Andreas Doberstein
Georg Rauter

Delphi 6 für Profis

Galileo Computing

Inhalt

Vorwort 9

1 Organisation der Softwareentwicklung 15

1.1 Einleitung 15
1.2 Teamarbeit oder Einzelkämpfer? 15
1.3 Organisatorischer Aufwand 17
1.4 Einsparpotentiale in der Organisation 18
1.5 Definition der Projektziele 19
1.6 Besprechungen 29
1.7 Terminpläne 36
1.8 Störeinflüsse 44
1.9 Motivation und Weiterbildung 46

2 Softwareentwicklung 49

2.1 Kapazitäten planen 49
2.2 Anforderungen ermitteln 50
2.3 Rapid Prototyping 58
2.4 Strukturierung der Software 60
2.5 Programmierfehler 78

3 Neuerungen in Delphi 6 83

3.1 Erweiterungen von Delphi 83

4 Applikationsdesign 89

4.1 Was bedeutet Applikationsdesign? 89
4.2 Oberflächengestaltung 95
4.3 Weitere Aspekte zur Softwaregestaltung 105
4.4 Zusammenarbeit mit anderen Anwendungen 109
4.5 Abhängigkeit vom Einsatzbereich 110

5 Allgemeine Delphi-Kenntnisse 111

5.1 Programmierung mit Delphi 111
5.2 Anwendungstypen 121

5.3 Vorgehensweise bei der Entwicklung von Delphi-Anwendungen 121
5.4 Weitere Einstellungen 151
5.5 Komponenten und Packages 154
5.6 Entwicklung plattformübergreifender Anwendungen 160

6 Allgemeine Programmierthemen 169

6.1 Schnellkurs: Pascal in Delphi 169
6.2 Fehlerbehandlung und Programmsteuerung mit Exceptions 237
6.3 Fehlersuche 244
6.4 Objektorientierte Programmierung 254
6.5 Die Klasse TPersistent 280
6.6 Komponenten 284
6.7 Arbeiten mit Listen 291
6.8 Laufzeit-Typinformationen 297
6.9 Arbeiten mit Dateien 310
6.10 Streams 334

7 Programmierung für Fortgeschrittene 351

7.1 Struktur und Verhalten von VCL- Anwendungen 351
7.2 Botschaftsbehandlung unter Windows 361
7.3 Prozeßtheorie und Multithreading unter Windows 392
7.4 DLLs 413
7.5 Interprozeßkommunikation mit Memory-Mapped-Files 431
7.6 Versionsinformationen 439
7.7 Mehrsprachigkeit 441

8 Verstehen der VCL-Komponenten 447

8.1 Arbeiten mit Formularen 448
8.2 Erstellung einer MDI-Anwendung 473
8.3 Datenmodule 484
8.4 Verstehen der VCL: Hierarchische Betrachtung 485
8.5 Verwenden der VCL: Funktionelle Betrachtung 502

9 VCL für Fortgeschrittene 613

9.1 Aktionslisten: TActionList 613
9.2 Symbolleiste 623
9.3 Anpassen von Menü und Werkzeugleisten 635
9.4 Baum- und Listenansichten 638
9.5 Tabellen 644
9.6 Drag&Drop 648
9.7 Docking 654

10 Komponentenentwicklung 665

10.1 Einführung 665
10.2 Erstellen einer neuen Komponente 668
10.3 Fehlersuche in Komponenten 676
10.4 Komponentenschnittstelle 677
10.5 Designrichtlinien 711
10.6 Verhalten von Komponenten 713
10.7 Speichern von Komponenten 716
10.8 Eigenschafts- und Komponenteneditoren 719
10.9 Message-Verarbeitung in Komponenten 731
10.10 Weitergabe von Komponenten 734
10.11 Praxisbeispiele 739
10.12 Der ApplicationExplorer 769

11 COM 777

11.1 Einführung 777
11.2 Das COM-Konzept 784
11.3 COM und Delphi 792

Index 813

Vorwort

Eine Idee entsteht

In unserer beruflichen Praxis haben wir mannigfaltige Erfahrungen in der Programmierung mit Delphi gewinnen können. Neben vielen positiven Erfahrungen zählen dazu auch solche, auf die wir lieber verzichtet hätten. Zugegeben, diese Feststellung bedarf einer weiterführenden Erläuterung.

Einer der wichtigsten Faktoren im Wirtschaftsleben ist Effizienz. Um einigermaßen effizient zu arbeiten, sollte man – wie man so schön sagt – das Rad nicht jedesmal aufs neue erfinden. Das haben auch wir uns zu Herzen genommen und zu all unseren täglichen Aufgabenstellungen zunächst die uns zur Verfügung stehende Literatur befragt. Viel zu oft mußten wir dabei feststellen, daß die vielen Delphi-Bücher gerade unsere speziellen Probleme nicht ausführlich genug behandeln. Dadurch waren wir gezwungen, selbst nach Lösungen zu suchen und Wege zu beschreiten, die gepflastert sind mit Arbeitszeit, Programmiererschweiß und schlaflosen Nächten (das sind die eingangs angesprochenen Erfahrungen, auf die zu verzichten uns nicht schwer gefallen wäre).

So wurde der Gedanke geboren, endlich jenes Buch zu schreiben, das alle bestehenden Lücken schließt. Schnell wurde uns jedoch klar, daß es unmöglich sein wird, auf alle sich ergebenden Fragen Antworten zu geben und für alle erdenklichen Problemstellungen Patentrezepte präsentieren zu können. Also wieder nur eines jener Delphi-Bücher, die viele Themen anschneiden, ohne sie bis ins Detail zu behandeln oder unzählige Lösungen zu Problemen bieten, die sich leider gerade nicht stellen? Nein!

Dieses Buch beschreitet in mehrerlei Hinsicht neue Wege. Den Anfang machen praxisnahe Betrachtungen zur Softwareentwicklung im allgemeinen, die den Grundstein für eine immer wiederkehrende, strukturierte Vorgehensweise legen. Der wohl grundlegendste Unterschied zu den bisher bekannten Darstellungen ist die Betrachtung der Komponentenbibliothek, die sich mehr an der natürlichen Entstehung der VCL als an den fertigen Endprodukten orientiert. Es werden zuerst die Basisklassen ausführlich vorgestellt, dann folgen die davon abgeleiteten Klassen und so weiter, bis man am Ende zu den jeweiligen Komponenten gelangt. Diese Betrachtungsweise spiegelt den Aufbau der VCL wider und soll Einblicke in deren Innenleben ermöglichen. Die umfangreichen Programmierbeispiele zeigen, wie die dargestellten Techniken in der Praxis angewendet werden können. Dabei versuchen wir nicht, eine Lösung für ein spezielles

Problem zu präsentieren, vielmehr wollen wir die Techniken vermitteln, die Sie bei Ihren ganz speziellen Aufgabenstellungen anwenden können. Ein thematischer Schwerpunkt, dem Sie in allen Abschnitten begegnen werden, ist die Wiederverwendbarkeit von Software. Aber langsam und der Reihe nach.

Der Inhalt

Softwareentwicklung beginnt bereits lange vor der ersten Zeile Programmcode. Diese Tatsache wird unserer Meinung nach in vielen Fällen zu wenig gewürdigt. Wir haben die wichtigsten Aspekte des organisatorischen Umfelds herausgegriffen und in komprimierter Form dargestellt.

Wir haben uns auch nicht davor gescheut, Ihnen Themen in Erinnerung zu rufen, die auf den ersten Blick eher trivial erscheinen. In Kapitel 4, »Applikationsdesign«, werden Punkte angesprochen, die vermutlich sogar weniger erfahrenen Softwareanwendern als selbstverständlich erscheinen werden. Darin liegt unserer Meinung nach aber auch der Grund dafür, daß man viele der grundlegenden Erscheinungsmerkmale von Software nicht mehr bewußt wahrnimmt und damit auch leicht dazu neigt, diese bei der Implementierung zu übersehen. Gerade auf solche Punkte wollen wir gezielt hinweisen und Ratschläge geben, wie die Qualität verbessert werden kann.

Im darauf folgenden Kapitel 5, »Allgemeine Delphi-Kenntnisse«, schlagen sich die eigenen Praxiserfahrungen besonders nieder. Betrachten Sie das Kapitel nicht aus der Sicht »Welche Möglichkeiten bietet Delphi?«. Sehen Sie das Kapitel eher als Ratgeber, wie bestimmte Features des Entwicklungswerkzeugs in der Praxis vorteilhaft genutzt werden können. Üblicherweise sind die optimale Konfiguration und Anwendung der Entwicklungsumgebung ein ständiger Prozeß. Sie werden über kürzer oder länger feststellen, wo bei der Entwicklung unnötig viel Zeit verlorengeht. Dementsprechend erfolgt eine Optimierung der Arbeitsabläufe, der Systemkonfiguration und der Vorgehensweise bei der Entwicklung. Oft stellt man erst mit zunehmender Erfahrung fest, wie sich die Stärken der Entwicklungsumgebung am sinnvollsten nutzen lassen. Genau diesen Prozeß soll das Kapitel abkürzen. Wir beginnen es bewußt mit einigen sehr grundlegenden Erklärungen, um auch dem Umsteiger von anderen Programmiersprachen den Einstieg in Delphi und die spezielle Terminologie zu erleichtern.

Der Vollständigkeit halber wird die eigentliche Programmierung durch einen kurzen Abriß der Programmiersprache eingeleitet, der für den

erfahrenen Programmierer zum Großteil nichts Neues darstellt. Dennoch enthält die fundierte Betrachtung der verschiedenen Datentypen den einen oder anderen nützlichen Hinweis.

Nachdem Delphi dem Entwickler in vielen Fällen den direkten Kontakt mit dem Betriebssystem erspart, neigt man in der Praxis schnell dazu, die besonderen Gegebenheiten des Systems **Windows** zu übersehen. Man kann sich die Fähigkeiten des Betriebssystem allerdings hervorragend zunutze machen, um die Performance und das Verhalten von Programmen zu verbessern. Mit Themen wie Windows-Botschaften, Multithreading, Windows-API-Funktionen und der Programmierung von DLLs wird ein Einblick in die Funktionsweise der verschiedenen 32-Bit-Windows-Versionen gegeben.

Besonders in größeren Softwareprojekten wird man häufig dem Fall begegnen, daß Delphi-Programme im Verbund mit anderen Anwendungen eingesetzt werden. Selbst wenn die Implementierung einzelner Teile des Projekts mit verschiedenen Programmiersprachen realisiert wird, ist es erstrebenswert, bereits vorhandene Funktionalität unabhängig von ihrem Ursprung in allen Programmierumgebungen zur Anwendung zu bringen. Das ist allerdings nur bei Beachtung der dargestellten Grundregeln möglich.

Selbstverständlich zeigen wir auch die intensive Verwendung der VCL. Neben der reinen Anwendung von Komponenten wird so mancher Blick hinter die Kulissen das detaillierte Verständnis der Komponenten fördern und Ihnen die internen Vorgänge in Delphi-Anwendungen verständlich näherbringen. Nicht zuletzt zeigt das Kapitel 10, »Komponentenentwicklung«, wie Sie eigene Komponenten erstellen können. In diesem Kapitel soll Ihnen ein Gefühl dafür vermittelt werden, wie sich Ansatzpunkte finden lassen, um das Verhalten bestehender Komponenten zu erweitern, dabei aber so umsichtig vorzugehen, daß das Standardverhalten der Komponenten nicht unerwünscht aus dem Gleichgewicht kommt. Weiter wollen wir Ihnen Denkansätze an die Hand geben, die geeignete Basisklasse für neue Komponenten zu finden.

Ein weiterer Aspekt, der die Verwendung von Delphi interessant erscheinen läßt, ist die Möglichkeit der plattformübergreifenden Softwareentwicklung. Im Klartext bedeutet das, daß Sie Anwendungen entwickeln können, die sowohl in der Windows-Umgebung als auch unter Linux verwendet werden können. Dabei sind gewisse Einschränkungen zu beachten, auf die wir an den entsprechenden Stellen hinweisen. Sie finden diese Abschnitte gesondert gekennzeichnet.

Alle Programmbeispiele sind auf der beiliegenden CD enthalten und können somit unmittelbar nachvollzogen werden.

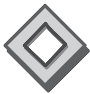
Danke für die Unterstützung

Wir wollen die Gelegenheit nutzen und uns bei der Firma GEFASOFT AG in München bedanken, die durch ihre Hilfe und Unterstützung zur Entstehung dieses Buchs beigetragen hat. Sie hat uns Arbeitszeit, Geräte und Software zur Verfügung gestellt.

Symbole

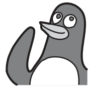
Manche Abschnitte in diesem Buch sind durch **Symbole** besonders gekennzeichnet. Diese Symbole haben die folgende Bedeutung:

Wichtige Hinweise und Tips finden Sie in Abschnitten, die mit diesem Symbol gekennzeichnet sind.



Dieses Symbol macht Sie auf Beispiele aufmerksam.

Abschnitte mit diesem Symbol sprechen eine Warnung aus!



Auf die Pinguine sollten Sie achten! Die so markierten Stellen kennzeichnen Hinweise und Anleitungen zur Entwicklung von Delphi-Programmen, die auf Kylix portierbar sind.

Ihr Feedback

Wir sind an Ihrer Meinung interessiert. Wie gefällt Ihnen das Buch? Entspricht es Ihren Erwartungen? Bitte lassen Sie uns wissen, was wir beim nächsten Mal besser machen können, oder ob Ihnen eine Idee besonders gut gefallen hat. Nur indem Sie uns Ihre Meinung und Erfahrungen mitteilen, können wir in Zukunft besser auf Ihre Bedürfnisse eingehen.

Ihre Meinung erreicht uns am schnellsten per E-Mail unter **delphi @gefasoft.de**. Selbstverständlich können Sie auch gerne die Post bemühen:

GEFASOFT AG
Andreas Doberstein
Landshuter Allee 94
D-80637 München

München, im September 2001
Andreas Doberstein & Georg Rauter

3 Neuerungen in Delphi 6

Alle Umsteiger von älteren Delphi-Versionen wird zunächst einmal interessieren, was Delphi 6 neues zu bieten hat. Dieses Kapitel gibt einen Überblick darüber, welche Möglichkeiten sich durch Delphi 6 eröffnen.

Gegenüber der Version 5 sind beim ersten Blick auf die Entwicklungsumgebung keine großen Änderungen erkennbar. Das bietet für den mit der bestehenden Oberfläche vertrauten Programmierer Vorteile, weil keine Umstellung erforderlich ist. In einigen Punkten wurde jedoch die IDE mit zusätzlichen Funktionen ausgestattet und verbessert.

Die eigentlichen Verbesserungen liegen eher im Verborgenen, wobei zwei Teilbereiche besonders hervorgehoben werden müssen. Zum einen eröffnet die Möglichkeit der plattformübergreifenden Entwicklung ganz neue Perspektiven, zum anderen sind im Lieferumfang zusätzliche Komponenten zu den Themenbereichen Internet, Web und Übertragungsprotokolle enthalten.

Die Umsetzung bestehender Projekte konnte in allen der getesteten Fälle problemlos durchgeführt werden.

3.1 Erweiterungen von Delphi

3.1.1 Änderungen an der IDE

Verbesserungen an der IDE

Obwohl bereits die IDE der Vorgängerversionen über viele Funktionen verfügte, die dem Programmierer die Arbeit erleichterte, ist es Borland erneut gelungen, den Funktionsumfang und damit den Bedienkomfort zu erhöhen.

Herausragend ist die neu hinzugekommene Objekthierarchie, welche die Anordnung der Komponenten in Form eines Baumdiagramms wiedergibt. Mittels der Objekthierarchie können Sie einfach mit Drag&Drop die Hierarchie visueller Komponenten verändern. Fügen Sie zum Beispiel in ein Formular einen Button und später ein Panel ein, können Sie nachträglich noch den Button dem Panel unterordnen, so als ob Sie den Button gleich auf dem Panel plaziert hätten. So wie bereits bisher der Formulardesigner und der Objektinspektor miteinander synchronisiert waren, fügt sich hier die Objekthierarchie nahtlos ein. Das heißt, wenn Sie in einem der drei Werkzeuge ein Element auswählen, wird es automatisch auch in den anderen ausgewählt.

Abbildung 3.1 Ausschnitt derObjekthierarchie eines Beispielprojekts

Neu ist, daß der Quelltexteditor (bisher bekannte Ansicht auf der Registerkarte »Code«) zusätzlich mit einer Registerkarte »Diagramm« versehen ist. Alle Arten von in einem Formular verwendeten Komponenten können im Diagramm abgebildet werden. Bezüge der Komponenten untereinander können mittels Verbindern dargestellt werden. Komponenten können einfach mittels Drag&Drops aus der Objekthierarchie in das Diagramm eingefügt werden. Bestehende Bezüge werden automatisch mit Verbindern gekennzeichnet. Im Diagramm neu erzeugte Verbindungen werden automatisch im Objektinspektor als Eigenschaft eingetragen. Die Darstellung kann beliebig durch Kommentarblöcke und Dokumentationsverbinder ergänzt werden. Mit der Möglichkeit diese auszudrucken, wird dem Entwickler ein praktisches Werkzeug zur Dokumentation an die Hand gegeben. Insbesondere Beziehungen zwischen nichtvisuellen Komponenten (z. B. Datenbankanbindung) können so auf ausgezeichnete Art und Weise sichtbar gemacht werden.

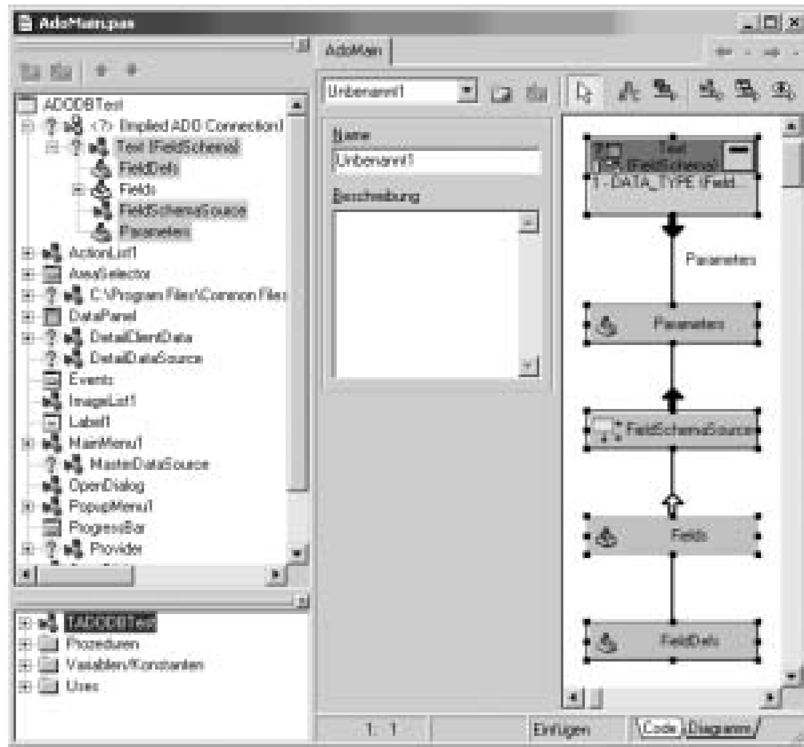


Abbildung 3.2 Übernahme einiger verbundener Komponenten in die Diagrammansicht

Die Funktionalität des Objektinspektors wurde ebenfalls erweitert. Wenn eine Eigenschaft einer Komponente auf eine andere Komponente verweist, kann über den Objektinspektor direkt auf die Eigenschaften der untergeordneten Komponente zugegriffen werden. Die zusätzlichen Felder lassen sich über den bereits bekannten Mechanismus (z.B. Font+Style) öffnen bzw. verbergen. Durch die farbliche Kennzeichnung der Eigenschaften, mit denen Komponenten verbunden sind (dunkelrot), sowie der Eigenschaften untergeordneter Komponenten (grün) bleibt die Übersichtlichkeit gewahrt.

