Entwicklerhandbuch

Borland®

Delphi™ 5

für Windows 95, Windows 98 & Windows NT

Inprise GmbH, Robert-Bosch-Straße 11, D-63225 Langen

Deutsche Ausgabe © 1999 Inprise GmbH, Robert-Bosch-Straße 11, D-63225 Langen, Telefon 06103/979-0, Fax 06103/979-290

Update/Übertragung ins Deutsche: Krieger, Zander & Partner GmbH, München

Satz: Krieger, Zander & Partner GmbH, München

Hauptsitz: 100 Enterprise Way, P.O. Box 660001, Scotts Valley, CA 95067-0001, +1-(408)431-1000

Niederlassungen in: Australien, Deutschland, Frankreich, Großbritannien, Hong Kong, Japan, Kanada, Lateinamerika, Mexiko, den Niederlanden und Taiwan

HDA1350GE21001

Inhalt

Kapitel 8
Multithread-Anwendungen entwickeln 8-1

Kapitel 9
Packages und Komponenten 9-1

Kapitel 13

Ein- und zweischichtige Anwendunen erstellen 13-1

Kapitel 14

Mehrschichtige Anwendungen erstellen 14-1

Kapitel 20

Kapitel 21
Abfragen 21-1

Kapitel 22
Stored Procedures 22-1

Kapitel 25
Zwischengespeicherte Aktualisierungen 25-1

Kapitel 49
Eine Active-Server-Seite erstellen 49-1

Kapitel 50
Mit Typbibliotheken arbeiten 50-1

Index

Kapitel

1

Einführung

Das *Entwicklerhandbuch* enthält Informationen über mittelschwere und fortgeschrittene Programmierthemen. Dazu gehört das Erstellen von Client / Server-Datenbankanwendungen, das Entwickeln von benutzerdefinierten Komponenten, das Erstellen von Web-Server-Anwendungen und das Bereitstellen der Unterstützung von Standardspezifikationen wie CORBA,TCP / IP, MTS, COM und ActiveX. Die Arbeit mit diesem Handbuch setzt voraus, daß Sie mit der Entwicklungsumgebung von Delphi und den elementaren Programmiertechniken vertraut sind. Eine Einführung in die Programmierung mit Delphi und einen Überblick über die IDE finden Sie in der Online-Hilfe.

Inhalt dieses Handbuchs

Dieses Handbuch besteht aus folgenden Teilen:

- **Teil I, »Programmieren mit Delphi«**, enthält Informationen darüber, wie Sie mit Delphi allgemeine Anwendungen erstellen können. Sie finden hier Einzelheiten über Programmiertechniken, die in jeder Delphi-Anwendung zum Einsatz kommen. So wird beispielsweise beschrieben, wie mit Hilfe der allgemeinen VCL-Objekte das Erstellen einer Benutzeroberfläche vereinfacht werden kann (z.B. String-Behandlung, Textbearbeitung, Standard-Dialogfelder, ToolBar- und CoolBar-Komponenten). Dieser Teil enthält auch Kapitel über die Arbeit mit Grafiken, über Fehler- und Ereignisbehandlung, DLLs, OLE-Automatisierung und das Schreiben mehrsprachiger Anwendungen.

  Im Kapitel über das Weitergeben von Anwendungen wird beschrieben, wie Sie Ihre Anwendungen an den Endbenutzer weitergeben können. Es enthält beispielsweise Informationen über Compiler-Optionen, InstallShield Express, die Lizenzierung und das Bestimmen der für die Anwendung benötigten Packages, DLLs und sonstigen Bibliotheken.

- **Teil II, »Datenbankanwendungen entwickeln«,** enthält Informationen, wie mit Hilfe der Datenbank-Tools und -komponenten professionelle Datenbankanwen-

dungen erstellt werden. In Delphi können Sie auf viele verschiedene Datenbanksysteme zugreifen. In Ihren Formularen und Reports können Sie lokale Datenbanken wie Paradox und dBASE, SQL-Server wie InterBase und Sybase und jede mit einem ODBC-Treiber ausgestattete Datenquelle für den Datenzugriff verwenden. Wenn Sie fortgeschrittene Client / Server-Datenbankanwendungen erstellen wollen, benötigen Sie die Client / Server- oder Enterprise-Edition von Delphi.

- **Teil III, »Verteilte Anwendungen entwickeln«,** beschreibt, wie Sie Anwendungen entwickeln, die über ein lokales Netzwerk verteilt werden. Dazu gehören CORBA- und Web-Server-Anwendungen (z.B. CGI-Anwendungen und NSAPI- und ISAPI-DLLs). Außerdem erfahren Sie hier, wie Sie mit Socket-Komponenten arbeiten, die sich bei TCP / IP und ähnlichen Protokollen um die Details der Kommunikation kümmern. Die Komponenten, die CORBA- und Web-Server-Anwendungen unterstützen, sind in den Client / Server- und Enterprise-Editionen von Delphi verfügbar. Die Socket-Komponenten sind auch in der Professional-Edition enthalten.
- Teil IV, »Benutzerdefinierte Komponenten erzeugen«**,** enthält Informationen, wie Sie Ihre eigenen Komponenten entwerfen, implementieren und anschließend in der Komponentenpalette installieren können. Als Komponente kann fast jedes Programmelement verwendet werden, das zur Laufzeit bearbeitet werden soll. Beim Implementieren einer benutzerdefinierten Komponente wird eine neue Klasse von einem bereits in der VCL vorhandenen Typ abgeleitet.
- **Teil V, »COM-Anwendungen entwickeln«**, enthält Informationen, wie Anwendungen erstellt werden können, die mit anderen COM-basierten API-Objekten wie beispielsweise Win95-Shell-Erweiterungen oder Multimedia-Anwendungen interagieren. Delphi stellt Komponenten zur Verfügung, die die COM-basierte ActiveX-Bibliothek für COM-Steuerelemente unterstützen. Diese Steuerelemente können für allgemeine Anwendungen und für Web-Anwendungen verwendet werden. Außerdem erfahren Sie hier, wie Server erstellt werden, die in der MTS-Laufzeitumgebung ausgeführt werden können. MTS bietet erweiterte Laufzeitunterstützung für Sicherheits- und Transaktionsfunktionen und für Ressourcen-Pooling.

  Die COM-Unterstützung ist in allen Editionen von Delphi verfügbar. Wenn Sie ActiveX-Steuerelemente erstellen wollen, benötigen Sie die Professional-, Client / Server- oder Enterprise-Edition. Um MTS-Server zu erstellen, benötigen Sie die Client / Server- oder die Enterprise-Edition.

Typographische Konventionen

Für alle Delphi-Handbücher gelten die nachfolgend aufgeführten typographischen Konventionen.

Tabelle 1.1 Schriftarten und Symbole in den Handbüchern

Schriftart/Symbol	Bedeutung
`Schreibmaschinenschrift`	Text in Schreibmaschinenschrift steht für Delphi Quelltext oder für Text, wie er auf dem Bildschirm erscheint. Weiterhin kann es sich um Text handeln, den Sie in dieser Form eingeben müssen.
[ ]	Eckige Klammern im Text oder in Syntax-Listings kennzeichnen optionale Elemente. Bei der Eingabe dieser Elemente sind die Klammern wegzulassen.
Fettschrift	Fettschrift kennzeichnet reservierte Schlüsselwörter von Delphi oder Compiler-Optionen.
Kursiv	Kursiv geschriebene Wörter verweisen auf Delphi-Bezeichner, wie Variablen, Komponenten, Ereignisse, Methoden und Eigenschaften. Kursivschrift dient auch zur Hervorhebung bestimmter Wörter, etwa neuer Begriffe.
Tasten	Diese Schriftart bezeichnet eine Taste. Beispiel: "Drücken Sie *Esc*, um das Menü zu verlassen."

Inprise Developer Support Services

Um den Bedürfnissen der Delphi-Entwickler zu entsprechen, bietet Inprise eine Reihe zusätzlicher Service-Leistungen an. Informationen über diese Leistungen finden Sie unter
http://www.inprise.com/europe/germany.

Auf dieser Web-Site finden Sie auch weitere technische Informationen zu Delphi sowie Antworten auf häufig gestellte Fragen (FAQs).

Von dieser Web-Site aus haben Sie auch Zugang zu zahlreichen Newsgroups, in denen Delphi-Entwickler Informationen und Tips austauschen. Sie finden hier auch eine Liste mit Büchern über Delphi.

Mehr Informationen erhalten Sie von unserem Kundendienst unter 0130 82 08 66

Gedruckte Dokumentation

Wie Sie die gedruckte Dokumentation bestellen können, erfahren Sie auf der Inprise-Web-Site unter http://www.borland.de.

Teil

I

Programmieren mit Delphi

Die Kapitel im ersten Teil dieses Handbuchs beschreiben die grundlegenden Konzepte der Anwendungsentwicklung mit einer beliebigen Delphi-Edition. Verschiedentlich werden auch die Konzepte angesprochen, die in den nachfolgenden Teilen dieses Handbuchs ausführlich erörtert werden.

Kapitel

2

Object Pascal mit der VCL verwenden

Dieses Kapitel beschreibt den Einsatz von Object Pascal und der Komponentenbibliothek in Delphi-Anwendungen.

