid 448
TStringList 410
  Duplicates 410
  OnChange 410
  Sorted 410
  Strings 473
  Übersicht TStrings 413
TStrings 135, 410
  AddObject 412
  CommaText 411
  Dateioperationen 67
  DelimitedText 411
  Methoden 413
  Übersicht 413
TStyle 314
TSuspendableThread 520
TSysInfo 543
TTable 790, 791, 795
TTabPage 364
TTabSheet 359, 361, 364
TTextBrowser 154, 337, 577
TTextViewer 153
TThread 503
  dynamische Verwaltung 523
  Execute 504
  FreeOnTerminate 525
  OnTerminate 525
  Suspend verbessern 517
  Synchronize 505
  Terminated 504
  Übersicht 507
  WaitFor 510
TThreadList 521
TThreads
  siehe auch Threads 508
TTimeField 820
TTimer 122, 497
  Interval 122
  verwenden 122
TToolBar 130, 604, 706
  für Menüs 972
TToolButton 130, 604
  Gruppen 607
  und Menüs 973
TTreeItems
  Add-Methoden 391
  SetNodeClass 392
TTreeNode
  AbsoluteIndex 362
  eigene Klassen ableiten 392
  Index 362
  SelectedIndex 390
TTreeNodes 391
TTreeView 133, 390
  Einträge hinzufügen 391
  für Verzeichnisbäume 966
  HasChildren 393
  Item-Indizes 362
  Komponentenbeispiel 967
  OnExpanding 393
  selbst definierte Node-Klassen 392
  ShowButtons 393
  zur Entwurfszeit 391
  zur Steuerung eines PageControls 362
TUpDown
  Ersatzkomponente 971
Turbo Pascal 165
TVarRec 249
TVerlaufObjAttrDlg(TreeDesigner) 400
TVirtualCanvas 680
  ClippingOff 678
  DPtoLP 689
  LPtoDP 690
  mmLoMetric-Implementation 701
  SetClippingEllipse 677
  SetMapMode 683, 701
  SetTextAlign 664
  SetViewportOrg/Ext 683
  SetWindowOrg/Ext 683
  TextRect 665
TWideStringField 820
TWidgetControl 302, 311, 320, 327, 336, 887
  als Basis für neue Widget-Komponenten 583
  ContainsControl 336
  ControlAtPos 336
  ControlCount 311
  Controls 311, 911

CreateWidget 294, 582
DestroyWidget 582
DisableAlign 325
DoEnter 907
DoExit 907, 924
EnableAlign 325
Ereignismethoden 907
EventFilter 306, 908
FindNextControl 337
Focused 336
Handle 302, 337
HookEvents 305
InsertControl 312
Invalidate 660
InvalidateRect 660
KeyDown 907
MainEventFilter 305, 917
OnKeyDown 327
OnKeyPress 327
OnKeyUp 327
Parent 311
RemoveControl 312
ScaleBy 337
ScrollBy 337
SetFocus 336, 735
TabOrder 336
X11-Fensterhandle 546
TWindowState 344
TwoDigitYearCenturyWindow 286
TWordField 820
TWriter 430, 482, 483, 910
Methoden 431
WriteListBegin 482
Typdeklarationen 176
Typen 168, 175, 213
Arrays 222
Aufzählungstypen 217
boolesche 218
Dateitypen 283
einfache 213
Fließkommatypen 215
generische und fundamentale 214
in C und Object Pascal 535
Integertypen 214
ordinale 213, 218
Prozedurtypen 253
Strings 225
strukturierte 232
Teilbereichstypen 217
Zeichen 216
Zeiger 234
TypInfo-Unit 212
Typkompatibilität 237
Typumwandlung 237, 629
Beispiel 352