3.1.2 Plattformübergreifende Entwicklung

Delphi 6 bietet die Möglichkeit, Anwendungen zu entwickeln, deren Quellcode auch für die Zielplattform Linux übernommen werden kann. Wenn diese Portierbarkeit angestrebt wird, müssen jedoch einige Einschränkungen in Kauf genommen werden. So darf zum Beispiel nicht mehr die VCL-Bibliothek eingesetzt werden. Dafür steht jetzt eine CLX-

Bibliothek zur Verfügung, die auch mit den noch geplanten Erweiterungen nicht alle Funktionen der VCL ersetzten kann. Selbstverständlich dürfen im Interesse der Kompatibilität Windows-API-Aufrufe nicht verwendet werden. Zur besseren Unterscheidung werden die Formulardateien in CLX-Projekten mit der Endung .XFM gespeichert.

Für Neuentwicklungen wird man sicherlich Lösungsansätze finden, die diese Einschränkungen berücksichtigen. Während man bei selbstgeschriebenem Code die Kompatibilitätsanforderungen leichter kontrollieren kann, wird man Komponenten von Drittanbietern unter diesem Gesichtspunkt wohl genauer unter die Lupe nehmen müssen. Was sich generell empfiehlt, jedoch besonders wenn eine portierbare Anwendung angestrebt wird, ist der Einsatz von Komponenten deren Quellcode verfügbar ist.

3.1.3 Sonstige Neuerungen

Webanwendungen

Delphi 6 unterstützt die Entwicklung von Webservern und Webservices. Die entsprechenden Schablonen werden in der Objektgalerie zur Verfügung gestellt.

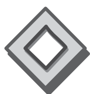
XML-Dokumente

Die mitgelieferte Bibliothek verfügt über Komponenten, welche die Behandlung von XML-Dokumenten erleichtern.

Mehrsprachige Anwendungen

Erwähnenswert erscheint auch die Unterstützung zur Erstellung mehrsprachiger Anwendungen, die (bereits seit Delphi 5) vollständig in die Entwicklungsumgebung integriert ist. Die Funktionen sind unter dem Begriff »Translation-Suite« zusammengefaßt. Sobald Sie Ihrer Anwendung die entsprechenden Sprachen hinzugefügt haben, läßt sich der Translation-Manager aktivieren, der alle für die Übersetzung relevanten Eigenschaften und Ressourcen anbietet. Erfreulicherweise sind das nicht nur die Texteigenschaften, sondern auch Eigenschaften wie Images oder Größen- und Positionsangaben von Elementen. Bei Bedarf kann somit auch die Größe und Position von Komponenten von der gewählten Sprache abhängig gemacht werden.

Der Translation-Manager ist ein eigenständiges Programm und kann auch außerhalb der IDE ausgeführt werden. So kann der Translation-Manager für Übersetzungen verwendet werden, ohne daß Delphi installiert werden muss.



Der Umfang der mitgelieferten Beispiele wurde erheblich erweitert. Praktisch zu allen Themen werden funktionsfähige Beispiele mitgeliefert, die natürlich auch die Einarbeitung in neue Technologien wesentlich erleichtern.

Delphi 6 ist in drei verschiedenen Versionen erhältlich, welche sich im Funktionsumfang und daher auch im Preis unterscheiden. Der folgende Überblick soll die Entscheidung bei der Auswahl der Version erleichtern.

Eine detaillierte Aufstellung darüber, welche Funktionen im Lieferumfang der jeweiligen Ausbaustufe enthalten sind, kann der **Delphi-Produktmatrix** auf der Borland-Homepage (www.borland.de) entnommen werden.

3.1.4 Die verschiedenen Versionen

Delphi Personal ist die kleinste Ausbaustufe und kann eigentlich nur für Ausbildungszwecke sinnvoll genutzt werden. Der Umfang der mitgelieferten Komponenten ist stark eingeschränkt, die CLX-Bibliothek wird nicht mitgeliefert und mit dieser Version erstellte Applikationen dürfen entsprechend der Lizenzvereinbarungen nicht kommerziell genutzt werden.

Delphi Professional ist die mittlere Ausbaustufe, die für eine kommerzielle Nutzung geeignet ist. Diese Version ist dann für Ihre Zwecke ausreichend, wenn Sie keine der nur in der Enterprise-Version verfügbaren Funktionen benötigen.

Delphi Enterprise beinhaltet sämtliche Funktionen und Hilfsmittel, die Delphi bieten kann. Im folgenden sind die wichtigsten Unterschiede gegenüber der Professional-Version aufgelistet:

- BizSnap: Webdienste mit XML-Technologie
- WebSnap: Webserver-Funktionalitäten
- DataSnap: mehrschichtige Datenbankanwendungen
- CORBA: mehrschichtige Datenbankanwendungen
- InternetExpress: Webserver-Funktionalitäten
- TeamSource: Versionsverwaltung und Koordinierung mehrerer Entwickler an einem Projekt
- Erweiterte Unterstützung beim Zugriff auf SQL-Datenbanken
- Treiber und Verbindungen zu verschiedenen SQL-Datenbanken

Zusammenfassung

Für den professionellen Einsatz ist mindestens die Professional-Version erforderlich. Wegen des hohen Preisunterschieds zwischen Professional- und Enterprise-Version ist es angebracht zu prüfen, ob die zusätzlichen Funktionen der Enterprise-Version benötigt werden. Für die Entwicklung von Internetanwendungen oder mehrschichtigen Datenbankanwendungen führt jedoch an der Enterprise-Version kein Weg vorbei.

9 VCL für Fortgeschrittene

Während im vorangegangenen Kapitel eine strukturierte Vorgehensweise für ein besseres Verständnis der VCL-Hierarchie im Vordergrund stand, soll dieses Kapitel Techniken zur Entwicklung von Anwendungen mit der VCL zeigen. Dazu werden weitere Komponenten vorgestellt, die aufgrund ihrer Komplexität besondere Anforderungen an die Programmiertechnik stellen. Einige Komponenten werden auch praktisch angewendet, die bislang eher unter theoretischen Gesichtspunkten dargestellt wurden.

Die zur Lösung der Aufgabenstellungen erforderlichen Maßnahmen werden Schritt für Schritt erklärt. Für ein besseres Verständnis empfehle ich, die Schritte mit Delphi parallel zur Lektüre dieses Kapitels nachzuvollziehen. Die fertige Lösung ist immer in den zugehörigen Programmbeispielen auf der CD enthalten.

9.1 Aktionslisten: TActionList

Vererbung	TObject→ TPersistent→ TComponent→ TCustomActionList→ TActionList
Palettenseite	Standard
Deklariert in Unit	ActnList

Vererbung (CLX)	TObject→ TPersistent→ TComponent→ TCustomActionList→ TActionList
Palettenseite	Standard
Deklariert in Unit	QActnList

9.1.1 Einführung

TActionList

TActionList ist eine Komponente, die den Anwendungsentwurf hinsichtlich des Aufbaus von Menüs und Symbolleisten erheblich vereinfachen kann. Als Beispiel habe ich angenommen, daß eine Anwendung entwikkelt werden soll, die über ein Hauptmenü, verschiedene Kontextmenüs und eine Symbolleiste verfügt. Dabei ist es üblich, daß dieselbe Funktion auf verschiedene Arten ausgelöst werden kann, daß sie also zum Beispiel im Hauptmenü, im Kontextmenü und in der Symbolleiste ausgewählt werden kann. Die Buttons in der Symbolleiste verfügen über Icons, die in

einer Bilderliste enthalten sind. Genauso können die Menüs über Symbole verfügen, die möglicherweise aus derselben Bilderliste stammen. Menüeinträge erhalten eine Beschriftung und einen Shortcut, wobei die Einträge für das Hauptmenü und die Popup-Menüs gesondert erstellt werden müssen. Für die Buttons und Menüeinträge gibt es dann abhängig vom aktuellen Zustand des Programmes verschiedene Status, wie etwa **enabled/disabled** oder **gedrückt/nicht gedrückt** bzw. **ausgewählt/nicht ausgewählt**. Bei der herkömmlichen Vorgehensweise müssen diese Status explizit im Quelltext gesetzt werden, und zwar für jedes Element, das der Funktion entspricht. Alle Elemente, die dieselbe Funktion auslösen, werden über dieselbe Ereignisbehandlung verfügen. Das bedeutet, daß die Ereignisse aller Elemente ausgewertet werden müssen.

Wer schon einmal ein komplexeres Programm entwickelt hat, wird wissen, daß die Erstellung der Oberfläche auf diese Art und Weise äußerst zeitaufwendig und mühsam ist. Diesen Arbeitsaufwand haben die Delphi-Entwickler erkannt und mit der Komponente **TActionList** eine sehr flexible Möglichkeit geschaffen, um diesen Entwicklungsschritt zu vereinfachen.

Mit der Komponente **TActionList** ist es möglich, Aktionen zu definieren. Diese Aktionen entsprechen normalerweise Programmaktionen, wie zum Beispiel **Datei öffnen**, **Datei speichern**, **Text ausschneiden**, **Kopieren** oder **Einfügen**. Für jede Aktion sind bestimmte Eigenschaften festgelegt, wie zum Beispiel die Bezeichnung der Aktion (z.B. »Datei öffnen«), ein Icon zur Darstellung der Aktion und ein Shortcut, um die Aktion über die Tastatur auszulösen. Ebenso zeigt die Aktion den Status dieser Programmaktionen an, ob also die Aktion beispielsweise verfügbar ist und ausgeführt werden kann. Jede Programmfunktion, die vom Benutzer über das Menü oder die Symbolleiste aufgerufen werden kann, wird somit als **Aktion** ausgeführt.

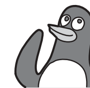
Eigenschaft Action

Letztlich müssen die Menüpunkte und Toolbuttons über die Eigenschaft **Action** nur noch der jeweiligen Aktion zugeordnet werden. Sie übernehmen dann automatisch die Eigenschaften der Aktion. Die Bezeichnung der Aktion wird als Menüeintrag verwendet, das Symbol wird zu dem Menüpunkt genauso wie für die Buttons angezeigt. Gleichfalls wird der Menüeintrag oder Toolbutton deaktiviert dargestellt, sofern die Aktion gerade nicht ausgeführt werden kann. Im Quelltext muß nicht mehr auf jeden Menüeintrag oder Toolbutton gesondert reagiert werden. Es genügt, lediglich die Ereignisse der Aktion auszuwerten und dementsprechend zu reagieren. Die Aktion sorgt selbständig dafür, daß alle an die

Aktion angeschlossenen Menüpunkte und Toolbuttons entsprechend dargestellt werden. So wird die Aktion auch ausgeführt, wenn sie über irgendein angeschlossenes Element ausgelöst wird.

Damit ergibt sich allerdings auch eine geringfügig andere Vorgehensweise beim Applikationsentwurf, denn an erster Stelle steht nun das Anlegen der Aktionslisten. Anschließend werden die Menüs und Symbolleisten erstellt. Es muß dann nur noch der entsprechende Eintrag angelegt und der zugehörigen Aktion zugewiesen werden. Die Einstellung der Eigenschaften und Auswertung der Ereignisse erledigt das Aktionsobjekt dann selbständig.

Mit der neuen Delphi-Version kommen sogar noch einige neue Möglichkeiten für einen noch effizienteren Einsatz ins Spiel. Zunächst gibt es eine ganze Reihe neuer Standardaktionen. Diese Aktionen müssen nur noch in die Anwendung aufgenommen werden und haben beispielsweise bereits die Fähigkeit, markierten Text in die Zwischenablage zu übernehmen. Dazu muß nicht eine Zeile Code geschrieben werden. Zudem wird ein Aktionsmanager als Komponente angeboten, mit dem der Anwender seine Menüs und Werkzeugleisten selbst konfigurieren kann. Selbst das Speichern und Laden der benutzerspezifischen Einstellungen wird von dieser Komponente automatisch erledigt. Zum besseren Verständnis wird zunächst die herkömmliche Arbeitsweise mit Aktionen beschrieben. Anschließend gehe ich näher auf die neuen Möglichkeiten mit dem Aktionsmanager ein.



Abgesehen davon ist die gezeigte Vorgehensweise die einzige Möglichkeit bei der Entwicklung von CLX-Anwendungen, da dort der Aktionsmanager nicht verfügbar ist.

Erstellen von Aktionslisten

Aktionslisteneditor

Mit der Komponente **TActionList** wird eine Aktionsliste in das Formular eingebunden. Die für die Aktionen verwendeten Bitmaps sind in der für die Eigenschaft **ImageList** angegebenen Bilderliste (**TImageList**) enthalten. Die Aktionsliste wird über den **Aktionslisteneditor** (siehe Abbildung 9.1) bearbeitet, der über das lokale Menü der Komponente geöffnet wird.

Die Aktionsliste beinhaltet also mehrere Aktionen. Die einzelnen Aktionen sind Komponenten, die von der Basisklasse **TContainedAction** abgeleitet sind. Die Aktionen werden in Kategorien eingeteilt, welche in der linken Liste im Aktionseditor dargestellt werden. In der rechten Liste sehen Sie die einzelnen Aktionen. Wird eine Aktion in der Liste selektiert,

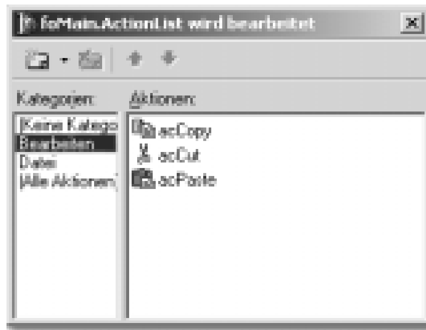


Abbildung 9.1 Aktionslisteneditor

kann diese im Objektinspektor bearbeitet werden. Über den Button **Neue Aktion** wird eine neue Aktion angelegt. Es stehen bereits sehr viele Standardaktionen zur Verfügung, es können aber auch neu definierte Aktionen angelegt werden.

Anlegen von Aktionen

Die Standardaktionen sind vordefinierte Aktionen, bei denen die Einstellungen der Aktion bereits festliegen. Dazu besitzen diese Aktionen einige erweiterte Möglichkeiten, die noch einmal zusätzliche Arbeit abnehmen, was die Verwaltung der Aktionen betrifft.

Wird über den Button **Neue Aktion** der Eintrag **Neue Standardaktion** ausgewählt, öffnet sich, wie in Abbildung 9.2 gezeigt, der Auswahldialog für Standardaktionen.

Abbildung 9.2 Standardaktionen

Hier können die gewünschten Aktionen ausgewählt werden (auch eine Mehrfachauswahl ist möglich). Mit **OK** werden die Aktionen in die Aktionsliste aufgenommen. Sofern für eine Standardaktion bereits ein Symbol existiert, wird dieses automatisch in die angegebene Bilderliste aufgenommen. Andere Aktionen werden über **Neue Aktion** angelegt.

Eigenschaften von Aktionen

Aktionen werden über den Objektinspektor bearbeitet. Mit der Eigenschaft **Category** wird die Aktion in eine Kategorie eingeordnet. Der Objektinspektor stellt eine Liste der bereits vorhandenen Kategorien zur Verfügung. Soll die Aktion einer neuen Kategorie zugeteilt werden, wird hier einfach der gewünschte Name der Kategorie direkt eingegeben.

Die Eigenschaft **Caption** legt die Bezeichnung der Aktion fest. Später wird diese Beschriftung für die Bezeichnung von Menüeinträgen und als Buttonbeschriftung verwendet. Mit **ImageIndex** wird der Aktion ein Bitmap der Bilderliste zugeordnet. Dieses Bitmap wird schließlich im Menü und für die Toolbuttons angezeigt. Der Wert –1 bedeutet, daß für den Eintrag kein Bitmap dargestellt wird. Die Eigenschaft **HelpContext** gibt den Index der kontextsensitiven Hilfe an. Mit **Hint** wird der Hinweistext für den Eintrag angegeben. Die Eigenschaft **ShortCut** bestimmt den Shortcut für die Aktion, und der Shortcut wird im Menü dargestellt.

Letztlich gibt es Eigenschaften, die den Status der Aktion kennzeichnen. Mit **Visible** wird festgelegt, ob die angeschlossenen Elemente (Menüpunkte, Toolbuttons) sichtbar sind. **Enabled** legt fest, ob der Eintrag ausgewählt werden kann oder grau dargestellt wird. **Checked** wird verwendet, wenn die Aktion eine Option darstellt, die an- und abgewählt werden kann. Ist die Eigenschaft gesetzt, werden die zugehörigen Menüpunkte mit einem Häkchen dargestellt, und die Buttons erscheinen gedrückt.

Zuweisen der Aktion zu Elementen

Die Eigenschaft **Action** ist bereits in der Klasse **TControl** öffentlich deklariert. Veröffentlicht wird die Eigenschaft in der Hauptsache von den Buttonkomponenten. Genauso verfügen natürlich Menüeinträge (**TMenuItem**) über diese Eigenschaft. Mit ihr werden diese Elemente mit einer Aktion verknüpft. Beim Zuweisen einer Aktion übernimmt die Komponente die Eigenschaften der Aktion. Im Falle eines Menüeintrags werden also beispielsweise die Eigenschaften **Caption**, **ImageIndex**, **HelpContext** und **ShortCut** übernommen.