Object Pascal und die VCL

Object Pascal, die Programmiersprache von Delphi, erweitert Standard-Pascal um objektorientierte Programmiertechniken. Die VCL (Visual Component Library = Bibliothek visueller Komponenten) ist eine Hierarchie von Object-Pascal-Objekten. Sie ist in die Delphi-IDE integriert und ermöglicht so das rasche Erstellen von Anwendungen. Die Objekte können aus der Komponentenpalette in ein Formular eingefügt werden. Ihre Eigenschaften und Ereignisse können dann mit Hilfe des Objektinspektors bearbeitet werden, ohne daß auch nur eine Zeile Quelltext geschrieben werden muß.

Sämtliche VCL-Objekte sind von *TObject* abgeleitet. Die Methoden dieser abstrakten Klasse kapseln Grundfunktionen wie Erstellen, Freigeben und Botschaftsbehandlung. *TObject* ist der direkte Vorfahr vieler einfacher Klassen.

Die Komponenten in der VCL sind von der abstrakten Klasse *TComponent* abgeleitet. Es handelt sich dabei um Objekte, die während des Entwurfs in Formularen plaziert und bearbeitet werden können. Die visuellen Komponenten, die zur Laufzeit auf dem Bildschirm zu sehen sind (z. B. *TForm* und *TSpeedButton*), heißen Steuerelemente. Sie sind von *TControl* abgeleitet.

Trotz ihres Namens besteht die VCL hauptsächlich aus nichtvisuellen Objekten. Einige dieser Komponenten können ebenfalls aus der Palette in Anwendungen eingefügt werden. So kann beispielsweise durch Plazieren einer *TDataSource*-Komponente in einem Formular oder Datenmodul eine Verbindung mit einer Datenbank hergestellt werden. Das Objekt wird dann im Dokument durch ein Symbol dargestellt, das zur

Laufzeit nicht angezeigt wird. Seine Eigenschaften und Ereignisse können wie bei einem visuellen Steuerelement mit Hilfe des Objektinspektors bearbeitet werden.

Wenn Sie in Object Pascal eigene Klassen erstellen, sollten Sie diese von *TObject* (oder einem Nachfahr) ableiten. Auf diese Weise ist sichergestellt, daß die neue Klasse über die wesentlichen Grundfunktionen verfügt und reibungslos mit der VCL zusammenarbeitet.

Das Objektmodell

Die Objektorientierung ist eine Erweiterung der strukturierten Programmierung. Sie ermöglicht das einfache Wiederverwenden von Quelltext und kapselt Daten mit der entsprechenden Funktionalität. Nachdem Sie ein Objekt (genauer: eine Klasse) erstellt haben, wird es jederzeit in anderen Anwendungen verwendet werden. Auf diese Weise wird die Entwicklungszeit verkürzt und die Produktivität erhöht.

Wie Sie neue Komponenten erstellen und in die Komponentenpalette aufnehmen können wird in Kapitel 31, »Die Komponentenentwicklung im Überblick,«, beschrieben.

Was ist ein Objekt?

Ein Objekt (eine Klasse) ist ein Datentyp, in dem Daten und die Datenoperationen zusammengefaßt sind. Vor der Einführung der objektorientierten Programmierung wurden diese Elemente getrennt definiert.

Das Verständnis von Objekten in Object Pascal wird erleichtert, wenn Sie sich vergegenwärtigen, was ein Record ist. Records (in C Strukturen) bestehen aus Feldern mit Daten. Dabei kann jedes Feld einen anderen Datentyp haben. Deshalb kann mit Hilfe von Records auf unterschiedliche Datenelemente zugegriffen werden.

Objekte sind ebenfalls Sammlungen von Datenelementen. Sie enthalten aber zusätzlich Prozeduren und Funktionen zum Bearbeiten der Daten. Diese Routinen heißen Methoden.

Auf die verschiedenen Datenelemente eines Objekts kann mit Hilfe seiner Eigenschaften zugegriffen werden. Deren Werte können während des Entwurfs jederzeit visuell geändert werden. Um sie zur Laufzeit zu ändern, müssen Sie entsprechenden Quelltext schreiben.

Das Zusammenfassen von Daten und Quelltext nennt man Kapselung. Die beiden anderen Merkmale der objektorientierten Programmierung sind Vererbung und Polymorphie. Vererbung bedeutet, daß Objekte ihre Funktionalität von anderen Objekten (ihren Vorfahren) erben. Das geerbte Verhalten kann aber jederzeit geändert werden. Polymorphie heißt, daß von demselben Vorfahr abgeleitete Objekte dieselben Methoden- und Eigenschaftsschnittstellen unterstützen, die oftmals austauschbar aufgerufen werden können.

Ein Delphi-Objekt untersuchen

Wenn Sie ein neues Projekt erstellen, wird automatisch ein neues Formular geöffnet, das Sie als Ausgangspunkt für die Anwendung verwenden können. Im Quelltexteditor wird ein neuer Klassentyp für das Formular deklariert und Code eingefügt, um die Formularinstanz zu erzeugen. Das folgende Beispiel zeigt den automatisch generierten Quelltext:

```
unit Unit1;
interface

uses Windows, Classes, Graphics, Forms, Controls, Apps;

type
  TForm1 = class(TForm){ Hier beginnt die Typdeklaration des Formulars }
  private
    { Private-Deklarationen }
  public
    { Public-Declarationen }
  end;{ Hier endet die Typdeklaration des Formulars }

var
  Form1: TForm1;

implementation{ Beginn des Implementierungsabschnitts }
{$R *.DFM}
end.{ Ende des Implementierungsabschnitts und der Unit}
```

Der neue Klassentyp *TForm1* ist von der Klasse *TForm* abgeleitet.

Eine Klasse entspricht darin einem Record, daß beide Datenfelder enthalten. Eine Klasse verfügt aber zusätzlich über Methoden (Quelltext), um die Daten zu bearbeiten. *TForm1* enthält bisher keine Felder oder Methoden, weil Sie noch keine Komponenten (die Felder des neuen Objekts) hinzugefügt oder Ereignisbehandlungsroutinen (die Methoden des neuen Objekts) erstellt haben. Es sind aber dennoch Felder und Methoden vorhanden, auch wenn sie nicht in der Typdeklaration aufgeführt sind.

In der folgenden Anweisung wird die Variable *Form1* des neuen Typs *TForm1* deklariert:

```
var
   Form1: TForm1;
```

Form1 ist eine Instanz oder ein Objekt des Klassentyps *TForm1*. Es können auch mehrere Instanzen eines Typs deklariert werden, beispielsweise für die untergeordneten Fenster einer MDI-Anwendung (Multiple Document Interface). Jede Instanz verfügt dabei über eigene Datenfelder, die Methoden werden jedoch gemeinsam verwendet.

Obwohl Sie bis jetzt noch keine Komponente in das Formular eingefügt und noch keine einzige Zeile Quelltext geschrieben haben, ist bereits ein vollständiges Delphi-Programm entstanden, das Sie kompilieren und starten können. Es zeigt aber lediglich ein leeres Formular an.

Wir werden daher eine Schaltfläche hinzufügen und eine Behandlungsroutine für das Ereignis *OnClick* erstellen, mit der die Hintergrundfarbe des Formulars geändert wird. Das Ergebnis sieht dann folgendermaßen aus:

Abbildung 2.1 Ein einfaches Formular

Sobald der Benutzer auf die Schaltfläche klickt, wird die Farbe des Formulars in Grün geändert. Dies wird mit Hilfe der folgenden Behandlungsroutine für das Ereignis *OnClick* der Schaltfläche erreicht:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Form1.Color := clGreen;
end;
```

Als Datenfelder können auch andere Objekte verwendet werden. Für jede in ein Formular eingefügte Komponente wird ein neues Feld in die Formulardeklaration aufgenommen. Wenn Sie die Anwendung erstellen, enthält der Quelltexteditor folgendes:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls, Apps;
type
   TForm1 = class(TForm)
    Button1: TButton;{ Neues Datenfeld }
    procedure Button1Click(Sender: TObject);{ Neue Methodendeklaration }
   private
    { Private-Deklarationen }
   public
    { Public-Deklarationen }
   end;
var
   Form1: TForm1;
implementation

{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject); { Der Quelltext der neuen Methode }
begin
   Form1.Color := clGreen;
end;
end.
```

TForm1 verfügt über das Feld *Button1*, das der hinzugefügten Schaltfläche entspricht. *TButton* ist ein Klassentyp, und daher verweist *Button1* auf ein Objekt.

Alle in Delphi erstellten Ereignisbehandlungsroutinen sind Methoden des Formularobjekts. Daher wird für jede Routine eine neue Methodendeklaration in das Formularobjekt eingefügt. Die Deklaration von *TForm1* enthält nun eine Methode, die Prozedur *Button1Click*. Der eigentliche Quelltext der Methode befindet sich im **implementation**-Abschnitt der Unit.