U

Überladen von Funktionen 252
Überlaufprüfung 261
Überwachung von Ausdrücken 115
UInt64 536
ukModify (und andere) 877
Umgebungsoptionen
Explorer 83
Umgebungsvariablen 563
Unabhängigkeit
von Formularen 730
Ungültigerklärung 660, 669
Unicode 216
UniqueString
Zusammenhang 229
unit 180
Units 89, 180
Aufbau 180
frühestmögliche Initialisierung 584
für Formulare 65
gegenseitiges Einbinden 180
until 241
UpCase 290
UpperCase 288
uses 66
usModified (und andere) 873

V

var 175, 247
var/log/messages 110
VarArrayOf 841
Variablen 64, 175
für Formulare 64
globale 64
im Debugger 114
initialisierte 177
lokale 178
Variablen-Parameter 246
VCanvas 680
VCL
Kompatibilität der CLX 293
VER140 (Compilersymbol) 174
Vererbung 191, 632
von Formularen – siehe Formularvererbung 397
von Frame-Eigenschaften 404
Vergrößerung
von Grafik 678
Verzeichnisbäume
anzeigen 966
Viewport (Koordinatentransformation) 678
virtual 200
Visual Basic
siehe Basic 168
Visuelle Formularverknüpfung 376

VMTs 202
ABI 564
oid (C-Typ) 536
Vorwärtsdeklarationen
von Prozeduren 245
vsList 157
vsRecord 157
vtInteger und andere 249

W

wait4 (Libc-Funktion) 558
Wartbarkeit 182
Warteschlange
für Nachrichten 304
Watch-Fenster 115
Weltkoordinatensystem 569, 672, 678, 685
Werkzeugleisten 603
auslagern 725
dynamische 716, 724
globale 725
in Elternfenster einblenden 723
mit TPanel 603
mit TToolBar 604
Wertparameter 246
WEXITSTATUS 787
while 242
WideChar 216, 231
WideString 225, 232
Beispiel 568
Widgets (Qt)
Tabelle 294
Wiedereintrittsfähigkeit 511
Wiederverwendbarkeit 182
Wiederverwendung
von Formularen 95
WIFEXITED 787
Win32 (Compilersymbol) 174
Window (Koordinatentransformation) 678
Windows
Geschwindigkeit 347
Wirth, Niklaus 175
with 243
Word 214
WordBool 218
write-Direktive 187
writeln 251, 691
auf Konsole 549
wsNormalized und andere 344
Wurzelfenster 25
WYSIWYG-Darstellung 693

X

X Window 293
Cursor-Formen 331
Display 313
Display-Daten 545
Ereignisse protokollieren (MessageDemo) 308
Expose 660
Fenster-Benchmark 347
Fenstermanager 323
Netzwerkarchitektur 544
Programmierung 545
und CLX 292
XNextEvent 304
Zwischenablage 488
xfm-Dateien 90, 310
Xlib 296, 545
XML 753
als Grafik im TreeDesigner 764, 765
Attribute setzen 774
Dateien einlesen 775
Dateien speichern 772
DTD 756
Gültigkeit von Dokumenten 756
Markup 756
Probleme beim Überschreiben von Dateien 776
Schemas 756
selbst definierte Dateiformate 771
siehe auch DOM 759
Tags 756
xor 219, 220
XPaint 57
XSetInputFocus 545

Z

Zeichen
konstante Angaben 170
Zeichenereignisse 659
Zeichenmodus 438
Zeichenwerkzeuge 436
verwenden 440
Zeichnen
mit der Maus 650
Zeiger 234
auf Prozeduren 253
Dereferenzieren 236
in C 539
speichern 649
untypisierte 236
Zuweisungen 235
Zeitfunktionen 285

Ziehvorgänge 650, 655
Zielkomponenten
für Aktionen 487
Zoomen 687
Z-Reihenfolge 321
Zugriffsschutz 917
Zuweisung 63
Zuweisungskompatibilität 237
von Objekten 198
Zwischenablage 488
Formate 491
Grafiken kopieren 491
selbst definierte Formate 493
technisch 488
Text kopieren 491
und Editierfelder 374
Zyklen (in Graphen) 645



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