Auslösen von Aktionen

Das Ereignis **OnExceute** wird ausgelöst, wenn die Aktion ausgeführt werden soll. Das ist zum Beispiel dann der Fall, wenn der Anwender auf einen der zugehörigen Buttons oder Menüpunkte klickt. Die Aktion ent-

spricht damit den Ereignissen **OnClick** der Steuerelemente. Es muß daher nicht mehr das **OnClick**-Ereignis eines jeden Elements ausgewertet werden, da dieses ohne weiteres Zutun zuverlässig an die Aktion weitergereicht wird.

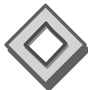
Die Standardaktionen besitzen bereits Funktionalität, um auf das Auslösen eines Ereignisses zu reagieren. Wird zum Beispiel die Standardaktion **Kopieren** ausgelöst, wird der Inhalt des aktuellen Eingabeelements automatisch in die Zwischenablage übernommen.

Status von Aktionen

Wie zuvor bereits angedeutet, besitzen die Aktionen die Eigenschaften **Visible**, **Enabled** und **Checked**, mit denen der Status der Aktion dargestellt wird. Diese Eigenschaften können jederzeit gesetzt werden. Dabei ist sichergestellt, daß der Statuswechsel an alle zugeordneten Elemente weitergereicht wird.

Eine Möglichkeit, den Status einer Aktion aufgrund des aktuellen Programmstatus zu setzen, ist das Ereignis **OnUpdate**. Dieses Ereignis wird ausgelöst, wenn die Methode **Update** der Aktion aufgerufen wird oder die Applikation sich im Leerlauf befindet. Den Status auf diese Art zu setzen ist immer dann sinnvoll, wenn man Statuswechsel nicht direkt über andere Ereignisse verfolgen kann. In diesem Fall bleibt dann tatsächlich nur die Möglichkeit, den Status gelegentlich zu überprüfen und die Aktion entsprechend anzupassen.

9.1.2 Ergänzung des Texteditors um eine Aktionsliste



Programmtechnische Umgestaltung

Der Texteditor beinhaltet bereits ein Menü, mit dem verschiedene Programmfunktionen ausgelöst werden. Dabei wurde die Entwicklung allerdings ein wenig voreilig begonnen, so daß eigentlich bereits etwas Arbeit umsonst gemacht wurde. Die zentrale Verwaltung der Funktionen soll von einer Aktionsliste durchgeführt werden. Deshalb werden alle Programmfunktionen des Texteditors in die Aktionsliste aufgenommen. So bleibt letztlich nur noch die Aufgabe, die Menüpunkte mit den Aktionen zu verbinden.

Die Ereignisbehandlungen für die Bearbeiten-Funktionen können bedenkenlos gelöscht werden, da hier mit den Aktionslisten eine elegantere Gestaltung möglich ist. Ebenso können die Ereignisbehandlungen der Menüpunkte **Neu**, **Öffnen** und **Beenden** gelöscht werden.

Das zeigt nun bereits, daß es durchaus vernünftig ist, die genannte Reihenfolge bei der Anwendungsentwicklung einzuhalten, also zuerst die Programmaktionen zu definieren und dann die Menüs und Symbolleisten zu erstellen. Wäre das bei der Entwicklung des Texteditors von vornherein beachtet worden, wären diese Änderungen nicht notwendig gewesen.

Erstellung der Aktionsliste

Zunächst wird die Komponente **TActionList** in das Formular **foMain** eingebunden. Außerdem wird eine Bilderliste eingefügt und der Eigenschaft **Images** der Aktionsliste zugewiesen. Über den Aktionslisteneditor werden dann die Aktionen definiert. Die Kategorien werden entsprechend den Haupteinträgen im Hauptmenü angelegt.

Standardaktionen

Im Beispiel können alle Standardaktionen zum Bearbeiten des Textes verwendet werden. Da es sich bei der Anwendung um eine MDI-Anwendung handelt, sind auch die Fenster-Aktionen dienlich (lediglich die Aktion **Anordnen** wird nicht benötigt). Zugunsten einer besseren Lesbarkeit des Quelltextes sollten den Aktionsobjekten sinnvolle Namen geben werden. Die Abbildung 9.3 zeigt die für den Texteditor vergebenen Namen.

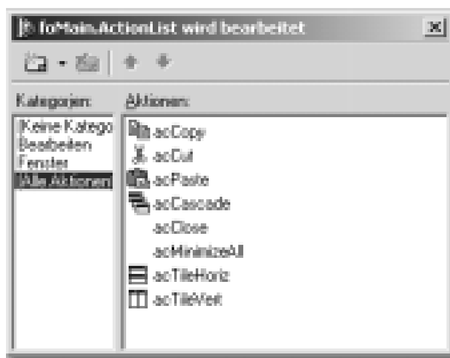


Abbildung 9.3 Standardaktionen des Texteditors

Des weiteren werden Aktionen für die Dateioperationen benötigt. Mit Delphi 6 sind nun auch hierfür bereits Standardaktionen vorbereitet. Wählen Sie also für die Funktion **Datei öffnen** die Standardaktion **TFileOpen**. Für das Druckersetup steht die Aktion **TFilePrintSetup** zur Verfügung. Auch wenn der Quellcode zum Beenden der Anwendung nicht sehr aufwendig ist, könnte man die Aktion **TFileExit** zum Beenden der Anwendung einsetzen. Zum Speichern einer Datei stände zwar eine Standardaktion **TFileSave** zur Verfügung. In diesem Fall finde ich es allerdings

praktischer, bei der herkömmlichen Vorgehensweise zu bleiben und die Aktionen selbst zu implementieren.

Sonstige Aktionen

Die Funktionen **Datei Neu, Speichern** und **Drucken** sind noch nicht durch Standardaktionen vorbereitet. Hierfür werden daher neue Aktionen angelegt und in die Kategorie **Datei** eingeordnet (siehe Abbildung 9.4). Dazu wurden zuvor bereits einige Standardbitmaps in die Bilderliste eingefügt. Für die einzelnen Einträge werden dann die Eigenschaften **Caption**, **ImageIndex**, **Hint** und **ShortCut** gesetzt. Auf eine detaillierte Aufzählung der Werte wird hier verzichtet.

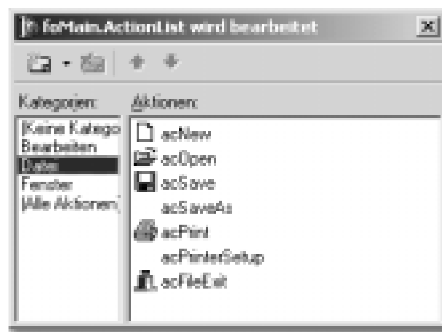


Abbildung 9.4 Dateiaktionen des Texteditors

Nachdem die erforderlichen Aktionen angelegt sind, können die Menüs erstellt bzw. um die fehlenden Einträge ergänzt werden. Dabei werden die einzelnen Menüpunkte mit den entsprechenden Aktionen verknüpft. Das erfolgt einfach durch die Zuweisung der Aktion über die Eigenschaft **Action** der Einträge. Damit auch die Bitmaps im Menü dargestellt werden, ist dem Menü noch die Bilderliste über die Eigenschaft **Images** zuzuweisen. Zunächst wird das Menü des Hauptformulars mit den zugehörigen Aktionen verknüpft. Dem Menüpunkt **Neu** wird also die Aktion **acNew** zugewiesen, dem Menüpunkt **Öffnen** die Aktion **acOpen**. Der Menüpunkt **Beenden** entspricht der Aktion **acFileExit**. Anschließend wird das Menü des Textformulars erstellt (vergessen Sie auch hier nicht, der Eigenschaft **Images** die Bilderliste des Hauptformulars **foMain.ImageList** zuzuweisen). Dieses Menü wird zunächst um das Fenstermenü erweitert. Dann werden die Eigenschaften **Action** aller dieser Menüpunkte mit den Aktionen des Hauptformulars verbunden. Nachdem das Hauptformular ohnehin schon unter **uses** des Formulars eingebunden ist, ist auch der Zugriff auf die Komponenten des Hauptformulars möglich. Der Objektinspektor stellt für die Eigenschaft eine Liste der verfügbaren Aktionen dar. Die Aktionen des Hauptformulars haben die Namen **foMain.ac**.XXX.

Die Anwendung kann jetzt gestartet werden. Die Menüs stellen sich nun entsprechend den Aktionen dar. Eine interessante Eigenschaft der Standardaktionen ist es, daß ganz automatisch und ohne eine einzige Zeile Programmcode sämtliche Bearbeiten- und Fensterfunktionen ihre Aufgabe erfüllen. Beachten Sie auch, daß die Bearbeiten-Menüpunkte entsprechend der aktuellen Selektion im Textfeld automatisch aktiviert und deaktiviert werden. Genauso beendet die Aktion **acFileExit** die Anwendung und **acPrinterSetup** zeigt den Dialog für die Druckereinstellungen.

Einige Aktionen haben noch keine Funktion. Das sind einerseits die Aktionen **acNew** oder **acPrint**, deren Funktion nicht in Form von Standardaktionen vorbereitet ist. Dann gibt es noch Aktionen wie zum Beispiel **acOpen** zum Öffnen einer Textdatei. Hier ist ein Teil der Funktionalität, nämlich das Anzeigen des Dateidialogs bereits in der Aktion implementiert. Den Code, der schließlich die Datei öffnet, müssen wir noch ergänzen.

Zunächst wird die Aktion **acNew** zum Anlegen einer neuen Datei implementiert. In diesem Fall genügt es vollkommen, das Ereignis **OnExecute** der Aktion auszufüllen. Dieses Ereignis wird ausgelöst, wenn die Aktion ausgeführt wird. Nach der Zuweisung der Ereignisbehandlung erscheint der Menüpunkt übrigens nicht mehr deaktiviert im Menü.

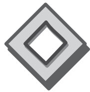
Die Aktion ruft letztlich nur die Funktion **CreateEditor** auf, um ein neues Fenster zu erzeugen:

```
procedure TfoMain.acNewExecute(Sender: TObject);
begin
  CreateEditor;
end;
```

Etwas interessanter ist die Aktion zum Öffnen einer Datei **acOpen**. Die Aktion ist bereits so vorbereitet, um den Datei-Öffnen-Dialog anzuzeigen. Um das Erscheinen des Dialogs zu definieren finden Sie im Objektinspektor die Eigenschaft **Dialog** für diese Aktion. Das ist nicht viel mehr, als eine in das Aktionsobjekt eingebettete TOpenDialog-Komponente. Das Ereignis **OnAccept** wird ausgelöst, wenn der Anwender den Dialog mit **OK** quittiert. Daher muß in der Ereignisbehandlung nur noch der Code zum Öffnen der Datei implementiert werden:

```
procedure TfoMain.acOpenAccept(Sender: TObject);
begin
  OpenTextfile(acOpen.Dialog.FileName);
end;
```

Die Aktionen zum Speichern werden nun nicht mehr im Textformular, sondern im MDI-Rahmen implementiert, da dieser die Aktionsliste enthält. Nachdem sich die Aktionen **acSave** und **acSaveAs** im Hauptformular befinden, muß in den Ereignisbehandlungen das Formular ermittelt werden, auf das sich die Funktionen beziehen. Das ist das aktive Formular innerhalb der MDI-Anwendung, welches mit **GetActiveMDIChild** ermittelt wird. Diese Funktion liefert als Ergebnis die Referenz auf das momentan geöffnete Formular. Ist gerade kein MDI-Formular geöffnet, wird der Wert **nil** zurückgegeben:



```
  public
    { Public declarations }
    function ActTxtForm: TfoText;
  end;
...
function TfoMain.ActTxtForm: TfoText;
begin
  if ActiveMDIChild<>nil then Result:=nil
  else Result:=TfoText(ActiveMDIChild);
end;
```

Die Ereignisbehandlungen **OnExecute** der Aktionen **acSave** und **acSaveAs** fallen dann recht einfach aus, denn sie müssen lediglich die Funktionen **Save** bzw. **SaveAs** des aktuellen Textfensters aufrufen:

```
procedure TfoMain.acSaveExecute(Sender: TObject);
begin
  ActTxtForm.Save;
end;
procedure TfoMain.acSaveAsExecute(Sender: TObject);
begin
  ActTxtForm.SaveAs;
end;
```

Genaugenommen müßte zur Vermeidung von Programmfehlern das Funktionsergebnis von ActTxtForm auf nil geprüft werden. Darauf wurde hier verzichtet, weil entweder die Aktion in einem Child-Formular ausgelöst wird (welches dann zweifelsohne vorhanden ist) oder die entsprechenden Bedienelemente des Hauptformulars deaktiviert sind, wenn kein Child-Formular existiert.

Zusammenfassung

In erster Linie wurde durch die Einführung der Aktionsliste ein klarerer Aufbau der Anwendung erreicht. Zusätzlich konnten ohne größeren Aufwand die Fensterfunktionen eingebaut werden. Außerdem werden die Menüpunkte des Bearbeiten-Menüs automatisch aktiviert und deaktiviert, abhängig davon, ob gerade ein Text in dem Textfeld ausgewählt ist. Der besondere Vorteil, der durch die Umgestaltung der Anwendung erzielt wurde, zeigt sich besonders bei der Einführung von Popup-Menüs und Symbolleisten. Dann ist nur noch ein minimaler Aufwand notwendig, um die neuen Menüeinträge und Toolbuttons mit den ohnehin schon vorhandenen Aktionen zu verknüpfen.

9.2 Symbolleiste

9.2.1 TCoolBar

Vererbung	TObject→ TPersistent→ TComponent→ TControl→ TWinControl→ TToolWindow→ TCoolBar
Palettenseite	Win32
Deklariert in Unit	ComCtrls

Diese Komponente ist in der CLX nicht enthalten. Bei CLX-Anwendungen müssen Sie sich behelfen, indem Sie die einzelnen Werkzeugleisten direkt im Formular plazieren. Der Anwender kann die Anordnung der einzelnen Bereiche dann allerdings nicht frei einteilen.

Ein sehr wichtiges Gestaltungselement für Anwendungen ist die Symbolleiste. Im letzten Kapitel wurde bereits die Erstellung einfacher Symbolleisten mit der Komponente **TToolBar** vorgestellt. Delphi bietet allerdings mittlerweile wesentlich flexiblere Möglichkeiten zur Erstellung von Symbolleisten. Symbolleisten setzen sich meist aus mehreren einzelnen Bereichen zusammen, die innerhalb der Leiste beliebig angeordnet und in der Größe geändert werden können. Dieser Aufgabe wird die Komponente **TCoolBar** gerecht, die mehrere Bereiche beinhaltet, die wie gewünscht verschoben und skaliert werden können. Diese Bereiche werden in Form von **TCoolBand**-Objekten verwaltet. Diese **Bänder** enthalten dann die eigentlichen Kontrollelemente der Symbolleiste. In den meisten Fällen wird man in die **CoolBar**-Komponente einfach mehrere Symbolleisten vom Typ **TToolBar** einfügen. Die ToolBars verwalten dann die darin ent-

haltenen Eingabeelemente, während die CoolBar-Komponente nur mit der Verwaltung der Bereiche, also der einzelnen ToolBars beschäftigt ist.

Arbeiten mit CoolBars

Einfügen von Elementen in die Symbolleiste

Sobald eine von **TWinControl** abgeleitete Komponente in die CoolBar-Komponente plaziert wird, wird automatisch ein neuer Bereich angelegt. Der Bereich besitzt auf der linken Seite einen Griff, mit dem er positioniert und in seiner Größe verändert werden kann. Mit den in der Symbolleiste enthaltenen Elementen kann weiter wie gewohnt gearbeitet werden. Bereiche in Form von CoolBand-Objekten können auch mit dem Abschnittseditor (lokales Menü des CoolBars) bearbeitet werden. Um die Eigenschaften eines Bereichs im Objektinspektor zu editieren, wird er in diesem Editor ausgewählt.

Darstellung

Zur Darstellung der CoolBar-Komponente existiert eine Reihe von Eigenschaften, die bereits hinlänglich bekannt sind, um beispielsweise die Darstellung der Rahmenlinien festzulegen.

Jeder Abschnitt kann eine Überschrift und ein Bitmap anzeigen. Dazu werden für das CoolBand-Objekt die Eigenschaften **Text** und **ImageIndex** gesetzt. **ImageIndex** gibt den Index des Bilds aus der Bilderliste an, die für die CoolBar-Komponente mit der Eigenschaft **Images** festgelegt ist. Die Eigenschaft **ShowText** des CoolBars bestimmt, ob die Überschriften der einzelnen Bereiche angezeigt werden.

Die Eigenschaft **FixedOrder** legt fest, ob der Anwender die Position der Bereiche im CoolBar-Bereich verändern kann. Analog dazu gibt **FixedSize** an, ob die Bereiche in der Größe verändert werden können. Mit **AutoSize** wird angegeben, ob der CoolBar-Bereich bei der Positionierung von Bereichen in der Größe angepaßt wird.

Einfügen von Werkzeugleisten

Üblicherweise wird man die einzelnen Schalter in Werkzeugleisten plazieren, die ihrerseits in einem CoolBand-Objekt der CoolBar-Komponente liegen. Dazu wird einfach der **ToolBar** aus der Komponentenpalette in den Zeichenbereich des CoolBars eingefügt. Dieser erstellt dann automatisch ein neues **CoolBand**, das den **ToolBar** enthält.

Da der Bereich im Normalfall ohnehin mit einer Rahmenlinie umgeben ist, ergibt sich eine optisch ansprechendere Gestaltung, wenn die Rahmenlinien des enthaltenen ToolBars entfernt werden (Eigenschaft **EdgeBorders**).

9.2.2 Erweiterung des Texteditors um eine Symbolleiste

Symbolleisten in MDI-Anwendungen

Die Erstellung von Menüs in MDI-Anwendungen gestaltet sich sehr einfach, da die Anwendung im Hauptfenster automatisch das Menü des momentan aktiven MDI-Child-Fensters einblendet. Damit kann jedes Child-Fenster ein eigenes Menü enthalten, das an den aktuellen Zustand des Fensters angepaßt ist. Leider funktioniert dieser Mechanismus nicht für Symbolleisten. Praktisch wäre es zwar, wenn jedes Fenster seine eigene Symbolleiste hätte, die dann im Hauptfenster eingeblendet wird. Damit wäre ein sauberer modularer Aufbau der Anwendung möglich. Prinzipiell kann man ein derartiges Verhalten selbst nachbilden, indem man zum Beispiel beim Aktivieren eines Child-Fensters die Symbolleiste des Hauptfensters verbirgt, statt dessen die Symbolleiste des Child-Fensters einblendet (das wäre durch das Setzen des Parents der Symbolleiste durchaus möglich). Eine wirklich saubere Lösung wird man hier allerdings nur sehr schwer erreichen, wenn die Symbolleisten konfigurierbar sind und damit auch die Höhe ändern können, sofern der Anwender die Bereiche anders anordnet. Eine saubere Umschaltung ist damit kaum möglich.