Den Namen einer Komponente ändern

Ändern Sie den Namen einer Komponente immer mit dem Objektinspektor. Ein Beispiel: Sie möchten den Standardnamen *Form1* in den aussagekräftigeren Namen *ColorBox* ändern. Wenn Sie nun die Formulareigenschaft Name im Objektinspektor ändern, werden alle Verweise auf das Formular in der DFM-Datei des Formulars (die normalerweise nicht manuell bearbeitet wird) und in dem von Delphi generierten Quelltext automatisch aktualisiert:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls, Apps;
type
   TColorBox = class(TForm){ Von TForm1 in TColorBox geändert }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
   private
    { Private-Deklarationen }
   public
    { Public-Deklarationen }
    end;
var
   ColorBox: TColorBox;{ Von Form1 in ColorBox geändert }
implementation

{$R *.DFM}
procedure TColorBox.Button1Click(Sender: TObject);
begin
   Form1.Color := clGreen;{ Der Verweis auf Form1 wurde nicht geändert! }
end;
end.
```

Beachten Sie, daß die *OnClick*-Ereignisbehandlungsroutine nicht aktualisiert wurde. In dem von Ihnen geschriebenen Quelltext müssen Sie die Änderungen selbst vornehmen:

```
procedure TColorBox.Button1Click(Sender: TObject);
begin
  ColorBox.Color := clGreen;
end;
```

Daten und Quelltext von einem Objekt erben

Das auf Seite 2-3 im Abschnitt »Ein Delphi-Objekt untersuchen« beschriebene Objekt *TForm1* ist auf den ersten Blick sehr einfach. Es enthält lediglich ein Feld (*Button1*), eine Methode (*Button1Click*) und keine Eigenschaften. Dennoch können Sie es anzeigen, schließen oder seine Größe ändern, Standardschaltflächen zum Minimieren und

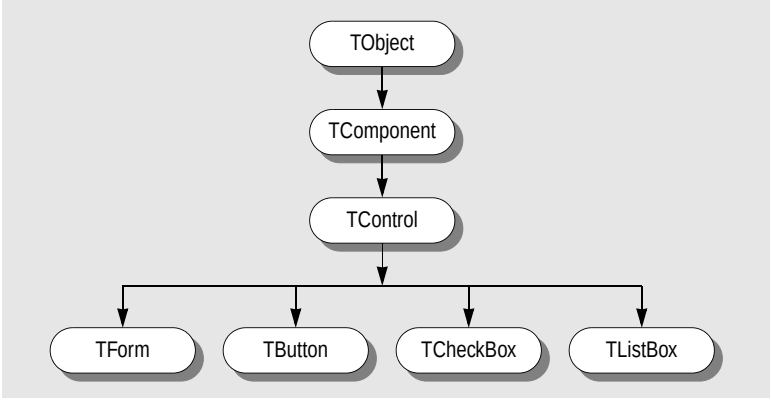
Maximieren in der Titelleiste anzeigen und es zu einem Fenster einer MDI-Anwendung machen. Dies ist möglich, da es alle Eigenschaften und Methoden der VCL-Komponente *TForm* erbt. Wenn Sie ein neues Formular in Ihr Projekt einfügen, wird automatisch ein neues Objekt von TForm abgeleitet, das Sie anschließend durch Hinzufügen von Komponenten, Ändern von Eigenschaftswerten und Schreiben von Ereignisbehandlungsroutinen Ihren Wünschen entsprechend anpassen können. Verwenden Sie diese Vorgehensweise für alle Objekte. Leiten Sie das neue Objekt von einem bereits vorhandenen ab, und nehmen Sie danach die gewünschten Änderungen vor. Beim Hinzufügen eines Formulars wird folgende Anweisung in den Quelltext aufgenommen:

```
TForm1 = class(TForm)
```

Ein abgeleitetes Objekt (der Nachkomme) erbt alle Eigenschaften, Ereignisse und Methoden der Klasse, aus der es abgeleitet wird (seines Vorfahren). In der Online-Hilfe finden Sie unter *TForm* eine Liste der Eigenschaften, Ereignisse und Methoden dieser Klasse, einschließlich der von ihren Vorfahren geerbten Elemente. Ein Objekt kann nur einen direkten Vorfahren, aber beliebig viele direkte Nachkommen haben.

Objekte, Komponenten und Steuerelemente

Abbildung 2.2 Vereinfachte VCL-Hierarchie



Die Abbildung zeigt eine stark vereinfachte Darstellung der Vererbungshierarchie in der VCL. Jedes Objekt erbt von *TObject*. Die Komponenten in der Hierarchie erben von *TComponent*. Die von *TControl* abgeleiteten Objekte heißen Steuerelemente und können zur Laufzeit angezeigt werden. So erbt beispielsweise *TCheckBox* alle Elemente der Klassen *TObject*, *TComponent* und *TControl* und verfügt über spezielle eigene Fähigkeiten.

Gültigkeitsbereich und Qualifizierer

Der Gültigkeitsbereich steuert den Zugriff auf die Felder, Eigenschaften und Methoden eines Objekts. Alle in einem Objekt deklarierten Elemente sind für dieses und seine Nachkommen verfügbar. Bei Methoden ist die Position ihrer Deklaration ent-

scheidend. Auch wenn sich der Quelltext (die Implementierung) außerhalb der Typdeklaration befindet, liegt die betreffende Methode dennoch im Gültigkeitsbereich des Objekts, in dem sie definiert ist.

Wenn Sie in einer Ereignisbehandlungsroutine eines Objekts auf dessen Eigenschaften, Methoden oder Felder zugreifen, müssen Sie die Objektvariable nicht vor dem Elementbezeichner angeben. Das folgende Beispiel zeigt die *OnClick*-Ereignisbehandlungsroutine einer Schaltfläche, die sich in einem Formular befindet:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Color := clFuchsia;
   Button1.Color := clLime;
end;
```

Die erste Anweisung ist mit folgender Zeile identisch:

```
Form1.Color := clFuchsia
```

Der Qualifizierer *Form1* muß nicht für die Eigenschaft *Color* verwendet werden, da sich die Methode *Button1Click* im Gültigkeitsbereich des Objekts *TForm1* befindet. Somit sind alle Bezeichner in der Methode für die Instanz von *TForm1* verfügbar, in der die Routine aufgerufen wird. Mit der zweiten Anweisung wird die Farbe der Schaltfläche (nicht des Formulars, in dem die Routine deklariert ist) geändert, und daher muß das Objekt angegeben werden.

Für jedes Formular wird automatisch eine eigene Unit (Quelltextdatei) erstellt. Um auf die Komponenten in einem bestimmten Formular in der Unit-Datei eines anderen Formulars zuzugreifen, müssen die Objektnamen in der folgenden Form qualifiziert werden:

```
Form2.Edit1.Color := clLime;
```

Auf die gleiche Weise können Methoden einer Komponente in einem anderen Formular aufgerufen werden:

```
Form2.Edit1.Clear;
```

Die Voraussetzung für den Zugriff auf die Komponenten von *Form2* ist aber, daß Sie dessen Quelltextdatei (Unit) in die **uses**-Klausel der Unit von *Form1* aufnehmen.

Der Gültigkeitsbereich eines Objekts erstreckt sich auch über seine Nachkommen. Sie können jedoch in den abgeleiteten Objekten jederzeit ein Feld, eine Eigenschaft oder eine Methode erneut deklarieren und dadurch das geerbte Element verdecken oder überschreiben.

Weitere Informationen zu Gültigkeitsbereich, Vererbung und **uses**-Klausel finden Sie in der *Object Pascal Sprachreferenz*.

private-, protected-, public- und published-Deklarationen

Bei der Deklaration eines Feldes, einer Eigenschaft oder einer Methode kann die Sichtbarkeit mit den Schlüsselwörtern **private**, **protected**, **public** und **published** angegeben werden. Sie bestimmt, ob auf das betreffende Element in anderen Objekten und Units zugegriffen werden kann.

- Auf **private**-Elemente kann nur in der Unit zugegriffen werden, in der sie deklariert sind. Sie werden häufig in einer Klasse verwendet, um andere Methoden (**public** oder **published**) und Eigenschaften zu implementieren.
- Auf **protected**-Elemente kann in der Unit, in der ihre Klasse deklariert ist, sowie in allen abgeleiteten Klassen unabhängig von deren Unit zugegriffen werden.
- Auf **public**-Elemente kann überall zugegriffen werden, wo das Objekt sichtbar ist, zu dem sie gehören (d. h. in der Unit, in der die Klasse deklariert ist, und in allen Units, in der die Datei per **uses**-Klausel eingebunden ist).
- Die **published**-Elemente haben dieselbe Sichtbarkeit wie **public**-Elemente, für sie werden aber vom Compiler Laufzeit-Typinformationen generiert. Auf sie kann während des Entwurfs im Objektinspektor zugegriffen werden.

Weitere Informationen zur Sichtbarkeit finden Sie in der *Object Pascal Sprachreferenz*.

Objektvariablen verwenden

Sie können Zuweisungen zwischen Objektvariablen durchführen, die denselben Typ haben oder zuweisungskompatibel sind. Dies ist insbesondere möglich, wenn die Variable, der ein Wert zugewiesen wird, ein Vorfahr des zuzuweisenden Typs ist. Im folgenden Beispiel werden der Typ *TDataForm* und die beiden Variablen *AForm* und *DataForm* deklariert:

```
type
  TDataForm = class(TForm)
  Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Database1: TDatabase;
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  AForm: TForm;
  DataForm: TDataForm;
```

AForm hat den Typ *TForm* und *DataForm* den Typ *TDataForm*. *TDataForm* ist von *TForm* abgeleitet, und daher ist folgende Zuweisung zulässig:

```
AForm := DataForm;
```

Nehmen wir an, Sie erstellen eine Behandlungsroutine für das Ereignis *OnClick* einer Schaltfläche. Die Routine wird dann zur Laufzeit beim Klicken auf die Schaltfläche aufgerufen. Jeder Ereignisbehandlungsroutine wird der Parameter Sender des Typs *TObject* übergeben:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
⋮
end;
```

Sender hat den Typ *Tobject*. Daher können dem Parameter beliebige Objekte zugewiesen werden. Es wird immer das Steuerelement oder die Komponente übergeben, das bzw. die auf das Ereignis reagiert. Sie können wie im folgenden Beispiel mit Hilfe des reservierten Wortes **is** ermitteln, welchen Objekttyp Sender enthält:

```
if Sender is TEdit then
   DoSomething
else
   DoSomethingElse;
```

Objekte erstellen, instantiieren und freigeben

Die meisten Objekte in Delphi-Anwendungen sind Steuerelemente wie Eingabefelder und Schaltflächen, die zur Entwurfs- und zur Laufzeit angezeigt werden. Manche von ihnen (z. B. die Standarddialogfelder) werden nur in der Anwendung, nicht aber im Entwurfsfenster angezeigt. Wieder andere, wie Zeitgeber und Datenquellen, werden zur Laufzeit nicht angezeigt, können aber in der Entwurfsumgebung visuell bearbeitet werden.

Sie können natürlich auch eigene nichtvisuelle Komponenten erstellen, wie beispielsweise ein *TEmployee*-Objekt mit den Eigenschaften *Name*, *Title* und *HourlyPayRate*. Sie können dann die Methode *CalculatePay* hinzufügen, um aus den Daten im Feld *HourlyPayRate* den Gesamtlohn für einen bestimmten Zeitraum zu berechnen. Die Typdeklaration lautet beispielsweise folgendermaßen:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Real;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Real read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Real;
  end;
```

Zusätzlich zu den hier definierten Feldern, Eigenschaften und Methoden erbt *TEmployee* alle Methoden von *TObject*. Sie können eine Typdeklaration wie diese in den **interface**- oder **implemetation**-Abschnitt einer Unit aufnehmen und dann mit Hilfe der von *TObject* geerbten Methode *Create* Instanzen der neuen Klasse erstellen:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

Die Methode *Create* wird *Konstruktor* genannt. Sie weist Speicher für ein neues Instanzobjekt zu und gibt einen Verweis auf dieses zurück.

Die Komponenten in einem Formular werden automatisch erstellt und freigegeben. Wenn Sie aber Objekte in selbstgeschriebenem Quelltext instantiieren, müssen Sie

diese explizit wieder freigeben. Jedes Objekt erbt die Methode Destroy (einen Destruktor) von *TObject*. Zum manuellen Freigeben eines Objekts ist aber die (ebenfalls von *TObject* geerbte) Methode *Free* vorzuziehen. Sie prüft zuerst, ob die Instanz noch vorhanden ist und ruft nur dann *Destroy* auf. Mit der folgenden Anweisung wird das Employee-Objekt freigegeben:

```
Employee.Free
```

Komponenten und Eigentümer

Delphi verfügt über eine integrierte Speicherverwaltung, durch die eine Komponente die Verantwortung für das Freigeben einer anderen übernehmen kann. Erstere bezeichnet man dabei als Eigentümer der letzteren. Diese wird automatisch freigegeben, wenn ihr Eigentümer aus dem Speicher entfernt wird. Der Eigentümer einer Komponente (der Wert ihrer Eigenschaft *Owner*) wird durch einen Parameter bestimmt, der bei ihrer Erstellung an den Konstruktor übergeben wird. Standardmäßig ist ein Formular Eigentümer seiner Komponenten und die Anwendung Eigentümer des Formulars. Daher werden beim Beenden der Anwendung automatisch alle Formulare und Komponenten freigegeben.

Beachten Sie aber, daß dieser Mechanismus nur *TComponent* und abgeleitete Klassen betrifft. Wenn Sie beispielsweise ein Objekt des Typs *TStringList* oder *TCollection* erstellen, müssen Sie es anschließend explizit freigeben, auch wenn es einem Formular zugeordnet ist.

Hinweis Verwechseln Sie den Eigentümer einer Komponente nicht mit ihrem übergeordneten Objekt. Weitere Informationen finden Sie unter »Parent-Eigenschaften« auf Seite 2-12.

Komponenten verwenden

Alle Komponenten verfügen über die von *TComponent* geerbten Merkmale. Mit ihnen können Sie die Benutzeroberfläche und die Funktionsweise Ihrer Anwendung erstellen. Die mit Delphi gelieferten Standardkomponenten reichen für die meisten Anforderungen aus, Sie können aber die VCL jederzeit um eigene Komponenten erweitern. Weitere Informationen über das Erstellen eigener Komponenten finden Sie in Kapitel 31, »Die Komponentenentwicklung im Überblick«.

Die Delphi-Standardkomponenten

Die Komponentenpalette enthält eine große Auswahl von Komponenten für verschiedene Programmieraufgaben. Sie können jederzeit neue Komponenten hinzufügen, vorhandene Komponenten entfernen oder die Komponenten in der Palette neu anordnen. Sie können auch Komponentenvorlagen und Rahmen erstellen, um mehrere Komponenten zu gruppieren.

Die Komponenten in der Palette sind entsprechend ihrer Funktion in verschiedene Registerkarten aufgeteilt. Die Registerkarten der Standardkonfiguration sind von der

Delphi-Version abhängig. Tabelle 2.1 enthält die Standardregisterkarten und die enthaltenen Objekte.

Tabelle 2.1 Registerkarten der Komponentenpalette

Registerkarte	Inhalt
Standard	Windows-Standardkomponenten, Menüs.
Zusätzlich	Weitere Steuerelemente.
Win32	Win32-Standardkomponenten (Windows 9x / NT 4.0).
System	Komponenten und Steuerelemente für den Zugriff auf Systemfunktionen (z. B. Zeitgeber, Multimedia und DDE).
Internet	Komponenten für Internet-Kommunikationsprotokolle und Web-Anwendungen.
Datenzugriff	Nichtvisuelle Komponenten für den Zugriff auf Tabellen, Abfragen und Reports.
Datensteuerung	Visuelle datensensitive Steuerelemente.
Datenanalyse	Steuerelemente, mit denen Informationen aus Datenbanken ausgewertet und unter Berücksichtigung verschiedener Gesichtspunkte angezeigt werden können.
QReport	QuickReport-Komponenten, mit denen auf einfache Weise eingebettete Reports erstellt werden können.
Dialoge	Windows-Standarddialogfelder.
Win 3.1	Komponenten für die Abwärtskompatibilität mit Delphi 1.0.
Beispiele	Verschiedene Beispielkomponenten.
ActiveX	ActiveX-Beispielkomponenten.
Midas	Komponenten für das Erstellen mehrschichtiger Datenbankanwendungen.

Informationen zu den einzelnen Komponenten finden Sie in der Online-Hilfe. Die Steuerelemente der Registerkarten *ActiveX* und *Beispiele* dienen jedoch nur als Beispiele und sind nicht dokumentiert.

Eigenschaften aller visuellen Komponenten

Alle visuellen Komponenten (Nachkommen von *TControl*) verfügen über folgende Eigenschaften:

- Positions- und Größeneigenschaften
- Anzeigeeigenschaften
- Parent-Eigenschaften
- Navigationseigenschaften
- Drag&Drop-Eigenschaften
- Drag&Dock-Eigenschaften

Obwohl diese Eigenschaften (**published**-Datenfelder) von *TControl* geerbt sind, werden sie nur für Komponenten im Objektinspektor angezeigt, für die sie verwendet werden können. So ist beispielsweise die Eigenschaft *Color* bei *TImage*-Objekten nicht vorhanden, da die Farbe durch die angezeigte Grafik bestimmt wird.

Positions- und Größeneigenschaften

Die Größe und Position einer Komponente in einem Formular kann mit den folgenden vier Eigenschaften festgelegt werden:

- *Height* gibt die Höhe der Komponente an.
- *Width* gibt die Breite der Komponente an.
- *Top* gibt die Position des oberen Komponentenrandes an.
- *Left* gibt die Position des linken Komponentenrandes an.

Diese Eigenschaften können für nichtvisuelle Komponenten nicht verwendet werden. Delphi verwaltet jedoch die Position der Komponentensymbole im Formular. Positions- und Größeneigenschaften werden meistens visuell im Formular, im Objektinspektor oder mit Hilfe der Ausrichtungspalette festgelegt. Sie können aber auch zur Laufzeit geändert werden.

Anzeigeeigenschaften

Die folgenden vier Eigenschaften legen das allgemeine Erscheinungsbild eines Steuerelements fest:

- *BorderStyle* gibt an, ob das Steuerelement mit einem Rahmen angezeigt wird.
- *Color* bestimmt die Hintergrundfarbe des Steuerelements.
- *Ctrl3D* gibt an, ob das Steuerelement räumlich oder flach angezeigt wird.
- *Font* legt Farbe, Schrift, Schnitt und Größe des Textes fest.