Für eine bessere Lösung halte ich es daher, die Symbolleiste nur im Hauptformular einzusetzen und diese aufgrund des aktiven Child-Fensters anzupassen. Weil ohnehin mit Aktionslisten gearbeitet wird, bereitet diese Lösung kaum Probleme, da das Update der einzelnen Aktionen sehr einfach durchgeführt werden kann.

Einfügen der Symbolleisten in das Hauptfenster des Texteditors

Zunächst wird eine **CoolBar**-Symbolleiste in das Hauptfenster **foMain** eingefügt. In diese werden zwei Toolbars aufgenommen (Komponente **ToolBar**). Es sollten nun zwei Abschnitte in der Symbolleiste dargestellt werden, die jeweils eine Symbolleiste beinhalten. Der erste ToolBar wird allgemeine Funktionen beinhalten (insbesondere Datei- und Bearbeiten-Funktionen), während der zweite die Funktionen zur Formatierung des Textes enthält.

Wie oben angedeutet, ergibt sich optisch eine ansprechende Gestaltung, wenn die Rahmenlinien der ToolBars entfernt werden (Eigenschaft **EdgeBorders**). Weiter sollten die in den Leisten enthaltenen Elemente flach dargestellt werden, was durch Setzen der Eigenschaft **Flat** der beiden ToolBars erreicht wird. Die Größe der Symbolleiste muß sich an die aktu-

elle Positionierung der einzelnen Symbolleisten anpassen. Dazu wird die Eigenschaft **AutoSize** der CoolBar-Komponente gesetzt. Die Buttons der Symbolleiste erscheinen, sofern die Höhenangaben der Elemente auf ihren Default-Werten belassen werden, immer ganz am oberen Rand der Symbolleiste.

Optisch ergibt sich eine bessere Darstellung, wenn für den ToolBar eine Höhe von 26 (Default ist 25) und der Rahmen mit der Eigenschaft **BorderWidth** auf den Wert 1 eingestellt wird. Damit die Hinweistexte angezeigt werden, sollte die Eigenschaft **ShowHint** der Symbolleisten gesetzt werden.

Symbolleiste konfigurieren

Die erste Symbolleiste soll die grundlegenden Datei- und Bearbeiten-Funktionen beinhalten. Die zugehörigen Bitmaps sind in der Bilderliste enthalten, die dem ToolBar mit der Eigenschaft **Images** zugewiesen werden. Über das lokale Menü des ToolBars können neue Schalter und Trenner eingefügt werden. Für das Beispiel wurden drei Schalter für die Menüpunkte **Neu**, **Öffnen** und **Speichern** eingefügt, davon durch eine Trennlinie abgeteilt der Schalter **Drucken** und davon abgetrennt die drei Schalter für die Funktionen **Ausschneiden**, **Kopieren** und **Einfügen**. In Abbildung 9.5 ist die Anordnung der Buttons dargestellt. Für die Buttons muß lediglich die entsprechende Aktion an die Eigenschaft **Action** zugewiesen werden. Anschließend ist die Symbolleiste bereits funktionsfähig.

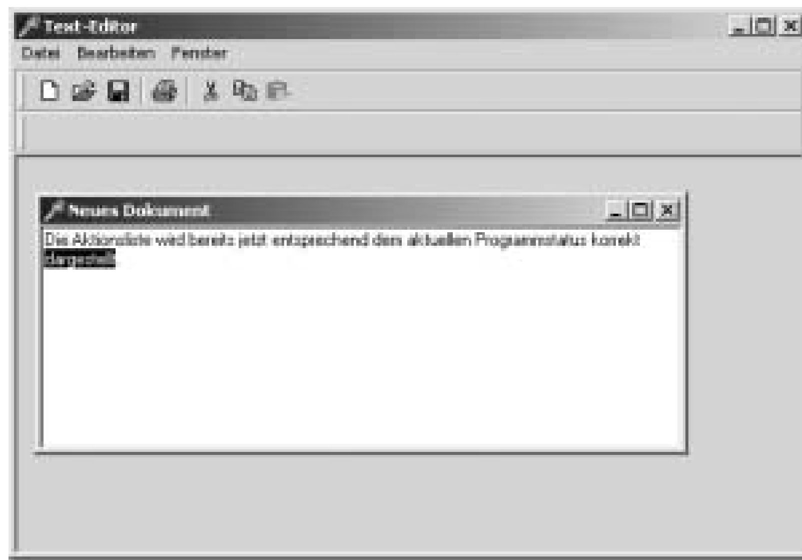


Abbildung 9.5 Der erste Teil der Symbolleiste

Update der Symbolleiste

Ein kleines Problem besteht nun noch mit dem Button **Speichern**. Er ist immer aktiv, kann also immer gedrückt werden, selbst wenn kein Dokument geöffnet ist. In diesem Fall wird das Auslösen der Aktion mit Sicherheit zu einem Programmfehler führen. Ob gerade ein Child-Fenster geöffnet ist, kann einfach über die Eigenschaft **ActiveMDIChild** des Hauptformulars ermittelt werden. Diese gibt den Wert **nil** zurück, sofern kein Fenster geöffnet ist. Konkret bedeutet das, daß die Aktion inaktiv sein muß, wenn die Funktion den Wert **nil** liefert.

OnUpdate

Jede Aktion besitzt das Ereignis **OnUpdate**. Dieses wird immer dann ausgelöst, wenn die Symbolleiste neu dargestellt werden muß bzw. wenn sich die Anwendung im Leerlauf befindet:

```
procedure TfoMain.acSaveUpdate(Sender: TObject);
begin
  (Sender as TAction).Enabled:=ActiveMDIChild<>nil;
end;
```

Diese Ereignisbehandlung wurde allgemein formuliert, so daß sie auch für das Update-Ereignis weiterer Aktionen verwendet werden kann. Es wurde daher nicht direkt Bezug auf die Aktion **acSave** genommen, sondern der Parameter **Sender** verwendet, um den Auslöser des Ereignisses zu bestimmen.

Funktionen zum Formatieren von Text

Der Texteditor soll um einige Funktionen zum Bearbeiten von Text erweitert werden. Sie finden bei den Standardaktionen bereits die wichtigsten Formatierungsfunktionen fertig vorbereitet. Es würde also genügen, diese Aktionen in die Aktionsliste aufzunehmen. Allerdings stellen sich genau diesen Programmfunktionen einige sehr interessante Anforderungen, die dem Verständnis von Aktionen sehr zugute kommen. Ich habe daher an dieser Stelle den schwierigeren Weg gewählt und die Formatierungsfunktionen selbst implementiert.

Die RichEdit-Komponente besitzt die beiden Eigenschaften **SelAttributes** und **Paragraph**, um das Zeichen- und Absatzformat zu setzen. Über diese Eigenschaften kann auch das Format an der aktuellen Cursorposition bestimmt werden. Damit wird sich der Quelltext darauf beschränken, lediglich einige Eigenschaftswerte zu schreiben. Eine Auswahl der häufigsten Formatierungsfunktionen wird über die Symbolleiste zugänglich gemacht. Auch hier muß der Status der Elemente in der Symbolleiste ent-

sprechend dem momentan aktiven Textfenster dargestellt werden. Damit bietet es sich auch in diesem Fall wieder an, mit dem Ereignis **OnUpdate** der Aktionslisten zu arbeiten.

Vorbereiten der Aktionsliste

Im ersten Schritt wird die Aktionsliste für die Formatierungsfunktionen erstellt. Um eine etwas bessere Übersichtlichkeit zu erreichen, werden im Hauptformular eine neue Aktionsliste (Name: **alFormat**) und eine neue Bilderliste (Name: **ilFormat**) erstellt. In die Bilderliste werden die Icons für die verschiedenen Formatfunktionen eingefügt. Vergessen Sie auch nicht, die Aktionsliste über die Eigenschaft **Images** mit der Bilderliste zu verbinden.

In die Aktionsliste werden dann die Formatfunktionen mit den entsprechenden Bitmaps aufgenommen. Das sind die Aktionen **acBold** (Fett), **acItalic** (Kursiv), **acUnderline** (Unterstrichen), **acAlignleft** (Linksbündig), **acAlignCenter** (Zentriert), **acAlignRight** (Rechtsbündig) und **acEnumerate** (Aufzählungszeichen).

Erstellen der Symbolleiste

Zu den besagten Formatierungsfunktionen sollen in der Symbolleiste noch zwei Auswahlelemente für Schriftart und Schriftgrad eingefügt werden. Die Abbildung 9.6 zeigt, daß man sich bei der Erstellung der Symbolleiste nicht auf ToolButtons beschränken muß, sondern daß auch andere Dialogelemente eingebunden werden können.

Abbildung 9.6 Symbolleiste für die Formatierungsfunktionen

Die beiden ComboBox-Komponenten **cbFontName** und **cbFontSize** erhalten ebenfalls einen Hinweistext (nicht vergessen, die Eigenschaft **ShowHint** zu setzen!). Die Liste für die Schriftgröße wird über den Objektinspektor erstellt. Es werden einfach einige Zahlenwerte in sinnvoller Schrittweite eingefügt. Die Liste mit den Schriftarten kann zur Entwicklungszeit noch nicht festgelegt werden, da diese ja von den auf dem jeweiligen Rechner installierten Schriftarten abhängt. Das globale Screen-Objekt bietet zur Laufzeit die Liste der verfügbaren Schriften. Daher wird im OnCreate-Ereignis des Formulars der Inhalt der Schriftauswahl gesetzt:

```
procedure TfoMain.FormCreate(Sender: TObject);
begin
  cbFontName.Items:=Screen.Fonts;
end;
```

Zum Ausführen der Aktionen müssen wieder die **OnExecute**-Ereignisse ausgewertet werden. Da sich die Aktionen ebenfalls im Hauptformular befinden, muß auch hier zunächst das aktive Textfenster ermittelt werden. Die dazu notwendige Funktion **ActTxtForm** steht bereits zu Verfügung. Es ergeben sich nun zwei grundsätzliche Verhaltensweisen bei den Buttons für die Formatierung. Bei den Zeichenformaten **Fett**, **Kursiv** und **Unterstrichen** können die Buttons umgeschaltet werden. Es können mehrere oder keines dieser Formate ausgewählt sein. Damit wird beim Auslösen der Aktion die Eigenschaft **Checked** einfach umgeschaltet. Die Eigenschaft **Checked** von Aktionen entspricht dem Status **gedrückt/nicht gedrückt** von Buttons. Ist die Eigenschaft gesetzt, wird das entsprechende Zeichenformat gesetzt, andernfalls wird es zurückgesetzt:

Ausführen der Aktionen

```
procedure TfoMain.acBoldExecute(Sender: TObject);
begin
  with ActTxtForm.RichEdit.SelAttributes do begin
    acBold.Checked:=not acBold.Checked;
    if acBold.Checked then Style:=Style+[fsBold]
    else Style:=Style-[fsBold];
  end;
end;
```

So in etwa sehen auch die Ereignisbehandlungen für die Formate **Kursiv** und **Unterstrichen** aus. Die Aktion für das Absatzformat **Aufzählung** verhält sich in exakt derselben Weise.

Die Ereignisbehandlungen für die Aktionen zur Ausrichtung von Absätzen werden analog aufgebaut. Hier kann immer nur eine Option ausgewählt sein. Man kann es sich hier sehr einfach machen, indem man sich darauf beschränkt, beim Auslösen einer dieser Aktionen das entsprechende Absatzformat zu setzen. Der Status der anderen Aktionen wird beim nächsten Update korrekt gesetzt. Theoretisch könnte das Setzen der Eigenschaft **Checked** komplett entfallen. Der Button reagiert so auf das Anklicken durch den Benutzer jedoch ein wenig schneller:

```
procedure TfoMain.acAlignLeftExecute(Sender: TObject);
begin
  with ActTxtForm.RichEdit.Paragraph do begin
    acAlignLeft.Checked:=true;
    Alignment:=taLeftJustify;
  end;
end;
```

Update der Aktionen

Beim Update der Aktionen müssen die Aktionen an den aktuellen Programmstatus angepaßt werden. Die Aktionen werden aktiviert, sobald ein Textfenster geöffnet ist. Die Buttons sind abhängig vom Format an der Cursorposition gedrückt oder nicht gedrückt darzustellen. Das wird in den Aktionsobjekten über die Eigenschaften **Enabled** und **Checked** bestimmt. Gesetzt werden diese Eigenschaften im **OnUpdate**-Ereignis der Aktionen. Die Auswahllisten für Schriftart und -grad können nicht direkt über Aktionen gesteuert werden, sollen aber trotzdem entsprechend den anderen Elementen aktiviert und deaktiviert werden. Daher kann das Setzen der Eigenschaften dieser Komponenten im Update-Ereignis einer beliebigen anderen Formataktion erfolgen, im vorliegenden Fall wurde dazu die **acBold**-Aktion verwendet:

```
procedure TfoMain.acBoldUpdate(Sender: TObject);
begin
  acBold.Enabled:=ActTxtForm<>nil;
  if acBold.Enabled then
    acBold.Checked:=
      fsBold in ActTxtForm.RichEdit.SelAttributes.Style;
  {Update der ComboBoxes}
  cbFontName.Enabled:=ActTxtForm<>nil;
  cbFontSize.Enabled:=cbFontName.Enabled;
  if cbFontName.Enabled then begin
    if not cbFontName.Focused then
      cbFontName.Text:=ActTxtForm.RichEdit.SelAttributes.Name;
    if not cbFontSize.Focused then
      cbFontSize.Text:=IntToStr(ActTxtForm.RichEdit.SelAttributes.Size);
  end;
end;
procedure TfoMain.acItalicUpdate(Sender: TObject);
begin
  acItalic.Enabled:=ActTxtForm<>nil;
  if acItalic.Enabled then
    acItalic.Checked:=
      fsItalic in ActTxtForm.RichEdit.SelAttributes.Style;
end;
```

Abhängig davon, ob gerade ein Textfenster geöffnet ist, wird auch hier die Eigenschaft **Enabled** der Aktionen gesetzt. Die Eigenschaft **Checked** wird aufgrund der Formatierung an der aktuellen Cursorposition des aktiven Textfensters gesetzt. In der **OnUpdate**-Methode der Aktion **acBold** werden auch die beiden Auswahlfelder aktualisiert.

Die **OnUpdate**-Ereignisse werden immer dann ausgelöst, wenn die Applikation im Leerlauf ist. Dieser Fall tritt relativ häufig ein, also auch, während der Anwender in einem der Auswahlfelder den Wert manuell eingeben will. Der Effekt ist, daß ohne entsprechende Vorkehrung seine Eingabe bei jedem **OnUpdate**-Ereignis mit dem Wert des aktuellen Formats überschrieben würde. Er könnte die gewünschte Eingabe also nie beenden und bestätigen. Daher wird die Aktualisierung der Felder ausgesetzt, solange sich der Eingabefokus auf dem jeweiligen Eingabefeld befindet. Entfernen Sie diese Überprüfung ruhig einmal aus dem Programm und beobachten Sie das Verhalten.

Die restlichen Ereignisbehandlungen sind entsprechend denen des **acItalic**-Objekts gestaltet. Die **OnUpdate**-Ereignisse werden, wie gesagt, relativ häufig ausgelöst. Trotzdem ergibt sich eine gewisse Verzögerung bei der Darstellung, wenn der Anwender die Cursorposition im Textfeld ändert. Um in diesem Fall eine etwas bessere Reaktionszeit zu erhalten, werden die Formataktionen mit der Methode **Update** beim Ändern der Selektion im Textfeld ausgelöst. Die RichEdit-Komponente besitzt hierzu das Ereignis **OnSelectionChange**. In dessen Ereignisbehandlung wird die **Update**-Methode der Formataktionen aufgerufen:

```
procedure TfoText.RichEditSelectionChange(Sender: TObject);
var
  i: Integer;
begin
  with foMain.alFormat do
    for i:=0 to ActionCount-1 do
      Actions[i].Update;
end;
```

Übernehmen der Eingabe in den Auswahllisten

Für die Formatierungsfunktionen muß jetzt noch die Übernahme der Eingabe der Auswahlfelder für Schriftart und -grad festgelegt werden. Diese besitzen das Ereignis **OnChange**, das ausgelöst wird, sobald sich der Wert ändert. Das funktioniert ausgezeichnet, solange sich der Anwender darauf beschränkt, den Wert aus der Liste auszuwählen. Bei der manuellen Eingabe wird das Ereignis allerdings bei jedem Tastendruck ausgelöst. Eleganter ist es, den Wert aufgrund des **OnClick**-Ereignisses zu überneh-

men. Das funktioniert problemlos, wenn der Anwender eine Auswahl aus der Liste vornimmt, selbst wenn die Auswahl mit der Tastatur erfolgt. Zusätzlich soll die Eingabe übernommen werden, wenn der Anwender die Enter-Taste drückt. Im **OnKeyPress**-Ereignis kann das recht einfach überprüft werden. Die Enter-Taste entspricht dem #13-Zeichen. Daraufhin wird einfach das **OnClick**-Ereignis der ComboBox ausgelöst. Die ComboBox quittiert das Drücken der Enter-Taste im Editiermodus noch mit einem Warnton im PC-Lautsprecher, da **Enter** eigentlich keine gültige Eingabe ist. Daher wird die Variable **Key** in diesem Fall auf #0 gesetzt, so daß auch diese Warnung unterbleibt. Das bringt allerdings den Nachteil mit sich, daß das Übernehmen der Auswahl bei aufgeklappter Liste mit Enter nicht mehr möglich ist. Sofern die Liste aufgeklappt ist, sollte die Eingabe von Enter daher nicht beeinflußt werden. Die Eigenschaft **DroppedDown** gibt darüber Auskunft:

```
procedure TfoMain.cbFontNameClick(Sender: TObject);
begin
  ActTxtForm.RichEdit.SelAttributes.Name:=cbFontName.Text;
end;
procedure TfoMain.cbFontNameKeyPress(Sender: TObject; var Key: Char);
begin
  if (Key=#13) and not cbFontName.DroppedDown then begin
    cbFontNameClick(Self);
    Key:=#0;
  end;
end;
```

9.2.3 Erstellen einer Favoritenliste

Viele Programme bieten eine Liste der zuletzt geöffneten Dateien zur Auswahl an. Diese Liste wird üblicherweise im Hauptmenü oder auf der Symbolleiste angezeigt. Beim Speichern und Öffnen von Dateien wird die Liste auf den neusten Stand gebracht. Die Liste wird beim Beenden der Anwendung gespeichert und beim nächsten Start wieder geladen, so daß die alten Einträge weiterhin zur Verfügung stehen. Für den Texteditor soll die Liste über den **Öffnen**-Button der Symbolleiste angezeigt werden. Dieser soll dazu einen DropDown-Button erhalten, mit dem die Liste angezeigt werden kann. Über diesen Button wird ein Popup-Menü eingeblendet, das die Favoritenliste beinhaltet.