Parent-Eigenschaften

Sie erhalten Anwendungen mit konsistentem Erscheinungsbild, indem Sie festlegen, daß jedes Steuerelement die Anzeigeeinstellungen seines Containers (seines übergeordneten Objekts) übernimmt. Weisen Sie dazu einfach den mit Parent beginnenden Eigenschaften den Wert *True* zu. Wenn Sie beispielsweise eine Schaltfläche in ein Formular einfügen und seiner Eigenschaft *ParentFont* den Wert *True* zuweisen, werden Änderungen der Formulareigenschaft Font automatisch an die Schaltfläche (und die anderen untergeordneten Objekte des Formulars) weitergegeben. Weisen Sie später der Eigenschaft *Font* der Schaltfläche einen Wert zu, wird dieser verwendet, und die Eigenschaft *ParentFont* wird auf *False* zurückgesetzt.

Hinweis Verwechseln Sie das übergeordnete Objekt einer Komponente nicht mit ihrem Eigentümer. Weitere Informationen finden Sie unter »Komponenten und Eigentümer« auf Seite 2-10.

Navigationseigenschaften

Mit den folgenden Eigenschaften können Sie festlegen, wie die Benutzer zu den verschiedenen Steuerelementen in einem Formular gelangen:

- *Caption* enthält die Beschriftung der Komponente. Sie können ein bestimmtes Zeichen als Direktzugriffstaste definieren, indem Sie ihm ein kaufmännisches Und (&) voranstellen. Es wird dann unterstrichen angezeigt. Zur Laufzeit kann das

Steuerelement oder die Menüoption direkt ausgewählt werden, indem die Taste *Alt* zusammen mit diesem Zeichen gedrückt wird.

- *TabOrder* gibt die Position des Steuerelements in der Tabulatorreihenfolge (die Reihenfolge, in der die Steuerelemente mit der Taste *Tab* aktiviert werden) seines Containers an. Anfänglich wird automatisch die Reihenfolge verwendet, in der die Komponenten in das Formular eingefügt wurden. Sie kann aber jederzeit geändert werden. Der Wert von *TabOrder* wird nur berücksichtigt, wenn *TabStop* auf *True* gesetzt ist.
- *TabStop* bestimmt, ob das Steuerelement mit der Taste *Tab* aktiviert werden kann. Hat die Eigenschaft den Wert *True*, wird das Steuerelement in die Tabulatorreihenfolge aufgenommen.

Drag&Drop-Eigenschaften

Mit den folgenden beiden Eigenschaften kann das Drag&Drop-Verhalten der Komponente gesteuert werden:

- *DragMode* bestimmt, wie die Ziehen-Operation gestartet wird. Bei der Standardeinstellung *dmManual* muß die Operation explizit durch einen Aufruf der Methode *BeginDrag* gestartet werden. Wenn *DragMode* den Wert *dmAutomatic* hat, beginnt die Operation automatisch mit dem Drücken der Maustaste.
- *DragCursor* legt fest, welche Form der Mauszeiger über einer Komponente annimmt, die das gezogene Objekt aufnehmen kann.

Drag&Dock-Eigenschaften

Mit den folgenden Eigenschaften kann das Drag&Dock-Verhalten der Komponente gesteuert werden.

- *DockSite*
- *DragKind*
- *DragMode*
- *FloatingDockSiteClass*
- *AutoSize*

Weitere Informationen zu diesen Eigenschaften finden Sie im Abschnitt »Drag&Dock in Steuerelementen implementieren« auf Seite 6-4.

Textkomponenten

In vielen Anwendungen müssen Texteingaben und Textausgaben durchgeführt werden. Je nach Umfang und Format der Informationen kann dazu eines der drei folgenden Steuerelemente verwendet werden.

Komponente:	Einsatzbereich:
Eingabefeld	Eingeben von einzeiligem Text.
Memokomponente	Eingeben von mehrzeiligem Text.

Komponente:	Einsatzbereich:
Maskenfeld	Eingeben von Text in einem bestimmten Format (z. B. Postleitzahl oder Telefonnummer).
RTF-Komponente	Eingeben von mehrzeiligem Text im RTF-Format.

Eigenschaften aller Textkomponenten

Alle Textkomponenten verfügen über folgende Eigenschaften:

- *Text* enthält den Text der Komponente.
- *CharCase* steuert die Groß-/Kleinschreibung des Textes (Großschreibung, gemischte Schreibweise oder Kleinschreibung).
- *ReadOnly* gibt an, ob der Text bearbeitet werden kann.
- *MaxLength* schränkt die Anzahl der Zeichen in der Komponente ein.
- *PasswordChar* ermöglicht das Anzeigen eines speziellen Zeichens (normalerweise ein Sternchen) anstelle des eingegebenen Textes.
- *HideSelection* bestimmt, ob die aktuelle Textauswahl auch angezeigt wird, wenn das Steuerelement nicht mehr den Eingabefokus hat.

Eigenschaften von Memo- und RTF-Komponenten

Die Memo- und RTF-Komponenten für die mehrzeilige Dateneingabe verfügen über folgende Eigenschaften:

- *Alignment* steuert die Textausrichtung (linksbündig, rechtsbündig oder zentriert) in der Komponente.
- *Text* enthält den Text des Steuerelements. Mit Hilfe der Eigenschaft *Modified* kann geprüft werden, ob der Text bearbeitet wurde.
- *Lines* enthält den Text als Liste von Strings.
- *OEMConvert* bestimmt, ob der Text bei der Eingabe von ANSI- in OEM-Zeichen konvertiert wird. Dies ist beispielsweise beim Validieren von Dateinamen hilfreich.
- *WordWrap* bestimmt, ob der Text am rechten Rand der Komponente umbrochen wird.
- *WantReturns* legt fest, ob Zeilenumbrüche in den Text eingefügt werden können.
- *WantTabs* bestimmt, ob Tabulatoren in den Text eingefügt werden können.
- *AutoSelect* gibt an, ob der Text automatisch markiert wird, wenn das Steuerelement den Fokus erhält.
- *SelText* enthält den aktuell markierten Textblock.
- *SelStart* und *SelLength* geben die Position und Länge des markierten Textblocks an.

Zur Laufzeit kann der gesamte Inhalt der Memokomponente mit Hilfe der Methode *SelectAll* markiert werden.

RTF-Komponenten

Die RTF-Komponente ist ein Memofeld, das die RTF-Formatierung unterstützt sowie Druck-, Such- und Drag&Drop-Operationen im Text ermöglicht. Sie können verschiedene Schrift-, Ausrichtungs-, Tabulator-, Einzugs- und Numerierungseinstellungen angeben.

Spezialisierte Eingabekomponenten

Mit den folgenden Komponenten können Dateneingaben auf andere Weise vorgenommen werden.

Komponente:	Einsatzbereich:
Bildlaufleiste (*TScrollBar)*	Auswählen von Werten in einem fortlaufenden Bereich.
Schieberegler (*TTrackBar*)	Auswählen von Werten in einem fortlaufenden Bereich (ermöglicht eine genauere Wertangabe als eine Bildlaufleiste).
Drehfeld (*TUpDown*)	Angeben eines Wertes mit Hilfe einer Kombination aus Wippregler und Eingabefeld.
Tastenkürzel (*THotKey*)	Eingeben von *Strg/Umschalt/Alt*-Tastenfolgen.

Bildlaufleiste (TScrollBar)

Die Komponente *TScrollBar* ist eine Windows-Standardbildlaufleiste. Mit ihr kann der aktuelle Bildausschnitt eines Fensters, Formulars oder Steuerelements verschoben werden. In der Ereignisbehandlungsroutine für *OnScroll* können die gewünschten Aktionen ausgeführt werden, sobald der Benutzer die Bildlaufleiste verwendet.

Die Komponente wird nur selten benötigt, da viele visuelle Steuerelemente über integrierte Bildlauffunktionen verfügen. So können in einem Formular (*TForm*) beispielsweise mit den Eigenschaften *VertScrollBar* und *HorzScrollBar* Bildlaufleisten angezeigt werden. Um einen Bildlaufbereich in einem Formular zu erstellen, verwenden Sie die Komponente *TScrollBox*.

Schieberegler (TTrackBar)

Mit einem Schieberegler können Integer-Werte in einem durchgehenden Bereich angegeben werden (z. B. zum Einstellen von Lautstärke oder Helligkeit). Wertänderungen können durch Ziehen des Positionszeigers oder Klicken auf die Leiste vorgenommen werden.

- Mit den Eigenschaften *Max* und *Min* können Sie den Wertebereich des Reglers festlegen.
- Die Eigenschaften *SelEnd* und *SelStart* ermöglichen das Auswählen eines bestimmten Wertebereichs (siehe Abbildung 2.3).
- Mit Hilfe der Eigenschaft *Orientation* kann die Ausrichtung des Schiebereglers (horizontal oder vertikal) festgelegt werden.
- Standardmäßig befinden sich die Skalenstriche am unteren Rand des Steuerelements. Sie können diese Einstellung jederzeit mit der Eigenschaft *TickMarks* ändern. Mit Hilfe der Eigenschaft *TickStyle* oder der Methode *SetTicks* können Sie den Abstand zwischen den Markierungen festlegen.

Abbildung 2.3 Drei Ansichten der Schiebereglerkomponente

Die Eigenschaft *Position* ermöglicht den Zugriff auf die Position des Schiebereglers.

Standardmäßig wird der Positionszeiger beim Drücken der Taste *Auf* oder *Ab* um einen Markierungsstrich verschoben. Dieser Wert kann mit der Eigenschaft *LineSize* geändert werden.

Mit Hilfe der Eigenschaft *PageSize* können Sie festlegen, um wie viele Markierungsstriche der Positionszeiger verschoben wird, wenn der Benutzer die Taste *Bild auf* oder *Bild ab* drückt.

Drehfeld (TUpDown)

Ein Drehfeld besteht aus zwei Pfeilschaltflächen, mit denen eine Integer-Zahl um einen bestimmten Wert erhöht oder verringert werden kann. Der aktuelle Wert wird mit der Eigenschaft *Position*, die Wertänderung (Standardeinstellung 1) mit der Eigenschaft *Increment* angegeben. Mit Hilfe von *Associate* kann dem Drehfeld eine andere Komponente (z. B. ein Eingabefeld) zugeordnet werden.

Tastenkürzelkomponente (THotKey)

Mit Hilfe der Komponente *THotKey* können Tastenkürzel für beliebige Steuerelemente definiert werden. Die Eigenschaft *HotKey* enthält die aktuelle Tastenkombination, und *Modifiers* bestimmt, welche Modifizierer (*Alt*, *Strg*, *Umschalt*) für *HotKey* zur Verfügung stehen.

Teilerleiste (TSplitter)

Wird eine Teilerleiste (*TSplitter*) zwischen zwei aneinander ausgerichteten Steuerelementen eingefügt, kann deren Größe zur Laufzeit geändert werden. Teilerleisten werden zusammen mit Steuerelementen wie Tafeln und Gruppenfeldern verwendet und ermöglichen das Unterteilen eines Formulars in mehrere Bereiche mit jeweils einem oder mehreren Komponenten.

Fügen Sie jeweils zunächst das gewünschte Steuerelement ein und dann die Teilerleiste mit identischer Ausrichtung hinzu. Weisen Sie dem letzten Steuerelement die Ausrichtung *alClient* zu, damit es automatisch den verbleibenden Raum ausfüllt, wenn die Größe der anderen Bereiche geändert wird. Sie können beispielsweise eine Tafel am linken Formularrand hinzufügen, seine Eigenschaft *Alignment* auf *alLeft* setzen, dann rechts daneben eine Teilerleiste (ebenfalls mit der Ausrichtung *alLeft*) plazieren und zum Schluß rechts von der Leiste eine weitere Tafel mit der Ausrichtung *alLeft* oder *alClient* in das Formular aufnehmen.

Mit der Eigenschaft *MinSize* können Sie festlegen, welchen Bereich die Teilerleiste bei Größenänderungen des benachbarten Steuerelements mindestens für das verkleinerte Steuerelement bereitstellen soll. Sie können der Komponente auch ein dreidimensionales Aussehen geben. Weisen Sie dazu *Beveled* den Wert *True* zu.

Schaltflächen und ähnliche Steuerelemente

Neben Menüs sind Schaltflächen die am häufigsten verwendeten Elemente, um in einer Anwendung Befehle aufzurufen. In Delphi stehen folgende schalterähnliche Steuerelemente zur Verfügung:

Komponente:	Verwendung:
Schaltfläche (*TButton)*	Anzeigen einer Befehlsauswahl.
Bitmap-Schaltfläche (*TBitBtn*)	Anzeigen einer Befehlsauswahl (im Gegensatz zu *Tbutton* können Grafiken verwendet werden).
SpeedButton (*TSpeedButton*)	Erstellen gruppierter Schaltflächen für Symbolleisten.
Kontrollkästchen (*TCheckBox*)	Anzeigen von Booleschen Optionen.
Optionsfeld (*TRadioButton*)	Anzeigen von sich gegenseitig ausschließenden Optionen.
Symbolleiste (*TToolBar*)	Anzeigen von ToolButton-Objekten und anderen Steuerelementen in einer Leiste (Größe und Position der Objekt wird automatisch angepaßt).
CoolBar (*TCoolBar*)	Anzeigen von fensterorientierten Steuerelementen in getrennten Abschnitten, deren Größe oder Position zur Laufzeit geändert werden kann.

Schaltfläche (TButton)

Der Benutzer kann durch Klicken auf eine Schaltfläche eine bestimmte Aktion ausführen. Wenn Sie in der Entwurfsumgebung auf eine Schaltfläche doppelklicken, wird automatisch der Quelltexteditor mit der Behandlungsroutine für das Ereignis *OnClick* geöffnet.

- Setzen Sie die Eigenschaft *Cancel* auf *True*, wenn das Ereignis *OnClick* der Schaltfläche durch die Taste *Esc* ausgelöst werden soll.
- Setzen Sie die Eigenschaft *Default* auf *True*, wenn das Ereignis *OnClick* der Schaltfläche durch die Taste *Return* ausgelöst werden soll.

Bitmap-Schaltfläche (TBitBtn)

In einer Bitmap-Schaltfläche (*BitBtn*) kann zusätzlich zur Beschriftung eine Grafik angezeigt werden.

- Geben Sie mit *Glyph* die gewünschte Grafik für die Schaltfläche an.
- Mit Hilfe von *Kind* kann der Schaltfläche eine vordefinierte Grafik und Funktion zugewiesen werden.
- Standardmäßig wird die Grafik links neben der Beschriftung angezeigt. Die Eigenschaft *Layout* ermöglicht das Ändern dieser Anordnung.
- Grafik und Text werden automatisch in der Schaltfläche zentriert. Sie können die Position ändern, indem Sie mit *Margin* einen Rand vor der Grafik einfügen. Die Angabe erfolgt in Pixel.

- Grafik und Text werden standardmäßig durch 4 Pixel voneinander getrennt. Sie können diesen Abstand mit Hilfe der Eigenschaft *Spacing* ändern.
- Bitmap-Schaltflächen können drei Modi haben: Nicht gedrückt, Gedrückt und Eingerastet. Soll für jeden Modus eine eigene Grafik angezeigt werden, setzen Sie *NumGlyphs* auf 3.

SpeedButton-Komponente (TSpeedButton)

SpeedButton-Objekte sind spezielle Schaltflächen (normalerweise mit einer Grafik), die als Gruppen verwendet werden können. Sie werden häufig mit einer Tafel (*TPanel*) zur Erstellung von Symbolleisten eingesetzt.

- Sie können mehrere Komponenten als Gruppe definieren, indem Sie der Eigenschaft *GroupIndex* der gewünschten Schaltflächen denselben Integer-Wert (ungleich Null) zuweisen.
- Ein SpeedButton-Objekt wird standardmäßig im nicht ausgewählten Status angezeigt. Um es anfangs als ausgewählt anzuzeigen, setzen Sie seine Eigenschaft *Down* auf True.
- Hat *AllowAllUp* den Wert *True*, können alle Schaltflächen der Gruppe nicht ausgewählt sein. Um die Komponenten als Optionsfeldgruppe zu verwenden, setzen Sie *AllowAllUp* auf *False*.

Kontrollkästchen (TCheckBox)

Ein Kontrollkästchen ist ein Umschalter, der dem Benutzer das Auswählen von zwei (manchmal auch drei) Optionen ermöglicht.

- Setzen Sie *Checked* auf *True*, wenn die Komponente standardmäßig aktiviert angezeigt werden soll.
- Setzen Sie *AllowGrayed* auf *True*, wenn das Kontrollkästchen drei mögliche Zustände (aktiviert, deaktiviert, nicht auswählbar) haben soll.
- Mit *State* können Sie angeben, ob die Komponente aktiviert (*cbChecked*), deaktiviert (*cbUnchecked*) oder nicht auswählbar (*cbGrayed*) angezeigt werden soll.

Optionsfeld (TRadioButton)

Mit Hilfe von Optionsfeldern können Auswahlmöglichkeiten angezeigt werden, die sich gegenseitig ausschließen. Sie können dazu entweder einzelne Optionsfelder oder eine Optionsfeldgruppe (*TRadioGroup*) verwenden, in der die Felder automatisch angeordnet werden. Weitere Informationen finden Sie im Abschnitt »Komponenten gruppieren« auf Seite 2-21.

ToolBar-Komponente (TToolBar)

Mit Hilfe von Symbolleisten können visuelle Steuerelemente auf einfache Weise angeordnet und verwaltet werden. Sie können dazu eine Tafelkomponente (*TPanel*) mit mehreren SpeedButton-Objekten oder die Komponente ToolBar verwenden. Um dieser Schaltflächen hinzuzufügen, klicken Sie mit der rechten Maustaste und wählen *Neue Schaltfläche*. Die Komponente *ToolBar* hat folgende Vorteile: Alle Schaltflächen erhalten automatisch dieselbe Größe und denselben Abstand, bei den anderen in die

Leiste eingefügten Steuerelemente wird automatisch die relative Positon und Höhe angepaßt, die Steuerelementen, die horizontal nicht in die Leiste passen, werden automatisch in eine neue Zeile verschoben, und außerdem stehen verschiedene Anzeigeoptionen wie Transparenz, Popup-Rahmen und Teilerleisten zur Verfügung.

CoolBar-Komponente (TCoolBar)

Eine CoolBar-Komponente (oder Rebar-Komponente) enthält fensterorientierte Steuerelemente in getrennten Abschnitten, deren Größe und Position unabhängig voneinander geändert werden kann. Die Abschnitte können zur Laufzeit mit Hilfe der Griffmarkierungen am linken Rand angepaßt werden.

CoolBar-Objekte können nur verwendet werden, wenn COMCTL32.DLL in der Version 4.70 oder höher im System installiert ist (normalerweise im Verzeichnis WINDOWS\SYSTEM oder WINDOWS\SYSTEM32).