Erzeugen und Aktualisieren der Favoritenliste

Die Favoritenliste wird in Form einer Stringliste (**TStringList**) im Programm verwaltet. Sie wird im Hauptformular deklariert, im **OnCreate**-Ereignis erzeugt und im **OnDestroy**-Ereignis des Formulars wieder aus dem Speicher entfernt:

```
type
  TfoMain = class(TForm)
  ...
  private
    { Private declarations }
    TopTenList: TStringList;
  ...
...
implementation
...
procedure TfoMain.FormCreate(Sender: TObject);
begin
  TopTenList:=TStringList.Create;
  ...
end;
procedure TfoMain.FormDestroy(Sender: TObject);
begin
  TopTenList.Free;
end;
```

Beim Öffnen einer Datei und beim Speichern unter einem neuen Namen soll die Liste aktualisiert werden. Beim Öffnen einer Datei wird der Dateiname an den Anfang der Liste gesetzt. Ist der Eintrag bereits in der Liste enthalten, wird er zuvor gelöscht, so daß keine doppelten Einträge entstehen und die zuletzt geöffnete Datei immer am Anfang der Liste erscheint. Beim Speichern einer Datei unter neuem Namen wird in der Liste gegebenenfalls der alte Dateiname durch den neuen ersetzt. Dazu wird im Hauptformular die Methode **SetTopTenList** erstellt und in der Öffnen- und Speichern-Methode eingefügt (diese befinden sich im Textformular **foText**). Die **SetTopTen**-Methode besitzt die beiden Parameter **OldFileName** und **NewFileName**, wobei für den **OldFileName** nur dann ein Wert angegeben wird, wenn die Datei zuvor unter einem anderen Namen gespeichert war. Die Implementierung für die Methoden zur Aktualisierung der Liste finden Sie im Quelltext auf der CD.

Update des Popup-Menüs

Interessant wird es beim Update des Popup-Menüs, über das die Einträge zugänglich gemacht werden. Das Popup-Menü wird in das Hauptformular eingefügt. Im Beispiel wurde dafür der Name **pmTopTen** vergeben. Weitere Eigenschaften müssen nicht verändert werden, da die Einträge zur Laufzeit dynamisch erstellt werden. Die dazu notwendigen Funktionen sind ebenfalls in der **SetTopTen**-Methode untergebracht. Der Fall, daß das Popup-Menü mehr Einträge als die Liste enthält, kann zwar theoretisch nicht vorkommen. Trotzdem wird sicherheitshalber in einer ersten Schleife die Überzahl der Einträge gelöscht. Die zweite Schleife fügt dem Popup-Menü neue Einträge hinzu, sofern die Favoritenliste größer ist als Einträge im Menü zur Verfügung stehen. Damit ist sichergestellt, daß das Menü immer genauso viele Einträge beinhaltet wie die Favoritenliste. Anschließend werden die Einträge der Liste in das Menü übernommen. Die Einträge werden mit einer laufenden Nummer versehen, die gleichzeitig als Tastenkürzel verwendet wird. Außerdem wird den Einträgen die Ereignisbehandlung **TopTenClick** zugewiesen, die ebenfalls für das Formular definiert ist. Dort wird mit der Funktion **OpenTextFile** die ausgewählte Datei geöffnet. Dabei wird davon ausgegangen, daß die Menüeinträge in gleicher Reihenfolge wie die Einträge der Liste vorliegen. Dabei wird der Eintrag der Liste verwendet, da die Beschriftung der Menüeinträge um die laufende Nummer erweitert ist, so daß der String zerlegt werden müßte:

```
procedure TfoMain.SetTopTen(OldFileName, NewFileName: String);
var
  i: Integer;
begin
  ...
  while pmTopTen.Items.Count>TopTenList.Count do
    pmTopTen.Items.Delete(pmTopTen.Items.Count);
  while pmTopTen.Items.Count<TopTenList.Count do
    pmTopTen.Items.Insert(0, TMenuItem.Create(Self));
  for i:=0 to TopTenList.Count-1 do begin
    pmTopTen.Items.Items[i].Caption:=
      '&'+IntToStr(i)+' '+TopTenList[i];
    pmTopTen.Items.Items[i].OnClick:=TopTenClick;
  end;
end;
```

```
procedure TfoMain.TopTenClick(Sender: TObject);
var
  i: Integer;
begin
  i:=TMenuItem(Sender).MenuIndex;
  OpenTextFile(TopTenList[i]);
end;
```

Anzeigen der Liste über die Symbolleiste

Bislang wird der **Öffnen**-Button in der Symbolleiste als normaler Button dargestellt. Um diesen um das DropDown-Feld zu erweitern, wird seine Eigenschaft Style auf **tbsDropDown** gesetzt, so daß er automatisch um den DropDown-Pfeil ergänzt wird. Wundern Sie sich nicht über die unpassende Größe des Buttons zur Entwicklungszeit. Zur Laufzeit wird die Größe automatisch korrekt gesetzt. Besonderer Vorteil dieser Darstellungsweise von Buttons ist, daß sie wie ein normaler Button funktionieren, wenn im linken Bereich mit der Maus geklickt wird. Es wird sich über den Button also weiterhin der Öffnen-Dialog anzeigen lassen. Wird auf den DropDown-Pfeil geklickt, wird das Popup-Menü angezeigt, das für den Button mit der Eigenschaft **DropDownMenu** angegeben ist. Im Beispiel wird dafür das Menü **pmTopTen** angegeben.

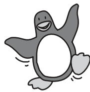
9.3 Anpassen von Menü und Werkzeugleisten

9.3.1 Überblick

Delphi 6 hat einige neue Komponenten, die es Ihnen vereinfachen sollen, dem Anwender anpassbare Menüs und Werkzeugleisten anzubieten. Das sind die Komponenten **TActionManager**, **TActionMainMenuBar**, **TActionToolbar** und **TCustomizeDlg** (Sie finden diese Komponenten auf der Palettenseite **Zusätzlich**). Mit diesen Komponenten können auf sehr einfachem Weg Oberflächen erstellt werden, die typisch sind für Anwendungen neueren Datums. Sie bieten die folgenden Möglichkeiten:

- Menüs, bei denen selten benötigte Menüpunkte verborgen werden
- Werkzeugleisten, die vom Anwender beliebig ein- und ausgeblendet werden können
- Werkzeugleisten, für die der Anwender die dargestellten Aktionen selbst definieren kann

Damit der Anwender wirklich Nutzen aus diesen Funktionen zieht, sollte die Erstellung der Aktionen, Menüs und Toolbars mit ein wenig Überlegung erfolgen. Sie haben einige Möglichkeiten, um die Gefahr zu verringern, daß der Anwender seine Werkzeuge allzu sehr »verkonfiguriert«. Außerdem sollte die Anwendung in ihrem Grundzustand mit einer möglichst sinnvollen Darstellung erscheinen. Die zugehörige Vorgehensweise möchte ich Ihnen anhand eines Beispiels schrittweise vorführen. Aus Gründen der Übersichtlichkeit möchte ich nicht auf unserem Beispiel, dem Texteditor, aufsetzen, sondern eine neue Anwendung erstellen.



Unter Kylix sind der Aktionsmanager und die zugehörigen Komponenten nicht verfügbar. Verwenden Sie in CLX-Anwendungen daher die Aktionslisten in der zuvor beschriebenen Form.

9.3.2 Beispiel

Schritt 1: CoolBar

Fügen Sie im ersten Schritt eine CoolBar-Komponente in das Formular ein. Der Einsatz der CoolBar-Komponente ermöglicht dem Anwender, die Anordnung der Werkzeugleisten selbst zu bestimmen, indem er die einzelnen Bereiche mit der Maus verschiebt. Die Anwendung soll ein Menü sowie zwei Werkzeugleisten erhalten. Fügen Sie daher eine TActionMainMenuBar und zwei TActionToolBar-Komponenten in die CoolBar ein. Damit sich die Größe der CoolBar-Komponente automatisch anpaßt, setzen Sie deren Eigenschaft **AutoSize** auf **true**.

Um das Verhalten der Aktionen zu testen, können Sie eine RichEdit-Komponente in das Formular ziehen. Sie können dann ohne weiteren Programmieraufwand auch die verschiedenen Standardaktionen zur Textformatierung verwenden.

Schritt 2: Aktionsmanager

Anschließend werden die möglichen Programmaktionen definiert. Fügen Sie dazu einen Aktionsmanager (**TActionManager**) und eine Bilderliste (**TImageList**) ein. Die Zuordnung der Bilderliste zum Aktionsmanager erfolgt mit der ActionManager-Eigenschaft **Images**. Ein Doppelklick auf den Aktionsmanager öffnet den entsprechenden Editor. Wie für die Aktionslisten lassen sich hier die einzelnen Programmaktionen erstellen. Es stehen dieselben Standardaktionen zur Verfügung (Abbildung 9.7).

Ich habe im Beispiel verschiedene Standardaktionen hinzugefügt. Eine bestimmte Aktion ist in diesem Zusammenhang besonders erwähnenswert: die Aktion **Tools/TCustomizeActionBars**, die den Aktionsmanager zur Laufzeit öffnet.

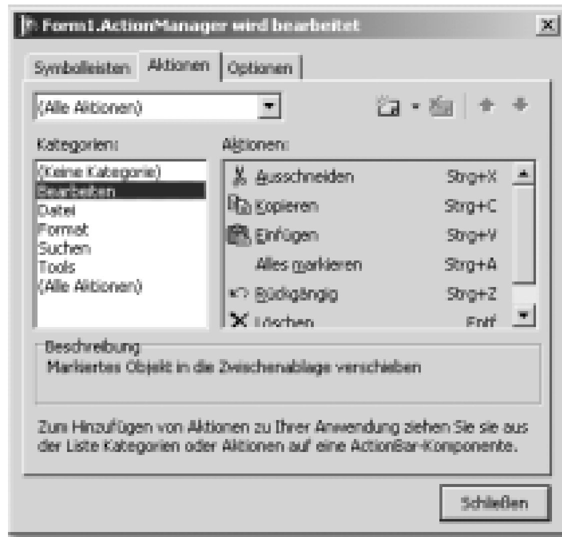


Abbildung 9.7 Editor des Aktionsmanagers

Schritt 3: Voreinstellung des Menüs

Zur Erstellung des Menüs müssen wir in diesem Fall nicht die herkömmliche Menükomponente bemühen. Vielmehr wird das Menü per Drag&Drop erstellt, indem Sie den Aktionsmanager öffnen und die einzelnen Menüpunkte auf die ActionMainMenuBar ziehen. Vorteilhaft ist es, die Aktionskategorien sorgfältig einzuteilen und zu benennen, da die Kategorieeinträge direkt auf die Menüzeile gezogen werden können. Der Name der Kategorie wird dann zum Hauptmenüpunkt, ihm werden die einzelnen Aktionen als Menüpunkte zugeteilt. Um Einträge wieder aus der Menüleiste zu entfernen, werden Sie per Drag&Drop aus der Menüleiste herausgezogen. Auch das Verschieben der Einträge erfolgt per Drap&Drop.

Schritt 4: Werkzeugleisten

Zur Erstellung der Werkzeugleisten fügen Sie einzelne Aktionen per Drag&Drop in die Werkzeugleisten ein. Gemäß der Voreinstellung werden die Buttons in der Werkzeugliste beschriftet. Im Aktionsmanager läßt sich diese Einstellung auf der Seite **Symbolleisten** anpassen. Für die Beschriftung gibt es die drei Möglichkeiten **Ohne**, **Selektiv** und **Alle**. Damit lassen sich die Beschriftungen generell ein- und ausschalten (Ohne/Alle). **Selektiv** bedeutet, daß sich die Anzeige der Einstellung nach der Einstellung der einzelnen Einträge richtet. Öffnen Sie den Editor der Eigenschaft **ActionBars** der Managerkomponente. Der Editor zeigt die Liste der »ActionBars«. Das ist eine Liste aller Menü- und Werkzeugleisten, denen bereits Aktionen zugewiesen wurden. Über deren Eigenschaft **Items** erreichen Sie die jeweiligen Einträge. Sie liegen in Form von

TActionClientItem-Komponenten vor, die die Verbindung zwischen Aktion und Werkzeugleiste darstellen. Hier läßt sich beispielsweise die Anzeige der Beschriftung mit der Eigenschaft **ShowCaption** festlegen. Sofern für die Werkzeugleiste die Beschriftungsart **selektiv** gewählt ist, richten sich die Beschriftungen nach der Einstellung der einzelnen Einträge.

Schritt 5: Speichern der Einstellungen

Damit der Aktionsmanager die benutzerdefinierten Einstellungen automatisch speichert, muß für die Eigenschaft **FileName** ein Dateiname angegeben werden.

Schritt 6: Weitere Einstellungen

Um zu verhindern, daß der Anwender eine Werkzeugleiste verbirgt, wird deren Eigenschaft **AllowHiding** auf **false** gesetzt. Für die Menüleiste ist das die Standardeinstellung. Standardmäßig werden selten benötigte Menüpunkte verborgen, häufig benötigte werden im Menü dargestellt. Leider funktioniert der Mechanismus manchmal erst nach dem wiederholten Start der Anwendung. Dieses Verhalten ist für mich allerdings nicht klar nachvollziehbar. Ich denke aber, daß Borland diesbezüglich mit einem Update-Pack Abhilfe schaffen wird. Um das Verbergen von Einträgen zu deaktivieren, ist die Eigenschaft **UsageCount** der jeweiligen ActionClient-Objekte auf den Wert –1 zu setzen. Ich habe festgestellt, daß sich in der derzeitigen Version auch diesbezüglich das gewünschte Verhalten erst dann einstellt, wenn nach der Änderung zunächst die Konfigurationsdatei mit alten Einstellungen gelöscht wird.

9.4 Baum- und Listenansichten

9.4.1 TTreeView

Übersicht

Vererbung	TObject→ TPersistent→ TComponent→ TControl→ TWinControl→ TCustomTreeView → TTreeView
Palettenseite	Win32
Deklariert in Unit	ComCtrls

Vererbung (CLX)	TObject→ TPersistent→ TComponent→ TControl→ TWidgetControl→ TFrameControl→ TCustomViewControl→ TCustomTreeView → TTreeView
Palettenseite	Common Controls
Deklariert in Unit	QComCtrls

Mit der Komponente **TTreeView** können auf einfache Art Baumansichten realisiert werden, wie sie zum Beispiel von der Verzeichnisansicht des Explorers bekannt sind. Die einzelnen Elemente in der Baumansicht sind vom Typ **TTreeNode**. Jedes Element in der Baumansicht wird mit einem Bitmap und einem Text dargestellt und kann weitere Untereinträge besitzen. Die Baumstruktur kann bereits zur Designzeit angelegt werden. Viele Aufgabenstellungen erfordern es jedoch, die Einträge zur Laufzeit dynamisch zu generieren.

Sofern Sie eine Komponente entwickeln wollen, die Daten in einer derartigen Struktur darstellt, bietet sich die Basisklasse **TCustomTreeView** an, da diese die gesamte Funktionalität der TreeView-Komponente besitzt, sie die Eigenschaften und Ereignisse jedoch noch nicht veröffentlicht.

Struktur der Einträge

Grundsätzlich kann die Baumansicht mehrere Wurzeln enthalten. Jede dieser Wurzeln ist, wie alle anderen Einträge auch, ein **TTreeNode**-Objekt. Jedes TreeNode-Objekt besitzt die Eigenschaft **Parent**, die angibt, welchem Knoten der Eintrag untergeordnet ist. Die Wurzeln der Struktur sind ihrerseits keinem Knoten untergeordnet. Ihre Eigenschaft **Parent** besitzt daher den Wert **nil**.

Jeder Knoten besitzt die Eigenschaft **HasChildren**, mit der festgestellt werden kann, ob der Knoten untergeordnete Elemente beinhaltet. Die Eigenschaft **Count** gibt an, wie viele Einträge zum Knoten gehören. Über die Eigenschaft **Item** können diese Einträge indiziert angesprochen werden.

Sämtliche in der Hierarchie enthaltenen Einträge werden mit dem Objekt **TTreeNodes** verwaltet, das durch die Eigenschaft **Items** der TreeNode-Komponente zugänglich gemacht ist. Das TreeNode-Objekt besitzt wiederum eine Eigenschaft **Item**, mit der die einzelnen Einträge indiziert angesprochen werden können. Die Eigenschaft **Count** gibt die Anzahl aller Einträge an.

Die TreeNode-Objekte besitzen die Eigenschaft **Owner**, mit der jenes TreeNodes-Objekt referenziert wird, dem die Einträge zugeordnet sind.

Darstellung der Einträge

Die TreeNode-Objekte besitzen eine Reihe von Eigenschaften, mit denen die Darstellung in der Baumansicht festgelegt wird. Der angezeigte Text wird mit der Eigenschaft **Text** angegeben.

Mit der Eigenschaft **ImageIndex** wird festgelegt, welches Bitmap aus der Bilderliste für den Knoten angezeigt wird. **SelectedIndex** gibt das Bitmap an, das dargestellt wird, wenn der Eintrag selektiert ist. Die Bilderliste wird mit der Eigenschaft **Images** der TreeView-Komponente angegeben.

Editieren von Einträgen

Die Einträge einer Listenansicht können direkt bearbeitet werden, sofern die Eigenschaft **ReadOnly** auf **false** gesetzt ist. Während dem Editieren eines Eintrags wird ein Eingabefeld über dem Element dargestellt. Um in den Eingabemodus zu gelangen, muß das Element angeklickt und dann sozusagen ein Rahmen aufgezogen werden. Das ist aber nicht unbedingt die gewohnte Vorgehensweise, um das Editieren zu starten. Erstens wird durch diese Mausaktion normalerweise eine Drag&Drop-Aktion gestartet. Zweitens ist eine reine Tastaturbedienung in diesem Fall nicht möglich.

Wechsel in den Editiermodus

Allgemein wird die Taste F2 verwendet, um für einen Eintrag in den Editiermodus zu wechseln. Mit dem **OnKeyDown**-Ereignis der Komponente läßt sich darauf einfach reagieren, indem das aktuell selektierte Objekt in den Eingabezustand versetzt wird. Dazu besitzen die TreeNode-Objekte die Methode **EditText**. Das aktuell selektierte Element wird durch die Eigenschaft **Selected** referenziert (ist kein Element selektiert, so besitzt die Eigenschaft den Wert **nil**). Beispiel:

```
procedure TfoMain.tvControlsKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if (Key=112) and (tvControls.Selected<>nil) then
    tvControls.Selected.EditText;
end;
```

Die Komponente stellt das Ereignis **OnEditing** zur Verfügung, das ausgelöst wird, wenn der Anwender versucht, einen Eintrag in den Eingabemodus zu versetzen. In diesem Ereignis kann dann das Editieren noch verhindert werden (das Ereignis besitzt dazu den Parameter **AllowEdit**). Nachdem der Editiervorgang abschlossen ist, wird das Ereignis **OnEdited** ausgelöst.