- *Bands* enthält die Abschnitte der Leiste, eine Kollektion von *TCoolBand*-Objekten. Diese können während des Entwurfs im Abschnittseditor hinzugefügt, entfernt oder bearbeitet werden. Sie öffnen den Editor, indem Sie auf die Wertespalte der Eigenschaft *Bands* im Objektinspektor doppelklicken oder die Ellipsen-Schaltfläche (...) wählen. Abschnitte können auch erstellt werden, indem Sie fensterorientierte Steuerelemente aus der Palette einfügen.
- *FixedOrder* bestimmt, ob die Anordnung der Abschnitte zur Laufzeit geändert werden kann.
- *FixedSize* bestimmt, ob die Abschnitte eine einheitliche Höhe erhalten.

Mit Listen arbeiten

Mit Listen können dem Benutzer mehrere Optionen zur Auswahl angeboten werden. Sie werden von folgenden Komponenten verwendet:

Komponente:	Bedeutung:
Listenfeld (*TListBox*)	Eine Liste von Texteinträgen.
CheckListBox (*TCheckListBox*)	Ein Listenfeld, in dem vor jedem Eintrag ein Kontrollkästchen angezeigt wird.
Kombinationsfeld (*TComboBox*)	Ein Eingabefeld mit einem Dropdown-Listenfeld.
Baumdiagramm (*TTreeView*)	Eine hierarchische Baumstruktur.
Listenansicht (*TListView*)	Ein Listenfeld, in dem Grafiken, Spalten und Kopfzeilen angezeigt werden können (die Einträge können verschoben werden).
Datums-/Zeitauswahl (*TDateTimePicker*)	Ein Listenfeld zum Eingeben von Datums- und Zeitwerten.
Monatskalender (*TMonthCalendar*)	Eine Kalenderkomponente zum Auswählen von Datumswerten.

Strings und Grafiken können mit Hilfe der nichtvisuellen Komponenten *TStringList* und *TImageList* verwaltet werden. Weitere Informationen zu Stringlisten finden Sie im Abschnitt »Mit Stringlisten arbeiten« auf Seite 2-32.

Listenfeld (TListBox und TCheckListBox)

Mit Hilfe von Listenfeldern (TListBox und TCheckListBox) kann dem Benutzer eine Liste von Auswahlmöglichkeiten angezeigt werden.

- *Items* enthält die Listeneinträge in Form eines *TStringList*-Objekts.
- *ItemIndex* gibt an, welcher Eintrag in der Liste ausgewählt ist.
- *MultiSelect* bestimmt, ob eine Mehrfachauswahl möglich ist.
- *Sorted* bestimmt, ob die Liste alphabetisch sortiert wird.
- *Columns* gibt die Anzahl der Spalten in der Liste an.
- *IntegralHeight* gibt an, ob nur Einträge in der Liste angezeigt werden, die vertikal Platz finden.
- *ItemHeight* gibt die Höhe der Listeneinträge in Pixel an. Dieser Wert wird bei bestimmten Einstellungen der Eigenschaft *Style* ignoriert.
- *Style* bestimmt, wie die Listeneinträge angezeigt werden. Bei der Standardeinstellung werden sie als reiner Text angezeigt. Sie können jedoch auch Owner-Draw-Listenfelder definieren, bei denen die Einträge grafisch oder mit variierender Höhe dargestellt werden. Informationen zu dieser Art von Komponenten finden Sie unter »Grafiken zu Steuerelementen hinzufügen« auf Seite 6-13.

Kombinationsfeld (TComboBox)

Ein Kombinationsfeld besteht aus einem Eingabefeld und einer Liste. Werte können entweder direkt in das Eingabefeld eingegeben oder in der Liste gewählt werden. Die Eingabe wird anschließend in der Eigenschaft *Text* gespeichert.

Mit der Eigenschaft *Style* können Sie den Stil des Kombinationsfeldes festlegen:

- Bei *csDropdown* wird ein Eingabefeld mit einer Dropdown-Liste angezeigt. Wenn Sie *csDropDownList* wählen, kann das Eingabefeld nicht bearbeitet werden (der Wert muß dann in der Liste gewählt werden). Die Anzahl der Listeneinträge kann mit der Eigenschaft *DropDownCount* festgelegt werden.
- Bei *csSimple* wird ein Kombinationsfeld mit einer Liste fester Größe erstellt, die nicht geschlossen werden kann. Bei diesem Stil muß die Größe der Komponente so festgelegt werden, daß alle Listeneinträge zu sehen sind.
- Mit dem Stil *csOwnerDrawFixed* oder *csOwnerDrawVariable* können Sie Owner-Draw-Kombinationsfelder erstellen. Weitere Informationen finden Sie unter »Grafiken zu Steuerelementen hinzufügen« auf Seite 6-13.

Baumdiagramm (TTreeView)

In einem Baumdiagramm werden die Einträge in einer hierarchischen Baumstruktur angezeigt. Mit Hilfe spezieller Schaltflächen können die einzelnen Zweige ein- und ausgeblendet werden. Sie können die Beschriftungen der Einträge zusammen mit einer Grafik anzeigen oder andere Symbole für die Schaltflächen verwenden. Es kann auch eine Grafik hinzugefügt werden, die den aktuellen Status des jeweiligen Eintrags angibt.

- *Indent* gibt an, um wie viele Pixel untergeordnete Ebenen eingerückt werden.

- Mit *ShowButtons* können Sie die Anzeige der Schaltflächen (+ und –) zum Erweitern der untergeordneten Zweige aktivieren.
- *ShowLines* aktiviert das Anzeigen von Verbindungslinien zwischen den Einträgen.
- *ShowRoot* bestimmt, ob Verbindungslinien zu den Einträgen der obersten Ebene angezeigt werden.

Listenansicht (TListView)

Mit Hilfe einer Listenansicht können Einträge auf verschiedene Arten angezeigt werden. Geben Sie den gewünschten Stil mit der Eigenschaft *ViewStyle* an:

- Bei *vsIcon* und *vsSmallIcon* werden die einzelnen Einträge als Symbol mit einer Beschriftung angezeigt. Sie können zur Laufzeit beliebig mit der Maus verschoben werden.
- *vsList* zeigt die Einträge als beschriftete Symbole an, die nicht verschoben werden können.
- Bei *vsReport* wird jeder Eintrag mehrspaltig in einer eigenen Zeile angezeigt. Die linke Spalte enthält eine kleines Symbol und die Beschriftung. Die nachfolgenden Spalten enthalten anwendungsspezifische Untereinträge. Die Anzeige von Spaltenüberschriften kann mit der Eigenschaft *ShowColumnHeaders* aktiviert werden.

Datums-/Zeitauswahl (TDateTimePicker) und Monatskalender (TMonthCalendar)

Mit Hilfe der Komponente *DateTimePicker* kann ein Listenfeld zum Eingeben von Datums- und Zeitwerten angezeigt werden. MonthCalendar-Komponenten ermöglichen das Anzeigen eines Kalenders, in dem ein bestimmtes Datum oder ein Datumsbereich ausgewählt werden kann. Beide Steuerelemente können nur verwendet werden, wenn eine aktuelle Version von COMCTL32.DLL im System installiert ist (normalerweise im Verzeichnis WINDOWS\SYSTEM oder WINDOWS\SYSTEM32).

Komponenten gruppieren

Eine grafische Benutzeroberfläche ist einfacher zu bedienen, wenn bestimmte Steuerelemente und Informationen gruppiert werden. In Delphi stehen dazu folgende Komponenten zur Verfügung:

Komponente:	Bedeutung:
Gruppenfeld (*TGroupBox)*	Ein Standardgruppenfeld mit einem Titel.
Optionsfeldgruppe (*TRadioGroup*)	Eine einfache Gruppe von Optionsfeldern.
Tafelkomponente (*TPanel*)	Eine flexiblere Gruppe von Steuerelementen.
Bildlauffeld (*TScrollBox*)	Ein bildlauffähiger Bereich.
Registerkomponente (*TTabControl*)	Eine Gruppe von Registerkarten.
Registerkartenkomponente (*TPageControl*)	Eine Gruppe von Registerkarten, die andere Steuerelemente aufnehmen können.
Kopfzeilenkomponente (*THeaderControl*)	Eine Spaltenüberschrift, deren Größe zur Laufzeit geändert werden kann.

Gruppenfeld (TGroupBox) und Optionsfeldgruppe (TRadioGroup)

Ein Gruppenfeld ist eine Windows-Standardkomponente. Mit ihr können zusammengehörige Steuerelemente in einem Formular gruppiert werden. Diese Möglichkeit wird am häufigsten für Optionsfelder verwendet. Plazieren Sie zunächst ein Gruppenfeld im Formular, und fügen Sie anschließend die gewünschten Steuerelemente aus der Komponentenpalette ein. Der Titel des Gruppenfelds wird mit der Eigenschaft *Caption* angegeben.

Die Komponente *TRadioGroup* vereinfacht die Arbeit mit Optionsfeldern. Mit Hilfe der Stringlisten-Eigenschaft *Items* können Optionsfelder im Objektinspektor der Gruppe hinzugefügt werden. Tragen Sie einfach für jede Option einen String in die Liste ein. Dieser wird dann als Beschriftung des betreffenden Optionsfeldes angezeigt. Der Wert der Eigenschaft *ItemIndex* bestimmt, welches Feld aktuell ausgewählt ist. Die Optionsfelder können ein- oder mehrspaltig angezeigt werden. Geben Sie mit der Eigenschaft *Columns* die gewünschte Spaltenanzahl an. Um den Abstand zwischen den einzelnen Optionsfeldern anzupassen, können Sie die Größe der Gruppenkomponente ändern.