Einträge zur Laufzeit hinzufügen

Um der Struktur zur Laufzeit weitere Einträge hinzuzufügen, wird generell mit dem **TreeNodes**-Objekt gearbeitet. Dieses stellt diverse Methoden zum Einfügen neuer Einträge bereit. Als ersten besitzen die Funktionen den Parameter **Node** vom Typ **TTreeNode**. Der zweite Parameter gibt immer den Text des neuen Eintrags an. Die Methoden zum Einfügen werden in zwei Gruppen aufgeteilt. Die erste Gruppe fügt den neuen Eintrag in derselben Hierarchiestufe ein, in der sich der angegebene Bezugsknoten befindet. Bei den anderen Methoden wird der neue Eintrag als Untereintrag des Bezugsknotens eingefügt. Die Methode **Add** fügt einen neuen Eintrag auf der Hierarchiestufe des angegebenen Bezugsknotens ein, mit **AddChild** wird der Eintrag als Untereintrag des Bezugsknotens eingetra-

gen. Diese beiden Methoden fügen den neuen Eintrag am Ende der anderen Einträge in derselben Hierarchiestufe ein. Analog gibt es die Methoden **AddFirst** und **AddChildFirst**, die den Eintrag an den Anfang der Einträge stellen.

Einträge mit Daten verknüpfen

Die Knotenobjekte besitzen eine Eigenschaft **Data**, mit der für den Eintrag zusätzliche Daten vermerkt werden können. Die Eigenschaft ist ein untypisierter Zeiger und kann damit eine beliebige Struktur oder Objekt referenzieren. Mit den Methoden **AddObject** und **AddChildObject** bzw. **AddObjectFirst** und **AddChildObjectFirst** kann bereits beim Einfügen ein Zeiger auf die Eintragsdaten übergeben werden.

Einblenden der vollständigen Eintragsnamen

Insbesondere bei der Verwendung vieler Hierarchieebenen kann es durchaus vorkommen, daß der Zeichenbereich nicht mehr für die vollständige Darstellung der Eintragsnamen ausreicht. Ist die Eigenschaft **ToolTips** gesetzt, werden abgeschnittene Eintragsnamen in einem Hinweisfenster eingeblendet, sobald sich der Mauszeiger über einem solchen Eintrag befindet.

9.4.2 TListView

Übersicht

Vererbung	TObject→ TPersistent→ Tcomponent→ TControl→ TWinControl→ TCustomListView → TListView
Palettenseite	Win32
Deklariert in Unit	ComCtrls

Vererbung (CLX)	TObject→ TPersistent→ TComponent→ TControl→ TWidgetControl→ TFrameControl→ TCustomViewControl→ TListTreeView → TListView
Palettenseite	Common Controls
Deklariert in Unit	QComCtrls

Mit der Komponente **TListView** kann ebenfalls eine Liste mehrerer Einträge verwaltet werden. In diesem Fall liegt jedoch kein hierarchischer Aufbau zugrunde. Diese Ansicht findet ebenfalls im Explorer Anwendung, um den Inhalt von Ordnern anzuzeigen. Die Besonderheit dieser Komponente liegt darin, daß sie verschiedene Darstellungsarten beherrscht. Sie kann die Einträge als Liste, als Detailliste sowie in Form von kleinen oder großen Icons darstellen.

Bei der detaillierten Ansicht erfolgt eine tabellarische Ausgabe der Einträge, wobei zu jedem Eintrag weitere Informationen spaltenweise angezeigt werden. Die Spalten können mit Überschriften versehen und in der Breite verändert werden.

Struktur der Einträge

Die Liste der Einträge wird mit dem Objekt **TListItems** verwaltet, das über die Eigenschaft **Items** zur Verfügung gestellt wird. Die Eigenschaft **Count** des **ListItems**-Objekt gibt die Anzahl der Einträge an. Die einzelnen Einträge sind vom Typ **TListItem** und können indiziert über die Eigenschaft **Item** angesprochen werden.

Jeder Eintrag der Liste liegt also in Form eines **TListItem**-Objekts vor. Die Bezeichnung des Eintrags wird mit **Caption** festgelegt. Die Eigenschaft **ImageIndex** legt den Bildindex des Eintrags fest. Je nachdem welche Darstellungsart der Liste gewählt ist (große/kleine Icons), wird das Bitmap aus der Bilderliste verwendet, das der Darstellung entspricht. Diese Bilderlisten sind mit den Eigenschaften **LargeIcons** und **SmallIcons** für die ListView-Komponente festgelegt.

Die Eigenschaft **Focused** legt fest, ob der Listeneintrag den Fokus besitzt (es kann immer nur ein Element den Fokus besitzen). Mit der Eigenschaft **Selected** wird bestimmt, ob der Eintrag selektiert ist. Das gleichzeitige Markieren mehrerer Einträge ist möglich, sofern die Eigenschaft **MultiSelect** der ListView-Komponente gesetzt ist.

Festlegen der Darstellungsart

Die Komponente beherrscht die vier Darstellungsarten **Listenansicht**, **Detailansicht**, **kleine Symbole** und **große Symbole**. Die Darstellung wird mit der Eigenschaft **ViewStyle** vom Typ **TViewStyle** festgelegt. Dafür sind folgende Werte möglich:

Wert	Bedeutung
vsIcon	Große Symbole (Beschriftung unter dem Icon)
vsSmallIcon	Kleine Symbole (Beschriftung rechts vom Icon)
vsList	Listenansicht, kleine Symbole (Beschriftung rechts vom Icon), die Darstellung erfolgt spaltenweise
vsReport	Detailansicht, tabellarische Darstellung mit Icons in der linken Spalte

Werden die Einträge in der Detailansicht dargestellt, legt die Eigenschaft **ShowColumnHeaders** fest, ob die Spaltenüberschriften angezeigt werden.

Detailansicht

Wird die Liste als Detailansicht angezeigt, werden die Einträge tabellarisch mit zusätzlichen Informationen dargestellt. Die Spaltenüberschriften werden mit der Eigenschaft **Columns** festgelegt. Hinter dieser Eigenschaft verbirgt sich das Objekt **TListColumns**, das die Verwaltung der Spalten übernimmt. Jede Spalte wird durch ein Objekt des Typs **TListColumn** repräsentiert. Die einzelnen Spaltenobjekte können über die Eigenschaft **Columns** der Listenansicht oder über die Eigenschaft **Items** des ListColumns-Objekts indiziert angesprochen werden. Die Eigenschaft **Count** dieses Objekts gibt die Spaltenanzahl an. Die Spalten werden zur Entwicklungszeit mit dem Spalteneditor definiert. Die Eigenschaften der Spaltenobjekte werden dann mit dem Objektinspektor bearbeitet. Genauso können die Spalten zur Laufzeit dynamisch erstellt werden. Die Methode **Add** des ListColumns-Objekts erstellt ein neues Spaltenobjekt und gibt dieses als Funktionsergebnis zurück. Um eine Spalte zu löschen, wird das entsprechende Objekt mit der Methode **Free** freigegeben. Die Tabelle zeigt die wesentlichen Eigenschaften der Spalten.

Eigenschaften von TListColumn

Eigenschaft	Bedeutung
Caption	Spaltenüberschrift
Alignment	Ausrichtung der Überschrift (taLeftJustify, taCenter, taRightJustify)
ImageIndex	Index des Symbols, das in der Überschrift angezeigt wird
MaxWidth	Maximale Spaltenbreite
MinWidth	Minimale Spaltenbreite
Width	Spaltenbreite

Die Spaltenbreite wird in Pixel festgelegt. Wird der Eigenschaft die Konstante **ColumnTextWidth** (–1) zugewiesen, wird die Spaltenbreite an die Länge der angezeigten Texte angepaßt. Wird die Konstante **ColumnHeaderWidth** (–2) verwendet, paßt sich die Spaltenbreite an die Breite der Spaltenüberschrift an.

Jeder Listeneintrag (**TListItem**) besitzt die Listeneigenschaft **SubItems**, welche die Detailinformationen des Eintrags beinhaltet. Die einzelnen Zeilen werden in der entsprechenden Spalte der Detailansicht dargestellt.

Das Ereignis **OnColumnClick** der ListView-Komponente wird ausgelöst, wenn der Anwender auf die Überschrift einer Spalte klickt. Als Parameter

wird eine Referenz auf das jeweilige Spaltenobjekt zurückgegeben. Davon ausgehend könnte die Liste beispielsweise anhand der Werte dieser Spalte sortiert werden.

Anordnen und Sortieren der Einträge

Sofern die Einträge in der Listenansicht als Icons dargestellt werden (**ViewStyle**=**vsIcon** oder **vsSmallIcon**), können die Symbole mit der Methode **Arrange** ausgerichtet werden. Der Parameter **Code** vom Typ **TListArrangement** bestimmt dann den Bezugspunkt, auf den die Symbole ausgerichtet werden:

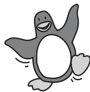
Wert	Bedeutung
arAlignBottom	Ausrichten der Einträge am unteren Rand
arAlignLeft	Ausrichten der Einträge am linken Rand
arAlignTop	Ausrichten der Einträge am oberen Rand
arAlignRight	Ausrichten der Einträge am rechten Rand
arDefault	Entsprechend der Voreinstellung ausrichten
arSnapToGrid	Einträge am Gitter ausrichten

9.5 Tabellen

TCustomGrid, TDrawGrid

Übersicht

Vererbung	TObject→ TPersistent→ TComponent→ TControl→ TWinControl→ TCustomControl→ TCustomGrid → TCustomDrawGrid → TDrawGrid
Palettenseite	Zusätzlich
Deklariert in Unit	Grids



Vererbung (CLX)	TObject→ TPersistent→ TComponent→ TControl→ TWidgetControl→ TCustomControl→ TCustomGrid → TCustomDrawGrid → TDrawGrid
Palettenseite	Zusätzlich
Deklariert in Unit	QGrids

TCustomGrid

Die Klasse **TCustomGrid** ist eine allgemeine Basisklasse für Komponenten, die Daten in tabellarischer Form darstellen. **TCustomGrid** implementiert eine Reihe von Eigenschaften und Methoden, mit denen die Darstel-

lung der Tabelle erfolgt. Die neu deklarierten Eigenschaften dieser Basisklasse sind geschützt deklariert. Damit ist diese Komponente auch eine geeignete Basisklasse für eigene Komponenten, bei denen die strukturierte Darstellung in Form einer Tabelle gefordert ist. Die Komponente beinhaltet keine Verwaltung von Tabellendaten.

Die Klasse implementiert damit im wesentlichen die Möglichkeit, Zellen darzustellen, die mit einem Rahmen umgeben sind. Ob die Zeilen- bzw. Spaltentrennlinien dargestellt werden, hängt von der Eigenschaft **Options** ab.

Die Spalten- und Zeilenanzahl wird mit den Eigenschaften **RowCount** und **ColCount** bestimmt. Die Zeilen- und Spaltenhöhe kann frei eingestellt werden. Genauso kann die Position von Zeilen und Spalten verschoben werden. Mit **FixedCols** wird die Anzahl derjenigen Spalten auf der linken Gitterseite angegeben, deren Position und Breite nicht geändert werden kann. Der Inhalt dieser Zellen kann auch nicht editiert werden. **FixedRows** gibt analog die Anzahl der oberen Zeilen an, die fixiert sind. Fixierte Zellen werden normalerweise mit der durch die Eigenschaft **FixedColor** festgelegten Farbe hinterlegt (in den Methoden zum Zeichnen kann dieses Verhalten geändert werden).

Um den Inhalt einer Zelle darzustellen, ist die Methode **DrawCell** abstrakt deklariert. Der Aufruf dieser Methode ist vollständig implementiert, so daß die Methode für alle im Zeichenbereich sichtbaren Zellen automatisch aufgerufen wird, sobald sich die Notwendigkeit der erneuten Darstellung ergibt. Das ist beispielsweise unmittelbar nach dem Anzeigen der Komponente der Fall. Genauso wird die Methode nach dem Scrollen innerhalb der Tabelle für alle sichtbaren Zellen aufgerufen. Um in dem Datengitter tatsächlich Daten darzustellen, muß somit diese Methode überschrieben werden.

Inplace-Editor

Diese Basisklasse stellt noch keinerlei Funktionalität für die Verwaltung und Speicherung der dargestellten Daten zur Verfügung. Daher ist auch die Art der dargestellten Daten noch in keiner Weise festgelegt. Trotzdem ist bereits die Möglichkeit gegeben, Zellen zu selektieren und in dem Gitter zu navigieren. Außerdem wurde bereits die Grundlage dafür geschaffen, um mit Hilfe eines sogenannten Inplace-Editors die Daten einer Zelle zu bearbeiten.

Leider ist diese Lösung nicht sehr allgemein gehalten. Als Editor wird nämlich grundsätzlich die Komponente **TInplaceEdit** verwendet, die sich direkt von **TCustomMaskEdit** ableitet. Damit ist bereits in dieser Klasse

implizit festgelegt, daß mit dem Gitter zwar beliebige Daten grafisch dargestellt, aber nur Stringwerte bearbeitet werden können. Besonders für den Komponentenentwickler wird es eine Herausforderung sein, das Beste aus dieser Bearbeitungsmöglichkeit zu machen. So könnte zum Beispiel die Textkomponente um einen DropDown-Button erweitert werden, mit dem eine Liste möglicher Eingabewerte zur Auswahl angeboten wird.

TDrawGrid

TDrawGrid ist direkt von **TCustomGrid** abgeleitet und verfügt damit über die grundlegende Fähigkeit, Daten in Tabellenform anzuzeigen. **TDrawGrid** steht in der Komponentenpalette zur Verfügung. Die Komponente ist gegenüber der Basisklasse erweitert, damit spezielle Benutzeraktionen behandelt werden können. Derartige Aktionen sind beispielsweise das Verschieben der Position von Spalten oder Zeilen. Außerdem wird die abstrakte Methode **DrawCell** für die Darstellung von Zellen überschrieben. Die Komponente implementiert zwar selbst keine Funktionen für die Darstellung, löst dafür aber das Ereignis **OnDrawCell** aus, in dessen Ereignisbehandlung das Zeichnen der Zelle implementiert werden kann. Das Ereignis übergibt als Parameter den Zeilen- und Spaltenindex sowie ein Rechteck, das den Zeichenbereich der Zelle eingrenzt. Zudem wird der Parameter **AState** übergeben. Dieser Wert ist vom Mengentyp **TGridDrawState**, um den Status der Zelle zu beschreiben. Der Wert kann somit drei Flags beinhalten.

Flag	Bedeutung
gdSelected	Die Zelle ist selektiert.
gdFocused	Die Zelle besitzt den Fokus.
gdFixed	Die Zelle ist Teil des fixierten Bereichs.

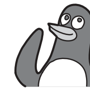
Die Zeichenfläche wird durch die Eigenschaft **Canvas** der Komponente referenziert.

TStringGrid

Übersicht

Vererbung	TObject→ TPersistent→ TComponent→ TControl→ TWinControl→ TCustomControl→ TCustomGrid → TCustomDrawGrid → TDrawGrid→ TStringGrid
Palettenseite	Zusätzlich
Deklariert in Unit	Grids

Vererbung (CLX)	TObject→ TPersistent→ TComponent→ TControl→ TWidgetControl→ TCustomControl→ TCustomGrid → TCustomDrawGrid → TDrawGrid→ TStringGrid
Palettenseite	Zusätzlich
Deklariert in Unit	QGrids



Abweichend von der Vorgängerkomponente **TDrawGrid** implementiert diese Komponente die Verwaltung eines zweidimensionalen String-Arrays, dessen Werte als Text in den entsprechenden Zellen des Gitters dargestellt werden. Diese Werte können über den Inplace-Editor direkt bearbeitet werden.

Der Zugriff auf die Zellenwerte erfolgt mit der Eigenschaft **Cells** unter Angabe des Zeilen- und Spaltenindex. Beispiel:

```
StringGrid.Cells[1, 2]:='Wert für Zeile 1, Spalte 2';
```

Zudem können alle Strings einer Zeile in Form einer Stringliste behandelt werden. Die indizierte Eigenschaft **Cols** ermöglicht den Zugriff auf alle Strings einer Zeile, wobei als Index die Zeilennummer angegeben wird. Entsprechend können mit der Eigenschaft **Rows** alle Strings der als Index angegebenen Spalte in Form eines **TStrings**-Objekts angesprochen werden.

TValueListEditor

Übersicht

Vererbung	TObject→ TPersistent→ TComponent→ TControl→ TWinControl→ TCustomControl→ TCustomGrid → TCustomDrawGrid → TValueListEditor
Palettenseite	Zusätzlich
Deklariert in Unit	Grids

Diese Komponente ist in der CLX nicht enthalten.

Mit dem ValueListEditor wurde in Delphi 6 die Möglichkeit geschaffen, Werte in einem Datengitter zu bearbeiten, die in einer Stringliste definiert sind. Die Einträge der Stringliste müssen dem Format einer Ini-Datei entsprechen, zum Beispiel `Eintrag1=17`. Dabei ist Eintrag1 der Schlüssel (Key), nach dem Gleichheitszeichen steht der zugehörige Wert. Die Liste ist der Eigenschaft **Strings** hinterlegt. Die Spaltentitel sind in der Eigenschaft **TitleCaptions** definiert. Um die in der Liste dargestellten Werte im Programm auszuwerten, verwenden Sie die entsprechenden Funktionen

von Stringlisten (beispielsweise liefert `Strings.Values['Eintrag1']` den Wert zu dem Eintrag Eintrag1). Mit der Eigenschaft **KeyOptions** läßt sich festlegen, ob Einträge aus der Liste gelöscht werden dürfen, und ob das Anlegen neuer Einträge und das Bearbeiten von Einträgen möglich ist. Außerdem kann der Eintragsname eindeutig gehalten werden.

Die übrigen Eigenschaften der Komponente werden Ihnen bereits von den vorher angesprochenen Grid-Komponenten bekannt vorkommen.

9.6 Drag&Drop

9.6.1 Einführung

Überblick

Allgemein gesprochen bedeutet der Vorgang Drag&Drop, daß der Anwender mit der Maus ein bestimmtes Element zu einem Zielobjekt verschiebt. Dazu wird das Element bei gedrückt gehaltener Maustaste zu dem anderen Objekt gezogen und dort losgelassen. Im folgenden wird das Objekt als **Quelle** bezeichnet, von dem der Drag&Drop-Vorgang ausgeht. Das Objekt, das das Quellobjekt aufnehmen soll, wird als **Ziel** bezeichnet.

Der Drag&Drop-Mechanismus ist bereits in der Klasse **TControl** implementiert, so daß alle Steuerelemente in Delphi von dieser Möglichkeit Gebrauch machen können, sofern die entsprechenden Eigenschaften und Ereignisse veröffentlicht sind.

Durchführung von Drag&Drop-Operationen

Jedes Objekt bestimmt für sich selbst, ob es der Ausgangspunkt einer Drag&Drop-Operation sein kann. Ist das der Fall, wird beim Start des Vorgangs der Mauszeiger entsprechend dargestellt. Während des Drag&Drop-Vorgangs kennzeichnet der Mauszeiger, ob das Element, über dem er sich befindet, als mögliches Ziel in Frage kommt. Es ist immer Sache des Zielobjekts, zu entscheiden, ob es das gezogene Element akzeptiert. Ist das der Fall, wird ein Ereignis ausgelöst, wenn der Anwender das Objekt an dieser Stelle ablegt.

Starten des Drag&Drop-Vorgangs

Sofern die Eigenschaft **DragMode** einer Komponente den Wert **dmAutomatic** besitzt, wird der Vorgang automatisch gestartet, sobald der Anwender das Steuerelement mit der linken Maustaste anklickt. Der Mauszeiger wird dann dementsprechend dargestellt, und das Ereignis **OnStartDrag** der Komponente wird ausgelöst. Das Ereignis **OnClick**

unterbleibt in diesem Fall. Ist ein automatischer Start nicht erwünscht, muß die Drag&Drop-Operation mit der Methode **BeginDrag** vom Programm gesteuert gestartet werden. Die günstigste Stelle hierfür ist das **OnMouseDown**-Ereignis. Diese Vorgehensweise hat den Vorteil, daß Elemente nach wie vor wie gewohnt mit der Maus angeklickt werden können, sofern für den Parameter **Immediate** der Wert **false** angegeben wird. In diesem Fall wird der Ziehvorgang erst gestartet, wenn der Anwender den Mauszeiger tatsächlich vom Element wegbewegt. Genaugenommen startet der Vorgang erst, wenn der Mauszeiger einen bestimmten Abstand zu dem Punkt einnimmt, an dem er gedrückt wurde. Per Voreinstellung beträgt dieser Radius 5 Pixel. Mit dem optionalen Parameter **Threshold** kann ein davon abweichender Radius beim Aufruf von **BeginDrag** explizit angegeben werden. Auf die weiteren Vorgänge der Drag&Drop-Operation hat die Art des Startens keinerlei Auswirkungen.

Unmittelbar nach dem Starten des Vorgangs wird das Ereignis **OnStartDrag** ausgelöst. Bei der nachfolgend gezeigten Vorgehensweise für die Implementierung von Drag&Drop-Operationen hat das Ereignis keine sehr große Bedeutung.

Verschieben des Objekts

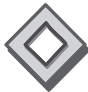
Der Anwender erwartet, daß die Form des Mauszeigers signalisiert, ob ein Objekt das Ziel der Drag&Drop-Operation sein kann. Ob ein Objekt zum Ziel der Operation werden kann, kann nur es selbst bestimmen. Wird der Mauszeiger während des Drag&Drop-Vorgangs über ein Steuerelement bewegt, wird für dieses das Ereignis **OnDragOver** ausgelöst. Das Ereignis besitzt den Parameter **Source**, der das Quellobjekt angibt. Anhand dessen entscheidet das Steuerelement, ob es als Ziel der Drag&Drop-Operation geeignet ist. Ist das der Fall, wird der Parameter **Accept** auf den Wert **true** gesetzt.

Beenden des Drag&Drop-Vorgangs

Der Drag&Drop-Vorgang wird beendet, sobald der Anwender die Maustaste losläßt. In jedem Fall wird dabei das Ereignis **OnEndDrag** des Quellobjekts ausgelöst. Dabei wird als Parameter das **Target**-Objekt angegeben. Das ist eine Referenz auf das Objekt, über dem der Vorgang beendet wurde, unabhängig davon, ob dieser das Drag-Objekt akzeptiert. Als weitere Parameter werden die Werte **X** und **Y** übergeben, welche die aktuelle Position des Mauszeigers kennzeichnen.

Sofern das Zielelement das Quellobjekt akzeptiert, wird das Ereignis **OnDragDrop** der Zielkomponente ausgelöst. Als Parameter **Source** wird die Referenz auf das Steuerelement übergeben, von dem die Drag&Drop-Operation ausging. Dazu wird die aktuelle Position des Mauszeigers über-

geben. Die Zielkomponente kann dann abhängig von der Quelle die gewünschte Funktion durchführen.



Auf der CD finden Sie das in Abbildung 9.8 dargestellte Beispiel »Drag-DropDemo«.

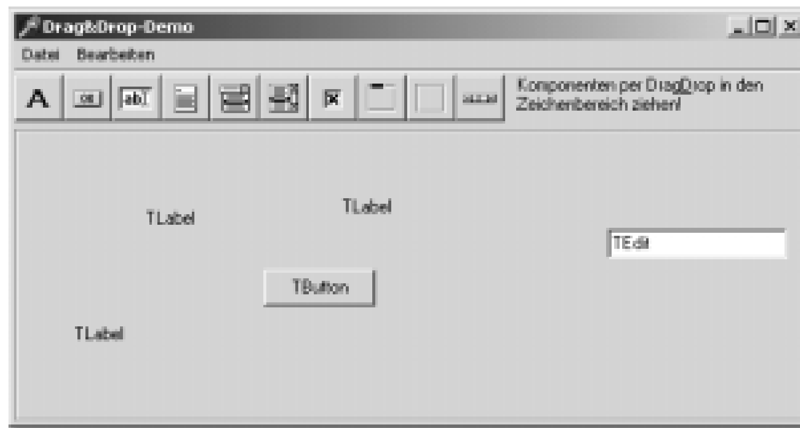


Abbildung 9.8 Drag&Drop-Demo

Über die Symbolleiste können Komponenten in den Zeichenbereich eingefügt werden. Die Komponenten werden an der Stelle dargestellt, an der die Drag&Drop-Operation beendet wurde.

Für die Label-Komponente wurde die beschriebene Vorgehensweise für Drag&Drop-Operationen realisiert. Die Eigenschaft **DragMode** des Label-Toolbuttons besitzt dazu den Wert **dmAutomatic**, so daß der Drag&Drop-Vorgang automatisch startet. Der Zeichenbereich wird durch eine GroupBox-Komponente (**gbEdit**) gebildet, welche die gesamte Fensterfläche ausfüllt. In dem Ereignis **OnDragOver** dieser Komponente wird geprüft, ob es sich bei dem Sender um den Button zum Einfügen der Label-Komponente handelt. Ist das der Fall, wird die Drag&Drop-Operation akzeptiert:

```
procedure TfoMain.gbEditDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source=tbLabel then Accept:=true;
end;
```

Im Ereignis **OnDragDrop** der GroupBox wird schließlich eine neue Label-Komponente erzeugt und an der Position dargestellt, an der die Maustaste losgelassen wurde:

```
procedure TfoMain.gbEditDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source=tbLabel then begin
    with TLabel.Create(Self) do begin
      Parent:=gbEdit;
      Left:=X;
      Top:=Y;
      Caption:='Label';
    end;
  end;
end;
```

Wollte man das gesamte Beispiel in dieser Art durchführen, müßte in den beiden Ereignissen **OnDragOver** und **OnDragDrop** für jeden der Toolbuttons eine entsprechende if-Anweisung eingefügt werden. Die Nachteile dieser Methode sind erstens der relativ große Schreibaufwand, und zweitens wäre die Anpassung der Anwendung relativ umständlich, wenn als Quellobjekte anstatt der Toolbuttons andere (oder sogar zusätzliche) Quellobjekte verwendet werden sollen.

Vorgehensweise unter Anwendung von Drag-Objekten

Eine von der obigen Vorgehensweise abweichende Möglichkeit zur Durchführung von Drag&Drop-Operationen ist die Verwendung von Drag-Objekten. Dabei verpackt das sendende Objekt die vom Empfänger benötigten Informationen beim Start des Drag&Drop-Vorgangs in ein Drag-Objekt. Die Auswertung der Drag&Drop-Ereignisse beim Empfänger wird dann nicht mehr aufgrund der Quellkomponente, sondern aufgrund des übertragenen Drag&Drop-Objekts durchgeführt. Der Empfänger hat damit keinen direkten Bezug zum Sender. Das bedeutet gleichzeitig, daß der Empfänger unabhängig vom Sender ist.

Beim Start der Drag&Drop-Operation wird im **OnStartDrag**-Ereignis der Quellkomponente das Drag-Objekt erstellt und dem Parameter **DragObject** übergeben. Die allgemeine Basisklasse dieser Objekte ist die Klasse **TDragObject**. Da dieses Objekt die Daten der Drag&Drop-Operation übertragen soll, wird hierfür in jedem Fall ein spezielles Objekt abgeleitet. Eine geeignete Basisklasse für diese Objekte ist die Klasse **TBaseDragControlObject**, die von **TDragObject** abstammt.

Übergibt die Quellkomponente im **OnStartDrag**-Ereignis ein spezielles Drag-Objekt, wird für die Ereignisse der Zielkomponente mit dem Parameter **Source** nicht mehr die Referenz auf die Quellkomponente übergeben, sondern die Referenz auf das Drag-Objekt. Daher kann im **OnDragOver**-Ereignis unabhängig von der Quelle aufgrund des Drag-Objekts bestimmt werden, ob die Drag&Drop-Operation möglich ist. Entsprechend kann im **OnDragDrop**-Ereignis verfahren werden.

Beispiel für Drag-Objekte

Es wird wieder das Beispiel »DragDropDemo« zugrunde gelegt. In diesem Beispiel ist ausschließlich gefordert, daß Komponenten in den Zeichenbereich eingefügt werden können. Alle Komponenten sind Steuerelemente und haben damit als Basisklasse zumindest die Klasse **TControl** gemeinsam. Das dynamische Erzeugen von Steuerelementen unterscheidet sich nicht.

Definition des Drag-Objekts

Daher muß keine weitere Unterscheidung der zu erzeugenden Steuerelemente getroffen werden. Ein Drag-Objekt wird von **TBaseDragControlObject** abgeleitet, das um ein Feld **FClass** vom Typ **TClass** erweitert wurde. Um das Erzeugen des Drag-Objekts etwas bequemer zu gestalten, wird ein zusätzlicher Konstruktor definiert, der als zusätzlichen Parameter den Klassentyp erwartet. Dieser neue Konstruktor ruft den geerbten Konstruktor **Create** auf und speichert den Klassentyp im Feld **FClass**. Dieses Feld ist vom Typ **TControlClass**:

```
constructor TDragClassObject.CreateCls(AControl: TControl;
  AClass: TControlClass);
begin
  inherited Create(AControl);
  FClass:=AClass;
end;
```

Starten des Drag&Drop-Vorgangs

Im Gegensatz zu der ersten Vorgehensweise muß in diesem Fall das Ereignis **OnStartDrag** der ToolButtons verwendet werden, um das Drag-Objekt zu erzeugen. Dieses Objekt wird dem Rückgabeparameter der Ereignismethode übergeben. Dem Konstruktor wird das auslösende Steuerelement sowie die zu erzeugende Klasse übergeben. Das Ereignis **OnStartDrag** muß für jeden ToolButton getrennt in dieser Art programmiert werden:

```
procedure TfoMain.tbLabelStartDrag(Sender: TObject;
  var DragObject: TDragObject);
begin
  DragObject:=TDragClassObject.CreateCls(TControl(Sender), TLabel);
end;
```

Als nächstes wird die Methode **OnDragOver** der GroupBox erstellt. Diese orientiert sich nun nicht mehr an der Quellkomponente, sondern am Drag-Objekt. Sofern dieses vom Typ **TDragClassObject** ist, kann der Vorgang akzeptiert werden:

Akzeptieren des Drag&Drop-Vorgangs

```
procedure TfoMain.gbEditDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TDragClassObject then Accept:=true;
end;
```

Auch im Ereignis **OnDragDrop** wird mit dem Parameter **Source** in diesem Fall nicht die Quellkomponente, sondern das Drag-Objekt referenziert. Zum Erzeugen der Steuerelemente habe ich hier etwas tiefer in die Trickkiste gegriffen. Zunächst wird für die im Drag-Objekt übergebene Klasse der Konstruktor **Create** zum Erzeugen einer neuen Objektinstanz aufgerufen. Damit die Komponenten eine Beschriftung erhalten, wird ihre Eigenschaft **Caption** gesetzt (einige der Steuerelemente zeigen die Beschriftung zwar nicht an, das soll an dieser Stelle aber nicht weiter stören). Die Eigenschaft **Caption** ist in der Basisklasse **TControl** geschützt deklariert, so daß der direkte Zugriff auf diese Eigenschaft nicht möglich ist. Innerhalb der Unit, die eine Klasse deklariert, kann der Zugriff auf die geschützten Felder hingegen uneingeschränkt durchgeführt werden. Es genügt daher, eine Dummy-Klasse anzulegen, die von **TControl** abgeleitet ist:

Übernehmen des Drag&Drop-Vorgangs

```
TPropControl = class(TControl);
```

Damit entspricht die Klasse **TPropControl** exakt der Klasse **TControl**. Auf diese Weise können auch beliebige **TControl**-Objekte als **TPropControl** interpretiert werden. Der Unterschied ist, daß aufgrund dieses Typecasts auch die geschützten Teile des Objekts angesprochen werden können. Es ist somit auch der Zugriff auf die Eigenschaft **Caption** verfügbar. Ich gebe gerne zu, daß diese Vorgehensweise nicht unbedingt der sauberste Programmierstil ist. In manchen Fällen kann diese Hintertür allerdings sehr viel Arbeit sparen, wie auch in diesem Beispiel:

```
procedure TfoMain.gbEditDragDrop(Sender, Source: TObject;
                                 X, Y: Integer);
var
  ctrl: TControl;
begin
  if Source is TDragClassObject then begin
    ctrl:=TDragClassObject(Source).FClass.Create(Self);
    with TPropControl(ctrl) do begin
      Parent:=gbEdit;
      Left:=X;
      Top:=Y;
      Caption:=ClassName;
    end;
  end;
end;
```

9.7 Docking

9.7.1 Einführung

Eine äußerst leistungsfähige Funktionalität, über die bereits die Basisklasse **TControl** verfügt, ist das **Docking**. Nachdem die grundlegende Funktionalität zum Docken von Elementen bereits in der Klasse **TControl** implementiert ist, kann der Mechanismus auf alle Steuerelemente angewendet werden. Der häufigste Einsatzfall ist allerdings zweifelsohne das Docking von Formularen, so daß ich mich im folgenden auf diesen etwas spezielleren Fall beziehen werde. Die Delphi-IDE macht selbst regen Gebrauch von dieser Möglichkeit, indem die verschiedenen Fenster der IDE zusammengefaßt werden können. Beispielsweise können Bereiche der Werkzeugleiste als eigene Toolfenster dargestellt werden, indem sie aus der Symbolleiste gezogen werden. Genauso können sie wieder in die Symbolleiste eingedockt werden.

Beim Docking von Elementen werden diese sozusagen aneinander geheftet und erscheinen in einem gemeinsamen Rahmenelement. Konkret bedeutet das, daß beispielsweise mehrere Formulare miteinander verbunden werden können, indem sie übereinander geschoben werden. So werden sie als eigenständige Bereiche in einem gemeinsamen Fenster dargestellt.

Wird ein Element mit der Maus über ein anderes bewegt, so daß diese aneinander geheftet werden, bezeichnet man das als **Drag&Dock-Vorgang**. Wenn zwei Elemente miteinander verbunden werden, so daß sie sich gemeinsam in einem übergeordneten Bereich darstellen, wird dieser Vorgang mit dem Begriff **Docking** bezeichnet. Der Auslöser einer Docking-Aktion muß demnach nicht unbedingt eine Mausaktion des Anwenders sein. Man kann sich daher Drag&Dock-Vorgänge auch besser vorstellen, indem man den Vorgang in zwei separate Teile trennt (in der Praxis ist das auch tatsächlich so!). Der erste Teil ist der Drag-Vorgang. Dabei wird ein Element mit dem Ziel, das Element mit einem anderen zu verbinden, mit der Maus an eine bestimmte Position gezogen. Während des Drag-Vorgangs wird ständig beim Ziel nachgefragt, ob dieses das gezogene Element aufnehmen kann und will. Ist das der Fall, wird der Bereich, in dem das Element aufgenommen wurde, durch einen Rahmen angedeutet. Der zweite Teil der Drag&Dock-Operation ist dann das eigentliche Docking. Dabei werden die beiden Elemente in einem gemeinsamen Parent zusammengefaßt.

Allgemeine Vorgehensweise für das Docking von Formularen

Dieser Abschnitt soll die Drag&Drop-Vorgänge anhand eines Beispiels aufzeigen. Ausgangspunkt sind zwei Formulare, die miteinander verbunden werden sollen. Abhängig von der Position, an die das Formular geschoben wird, sollen die Bereiche nebeneinander angeordnet werden oder in ein Register aufgenommen werden.

Dazu wird Formular A über das andere Formular B geschoben. Die Docking-Verwaltung sorgt dafür, daß in Formular B angefragt wird, ob eine Verbindung mit dem anderen Formular überhaupt möglich ist. Ist das nicht der Fall, wird das Docking sofort unterbrochen. Sofern das Docking durchgeführt werden kann, wird aufgrund der aktuellen Position des Mauszeigers der Fensterausschnitt berechnet, in den das Formular eingebunden wird. Befindet sich der Mauszeiger eher am oberen Fensterrand, werden die beiden Fenster übereinander angeordnet; steht der Mauszeiger eher im mittleren Fensterbereich, werden beide Formulare in ein Register eingefügt. Damit der Anwender sieht, wie die Formulare verbunden werden, berechnet das Formular A ein Docking-Rechteck. Dieses Rechteck stellt einen Rahmen dar, der die zukünftige Einordnung des Formulars zeigt. Dieses Fenster dient tatsächlich nur der Vorschau und wird beim endgültigen Einbetten des anderen Formulars nicht zugrunde gelegt.

Der Anwender kann den Docking-Vorgang nur dadurch unterbrechen, daß er das Fenster wieder aus dem Bereich des ersten Fensters zieht (entscheidend ist immer die Position des Mauszeigers). Läßt der Anwender die Maustaste los, während sich der Mauszeiger im Bereich von Formular A befindet, wird das Docking durchgeführt. Fenster A bekommt das durch die Komponentennachricht **CM_DOCKCLIENT** mitgeteilt. Aufgrund der aktuellen Position des Mauszeigers wird erneut das Docking-Rechteck berechnet. Außerdem wird die Ausrichtung der beiden Fenster zueinander bestimmt. Nun wird nicht etwa das Formular B direkt in das Formular A eingefügt. Vielmehr wird ein gemeinsamer Docking-Host erstellt, der beide Fenster aufnimmt. Der Docking-Host ist in diesem Fall ein Formular, das die anderen beiden Formulare als Teilbereiche aufnimmt. Um die unterschiedlichen Dockingtypen zu realisieren (Einfügen als Fensterbereich oder Einfügen in Register), werden auch verschiedene Docking-Hosts notwendig. Beim Einfügen als Teilbereich kann der Dokking-Host ein leeres Formular sein, im Falle von Registern kann als Dokking-Host eine PageControl-Komponente verwendet werden.

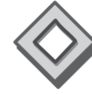
Fenster können aus dem gemeinsamen Docking-Host entfernt werden, indem sie geschlossen werden oder die Titelzeile doppelt angeklickt wird. Wird das vorletzte Fenster aus dem Docking-Host entfernt, wird dieser in vielen Fällen unnötig. Das letzte darin enthaltene Formular wird daher wieder als eigenständiges Formular dargestellt, und der Docking-Host wird freigegeben.

9.7.2 Implementierung der Drag&Dock-Funktionalität

Starten des Drag-Vorgangs

Wie bei Drag&Drop muß auch beim Drag&Dock der Vorgang aufgrund eines Ereignisses gestartet werden. Bei Formularen wäre das der Zeitpunkt, zu dem der Anwender beginnt, das Formular zu verschieben. Ist die Eigenschaft **DragKind** des Formulars auf **dkDrag** gesetzt, erfolgt der Start des Drag&Dock-Vorgangs, sobald der Anwender die Position des Formulars ändert. Sie erkennen das geänderte Verhalten sofort, denn normalerweise wird das Fenster mitsamt dem kompletten Inhalt verschoben und ständig neu gezeichnet. Bei dem laufenden Drag&Dock-Vorgang ist das nicht so. In diesem Fall zeigt ein grauer Rahmen die Zielposition an, das Fenster bleibt vorerst an seiner ursprünglichen Stelle. Vergleichen Sie beispielsweise das Verschieben des Objektinspektors, wenn Sie über dessen Kontextmenü die Option **Andockbar** umschalten.

Das Beispielprogramm »DragDockDemo« zeigt verschiedenfarbige Fenster an, die in einen Bereich (**Docking-Host**) des Hauptformulars eingefügt werden können. Die Fensterklasse **TfoClient** stellt die Client-Fenster (Docking-Clients) dar. Damit der Drag&Dock-Vorgang gestartet werden kann, wird die Eigenschaft **DragKind** auf den Wert **dkDrag** gesetzt. Mit der CheckBox **Andockbar** wird diese Eigenschaft umgeschaltet:



```
procedure TfoClient.chDockableClick(Sender: TObject);
begin
  {Docking aktivieren/deaktivieren}
  if chDockable.Checked then DragKind:=dkDock
  else DragKind:=dkDrag;
end;
```

Docking-Host

Theoretisch kann jede von **TWinControl** abgeleitete Komponente, die einen Container für andere Steuerelemente darstellt, als Docking-Host verwendet werden. Typische Vertreter dieser Klasse sind die Komponenten **TPanel** und **TPageControl**. Damit diese als Docking-Host zur Verfügung stehen, indem sie die Docking-Botschaften auswerten und verarbeiten, wird lediglich deren Eigenschaft **DockSite** gesetzt.

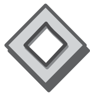
Docking mit TPageControl

Die Komponente **TPageControl** tut ihren Dienst als Docking-Host ausgezeichnet, sobald die Eigenschaft **DockSite** gesetzt ist. Wird ein beliebiger Docking-Client in den Bereich des PageControls gezogen, so wird für den Client eine neue Registerseite angelegt, in der er schließlich angezeigt wird. Um den Docking-Client wieder aus dem Register zu entfernen, genügt ein Doppelklick auf den Registereintrag.

Docking mit TPanel

Wird für eine Panel-Komponente die Eigenschaft **DockSite** gesetzt, kann auch sie als Docking-Host für beliebige Docking-Clients verwendet werden. Wird ein Client in den Bereich der Komponente gezogen, wird dieser in das Panel aufgenommen. Die ursprüngliche Größe des Client-Bereichs wird beibehalten, unabhängig davon, ob die im Panel verfügbare Zeichenfläche für eine vollständige Darstellung ausreicht. Diese Verhaltensweise ist für eine sinnvolle Verwendung von Drag&Dock unbrauchbar. Daher verfügt die Panel-Komponente über einen Dock-Manager, der das Einfügen des Docking-Clients in die Hand nimmt. Er sorgt dafür, daß der Client eine Titelzeile bekommt. Mittels dieser Titelzeile kann der Bereich aus dem Docking-Host wieder herausgetrennt werden, um ihn

wieder als eigenständiges Formular darzustellen. Dazu verfügt die Titelzeile über einen Button, der den Bereich verbirgt. Der Docking-Manager ist aktiv, sobald die Eigenschaft **UseDockManager** gesetzt ist.



Mit dem Beispiel »DragDockDemo« können Sie diese Schritte nachvollziehen. Das Hauptformular (siehe Abbildung 9.9) verfügt über zwei Docking-Hosts. Der erste Docking-Host wird durch eine Panel-Komponente gebildet. Den Effekt, den die Verwendung des Docking-Managers mit sich bringt, können Sie einfach ersehen, indem Sie die CheckBox **Docking-Manager** umschalten. Aufgrund dieser CheckBox wird lediglich die Eigenschaft **UseDockManager** beeinflußt.

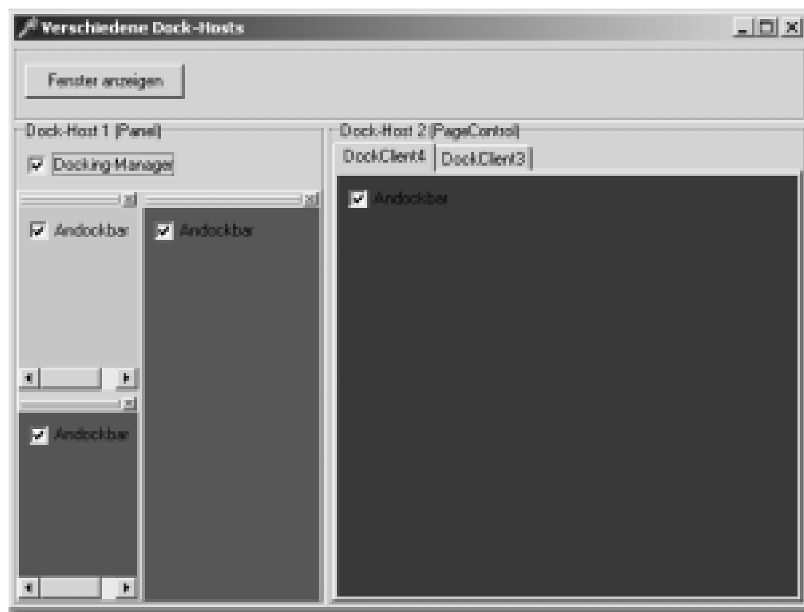


Abbildung 9.9 Verschiedene Formen des Formular-Dockings

9.7.3 Fortgeschrittene Docking-Techniken

Akzeptieren bestimmter Docking-Clients

Der Docking-Host kann aufgrund des Ereignisses **OnDockOver** entscheiden, ob er den Docking-Client aufnimmt. In dem Ereignis wird mit dem Parameter **Source** eine Referenz auf ein Drag&Dock-Objekt übergeben, das zur Verwaltung des aktuellen Drag&Dock-Vorgangs verwendet wird. Dieses Objekt referenziert den Docking-Client mit der Eigenschaft **Cont-**

rol. Anhand dessen kann überprüft werden, ob das Quellobjekt eingefügt werden kann. Das Ergebnis der Überprüfung wird für den Rückgabeparameter **Accept** angegeben:

```
procedure TfoClient.FormDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept:=Source.Control is TfoClient;
end;
```

Das erweiterte Beispiel

Das Beispiel »ExtDragDropDemo« baut auf dem im letzten Abschnitt gezeigten Beispiel auf. Ziel ist es, auch die verschiedenen Client-Fenster zusammenfassen zu können. Ich stelle in diesem Beispiel den kompliziertesten aller Fälle dar. Dabei werden die Docking-Clients abhängig von der Position des Mauszeigers entweder als nebeneinander liegende Bereiche dargestellt oder in einem Register aufgenommen. Dazu müssen die Client-Fenster entsprechend erweitert werden. Der erste Schritt dafür ist, daß diese Fenster andere Docking-Clients akzeptieren. Das wird dadurch erreicht, daß deren Eigenschaft **DockSite** gesetzt wird. Sofern lediglich gefordert wäre, die Clients in eigenen Bereichen darzustellen, könnte aufgrund der Aktivierung des Docking-Managers bereits das gewünschte Ergebnis erzielt werden. Nachdem hier aber alternativ auch die Darstellung in einem Register gefordert ist, führt die Anwendung des Docking-Managers nicht zum gewünschten Ziel. In diesem Fall ist wesentlich mehr Handarbeit erforderlich.

Erweitertes Docking mehrerer Fenster

In dem beschriebenen Fall, bei dem das Docking auf mehrere Arten durchgeführt werden kann, wird eine neue Vorgehensweise angewendet. Zunächst werden die Docking-Clients auch als Docking-Host gekennzeichnet, indem die Eigenschaft **DockSite** gesetzt wird. Sobald versucht wird, ein Fenster anzudocken, muß der Anwender den Hinweis bekommen, wie das Fenster eingedockt wird, ob es also als Bereich dargestellt oder in einem Register eingebunden wird. Dem Anwender wird das durch das Docking-Rechteck angezeigt. Auch in diesem Fall hängt die Art des Dockings wieder von der Position des Mauszeigers innerhalb des Docking-Hosts ab. Um das Docking-Rechteck entsprechend darzustellen, muß an dieser Stelle in die Docking-Verwaltung eingegriffen werden. Das

mit Delphi gelieferte Beispiel zum Thema Drag&Drop bietet hierzu einige nützliche Routinen an, die auch in diesem Beispiel ausgezeichnet verwendet werden können.

Berechnen des Docking-Rechtecks

Normalerweise wird das Docking-Rechteck automatisch berechnet und entsprechend der aktuellen Mausposition dargestellt. Die Standarddarstellung ist für diesen sehr speziellen Fall mit verschiedenen Docking-Möglichkeiten nicht aussagekräftig genug. Mit dem Docking-Objekt, das im **OnDockOver**-Ereignis zur Verfügung steht, kann die Größe und Position dieses Rechtecks benutzerdefiniert eingestellt werden. Daher wird das Rechteck im **OnDockOver**-Ereignis berechnet. Die Methode **ComputeDockingRect** liefert das Rechteck abhängig von der aktuellen Mausposition zurück und gibt darüber Auskunft, wie der neue Docking-Client den anderen gegenüber ausgerichtet wird:

```
procedure TfoClient.FormDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
var
  ARect: TRect;
begin
  Accept:=Source.Control is TfoClient;
  if Accept and
    (ComputeDockingRect(ARect, Point(X, Y))<>alNone)
  then Source.DockRect:=ARect;
end;
```

Damit wird das Rechteck bereits korrekt im Docking-Host eingeblendet.

Abschließen des Docking-Vorgangs

Erzeugen eines neuen Docking-Hosts

Sobald der Anwender den Docking-Vorgang beendet, wird die Nachricht CM_DOCKCLIENT an den Docking-Host versendet. Das ist genau der Eingriffspunkt, um die Standardbehandlung der Docking-Verwaltung zu umgehen. Im vorliegenden Fall sind die einzelnen Fenster in erster Linie Docking-Clients. Damit diese aneinander gedockt werden können, wird, indem die Eigenschaft **DockSite** gesetzt wird, der Docking-Verwaltung gegenüber so getan, als seien sie gleichzeitig auch Docking-Hosts. Sie bekommen damit auch die entsprechenden Nachrichten zugeschickt. Um nun die unterschiedlichen Docking-Arten zu realisieren, ist es keine geeignete Vorgehensweise, in einem dieser Client-Fenster ein anderes Fenster einzulagern. Dazu müßte dieses Formular geändert werden. Sollen die verschiedenen Bereiche nebeneinander dargestellt werden, wäre die Formularfläche ein geeigneter Docking-Bereich (das Verhalten gleicht

dem des Panels). Die Anordnung in Form eines Registers erfordert eine Page-Control-Komponente. Daher wird abhängig von der Art des Dokkings ein übergeordnetes Docking-Formular erzeugt, das dann beide Clients aufnimmt.

Vorbereitung des PlainHosts

Der **PlainHost** soll die Anordnung mehrerer nebeneinander liegender Client-Bereiche ermöglichen. Dazu wird ein neues Formular verwendet, zusätzliche Komponenten werden nicht benötigt. Damit das Formular Docking-Clients aufnehmen kann, wird die Eigenschaft **DockingSite** gesetzt. Die Docking-Verwaltung kann vom Docking-Manager übernommen werden, da hier keine weitere Unterscheidung der Docking-Art getroffen werden muß (**UseDockingManager=true**). Dieser Docking-Host kann schließlich auch weitere Clients aufnehmen. Daher wird in den Ereignissen **OnDockOver** und **OnGetSiteInfo** der Typ des Docking-Clients überprüft.

Vorbereitung des TabHosts

Mit dem **TabHost** werden die Clients in der darin enthaltenen PageControl-Komponente aufgenommen. In diesem Fall ist die PageControl-Komponente der Docking-Host, daher wird dessen Eigenschaft **DockSite** gesetzt, und die Docking-Ereignisse werden ausgewertet.

Vorbereitung des Docking-Clients

Das Erzeugen des speziellen Docking-Hosts erfolgt aufgrund der Nachricht CM_DOCKCLIENT im Client-Formular, das sich zu diesem Zeitpunkt noch verhält wie ein Docking-Host. Eigentlich ist es aber nur Aufgabe des Clients, die Anforderung zum Docken an den tatsächlichen Docking-Host weiterzugeben (**PlainHost** oder **TabHost**). Mit der Methode **ManualDock** wird ein Element in den angegebenen Dock-Host eingedockt. Als Parameter wird entweder das Kontrollelement angegeben, an dem der neue Bereich ausgerichtet wird, oder direkt die Ausrichtung innerhalb des Clients:

```
  TfoClient = class(TForm)
    ...
  private
    { Private-Deklarationen}
    procedure CMDockClient(var Message: TCMDockClient);
                                    message CM_DOCKCLIENT;
    ...
implementation
...
procedure TfoClient.CMDockClient(var Message: TCMDockClient);
var
  ARect: TRect;
  DockingAlign: TAlign;
```

```
  Host: TForm;
  ds: TControl;
begin
  if Message.DockSource.Control is TfoClient then begin
    {Ausrichtung des Clients berechnen}
    DockingAlign:= ComputeDockingRect(ARect,
                     SmallPointToPoint(Message.MousePos));
    {Zwischenvariable ds ist der Docking-Client}
    ds:=Message.DockSource.Control;
    {Abh. von der Ausrichtung -> mehrere Bereiche oder Register}
    if DockingAlign=alClient then begin
      Host:=TfoTabHost.Create(Application); {neuer Host}
      Host.BoundsRect:=Self.BoundsRect; {Größe d. Host anpassen}
      {Beide Clients im Host eindocken}
      ManualDock(TfoTabHost(Host).pcDockHost, nil, alClient);
      ds.ManualDock(TfoTabHost(Host).pcDockHost, nil, alClient);
    end else begin
      Host:=TfoPlainHost.Create(Application); {neuer Host}
      Host.BoundsRect:=Self.BoundsRect;
      ManualDock(Host, nil, alNone);
      ds.ManualDock(Host, nil, DockingAlign);
    end;
    {Die Clients müssen nun keine Dock-Ereignisse mehr
     weitergeben}
    DockSite:=False;
    TForm(ds).DockSite:=false;
    {neuen Dock-Host anzeigen}
    Host.Show;
  end;
end;
```

Aufheben von Docking-Verbindungen

Nachdem die Docking-Verbindung eines Clients aufgehoben wurde, muß überprüft werden, ob sich eventuell nur noch ein letzter Client im Dokking-Host befindet. In diesem Fall wird auch diese Verbindung aufgehoben, so daß das letzte Formular wieder für sich alleine steht. Anschließend wird der Docking-Host freigegeben. Um die letzte Verbindung aufzulösen, wird die Methode **ManualFloat** verwendet. Als Parameter wird die Position und Größe des Fensters angegeben, mit der es nach dem Beenden der Docking-Verbindung erscheinen soll. Als Position wird die aktuelle Position des Docking-Hosts verwendet. Hingegen soll das Formular die ursprünglichen Abmessungen annehmen. Diese Werte können mit den Eigenschaften **UndockWidth** und **UndockHeight** bestimmt werden.

Im Docking-Host wird vor dem Aufheben einer Docking-Verbindung das Ereignis **OnUndock** ausgelöst (beim Eintreten des Ereignisses besteht die Verbindung noch). Zunächst wird an dieser Stelle die Eigenschaft **DockSite** gesetzt, damit das Formular auf die Anforderung zum Eindocken eines anderen Formulars wieder entsprechend reagieren kann. Um den Docking-Host korrekt freizugeben, wird überprüft, ob sich gerade noch zwei Clients im Formular befinden. Ist das der Fall, schickt sich das Formular selbst die Nachricht zum Schließen. (Das Formular wird nicht direkt mit der Methode **Close** geschlossen, da in diesem Fall das Ereignis **OnClose** vor dem Entfernen des vorletzten Clients auftreten würde. Da beim Freigeben mit **Release** das OnClose-Ereignis ausbleibt, kommt auch diese Möglichkeit nicht in Frage.)

Im OnClose-Ereignis wird gegebenenfalls die Docking-Verbindung zum letzten Client aufgehoben und das Formular freigegeben:

```
procedure TfoPlainHost.FormUnDock(Sender: TObject;
  Client: TControl; NewTarget: TWinControl; var Allow: Boolean);
begin
  if Client is TfoClient then
    TfoClient(Client).DockSite:=true;
  if (DockClientCount=2) and (NewTarget<>Self) then
    PostMessage(Handle, WM_CLOSE, 0, 0);
end;

procedure TfoPlainHost.FormClose(Sender: TObject;
  var Action: TCloseAction);
var
  UndockRect: TRect;
begin
  if (DockClientCount=1) then begin
    with DockClients[0] do begin
      UndockRect.TopLeft:=ClientToScreen(Point(0, 0));
      UndockRect.BottomRight:=
             ClientToScreen(Point(UndockWidth, UndockHeight));
      ManualFloat(UndockRect);
    end;
    Action:=caFree;
  end;
end;
```