Tafelkomponente (TPanel)

Tafelkomponenten (Panel-Komponenten) sind generische Container, in die beliebige andere Steuerelemente eingefügt werden können. Sie können so ausgerichtet werden, daß sie bei Größenänderungen des Formulars ihre relative Position beibehalten. Mit der Eigenschaft *BorderWidth* kann die Breite des Rahmens (in Pixel) angegeben werden.

Bildlauffeld (TScrollBox)

Mit Hilfe von Bildlauffeldern können in einem Formular Bereiche erstellt werden, deren Inhalt verschoben werden kann. In einer Anwendung müssen oft mehr Informationen angezeigt werden, als in einem bestimmten Bildausschnitt Platz finden. Bei manchen Steuerelementen wie Listenfeldern, Memokomponenten und Formularen werden in diesem Fall automatisch Bildlaufleisten hinzugefügt. Es gibt aber Situationen, in denen bestimmte Formularbereiche mit einer Bildlauffunktion versehen werden sollen. Für diese Fälle steht die Komponente *TScrollBox* zur Verfügung.

Ein Bildlauffeld kann, ähnlich einer Tafelkomponente oder einem Gruppenfeld, andere Steuerelemente aufnehmen. Im Gegensatz zu diesen Komponenten wird das Bildlauffeld aber normalerweise nicht angezeigt. Wenn die Steuerelemente jedoch nicht vollständig in den Anzeigebereich passen, werden automatisch Bildlaufleisten angezeigt.

Registerkomponente (TTabControl)

Die Komponente *TTabControl* ähnelt den Registern eines Ordners. Um Register zu erstellen, bearbeiten Sie die Eigenschaft *Tabs* im Objektinspektor. Geben Sie dann im Stringlisten-Editor für jedes Register einen String ein. Die Registerkomponente besteht aus einer Tafel mit einer Gruppe von Steuerelementen. Sie können auf das Wählen der Register reagieren, indem Sie die entsprechenden Aktionen in eine Behandlungsroutine für das Ereignis *OnChange* aufnehmen. Um mehrseitige Dialogfelder zu erstellen, verwenden Sie die Registerkartenkomponente (*TPageControl*).

Registerkartenkomponente (TPageControl)

Mit Hilfe der Registerkartenkomponente können auf einfache Weise mehrseitige Dialogfelder erstellt werden. Um eine neue Registerkarte zu erstellen, klicken Sie mit der rechten Maustaste auf die Komponente und wählen *Neue Seite.*

Kopfzeilenkomponente (THeaderControl)

Eine Kopfzeilenkomponente besteht aus einer Gruppe von Spaltenüberschriften (Abschnitten), deren Größe zur Laufzeit geändert werden kann. Auf die Abschnitte kann mit Hilfe der Eigenschaft *Sections* zugegriffen werden.

Visuelle Rückmeldungen

Es gibt viele Möglichkeiten, Benutzern Informationen über den aktuellen Status einer Anwendung zu geben. So verfügen beispielsweise manche Komponenten wie *TForm* über die Eigenschaft *Caption*, die zur Laufzeit geändert werden kann. Sie können auch mit Hilfe von Dialogfeldern entsprechende Meldungen anzeigen. Außerdem stehen zu diesem Zweck folgende spezialisierte Komponenten oder Eigenschaften zur Verfügung.

Komponente oder Eigenschaft:	Bedeutung:
Beschriftung (*TLabel) und Statischer Text (TStaticText)*	Text, der zur Laufzeit nicht geändert werden kann.
Statusleiste (*TStatusBar*)	Ein Statusbereich (normalerweise am unteren Fensterrand).
Fortschrittsanzeige (*TProgressBar*)	Ein Balken, der anzeigt, wie weit eine bestimmte Aktion fortgeschritten ist.
Hint *und* ShowHint *(Eigenschaften)*	Tooltip-Hilfe (ein Popup-Fenster mit einem Kurzhinweis an der Mausposition).
HelpContext *und* HelpFile *(Eigenschaften)*	Eine Verknüpfung mit der kontextbezogenen Online-Hilfe.

Beschriftung (TLabel) und Statischer Text (TStaticText)

Mit Hilfe von Beschriftungen kann in einem Formular statischer Text angezeigt werden. Sie werden normalerweise neben anderen Steuerelementen plaziert. Die Standardbeschriftung *TLabel* ist nicht fensterorientiert und kann daher nicht den Eingabefokus erhalten. Benötigen Sie eine Beschriftung mit einem Fenster-Handle, verwenden Sie statt dessen *TStaticText*. Beschriftungen verfügen über folgende Eigenschaften:

- *Caption* enthält den Text der Beschriftung.
- *FocusControl* verknüpft die Beschriftung mit einem anderen Steuerelement. Enthält Caption ein Zeichen für den Direktzugriff, kann mit diesem Tastenkürzel auf das angegebene Steuerelement zugegriffen werden.
- *ShowAccelChar* bestimmt, ob in der Beschriftung ein unterstrichenes Zeichen für den Direktzugriff angezeigt wird. Hat die Eigenschaft den Wert *True*, wird das Zeichen nach einem kaufmännischen Und (&) unterstrichen angezeigt und kann als Tastenkürzel verwendet werden.

- *Transparent* gibt an, ob der Hintergrund der Komponente transparent angezeigt wird.

Statusleiste (TStatusBar)

Sie können zwar auch mit Hilfe einer Tafel Statusinformationen anzeigen, es ist aber einfacher, die Windows-Standardstatusleiste zu verwenden. Die Eigenschaft *Align* der Komponente hat den Standardwert *alBottom* (unterer Fensterrand), mit dem automatisch Position und Größe an das Formular angepaßt werden.

Eine Statusleiste ist normalerweise in mehrere Abschnitte unterteilt. Um diese zu erstellen, bearbeiten Sie im Objektinspektor die Eigenschaft *Panels*. Geben Sie dann mit Hilfe des Abschnittseditors zu jedem Abschnitt Werte für die Eigenschaften *Width*, *Alignment* und *Text* an. Letztere enthält den angezeigten Text.

Fortschrittsanzeige (TProgressBar)

Mit Hilfe einer Fortschrittsanzeige kann der Benutzer über den Fortgang einer laufenden Operation oder eines Hintergrundprozesses informiert werden. Die Anzeige besteht aus einem Balken, der während der Aktion von links nach rechts gefüllt wird.

Abbildung 2.4 Eine Fortschrittsanzeige

Position gibt die aktuelle Position an. Mit den Eigenschaften *Max* und *Min* kann der Bereich für die Anzeige festgelegt werden. Um die Fortschrittsposition zu vergrößern, erhöhen Sie Position durch einen Aufruf der Methode *StepBy* oder *StepIt*. Mit der Eigenschaft *Step* kann die von *StepIt* verwendete Schrittweite festgelegt werden.

Hilfe- und Hinweiseigenschaften

Die meisten visuellen Steuerelemente können zur Laufzeit kontextbezogene Hilfeinformationen oder Kurzhinweise anzeigen. Mit den Eigenschaften *HelpContext* und *HelpFile* kann eine Kontextnummer und eine Hilfedatei angegeben werden.

Die Eigenschaft *Hint* enthält den Text, der in einem Popup-Fenster angezeigt wird, wenn sich der Mauszeiger über dem Steuerelement oder Menüeintrag befindet. Um diese Funktion zu aktivieren, setzen Sie *ShowHint* auf *True*. Wenn Sie *ParentShowHint* auf *True* setzen, ist der Wert von *ShowHint* mit dem der Eigenschaft des Containers identisch.

Gitterkomponenten

Mit Hilfe von Gitterkomponenten können Informationen in Zeilen und Spalten angezeigt werden. In Datenbankanwendungen werden die Komponenten *TDBGrid* und *TDBCtrlGrid* verwendet. Informationen hierzu finden Sie in Kapitel 26, »Datensensitive Steuerelemente«. Setzen Sie zu anderen tabellarischen Anzeigezwecken die nachfolgend beschriebenen Zeichen- und String-Gitter ein.

Zeichengitter (TDrawGrid)

Mit einem Zeichengitter können beliebige Daten in einem tabellarischen Format angezeigt werden. Die Zellen können in einer Ereignisbehandlungsroutine für *OnDrawCell* mit Informationen gefüllt werden.

- Mit der Methode *CellRect* können die Bildschirmkoordinaten einer bestimmten Zelle und mit der Methode *MouseToCell* kann die Spalte und Zeile der Zelle an bestimmten Bildschirmkoordinaten ermittelt werden. Die Eigenschaft *Selection* gibt die aktuell ausgewählten Zellen an.
- Die Eigenschaft *TopRow* ermöglicht den Zugriff auf die oberste Gitterzeile. *LeftCol* gibt die erste sichtbare Spalte auf der linken Seite des Gitters an. *VisibleColCount* und *VisibleRowCount* geben die Anzahl der sichtbaren Spalten und Zeilen des Steuerelements an.
- Die Höhe bzw. Breite einer Spalte oder Zeile kann mit der Eigenschaft *ColWidths* oder *RowHeights* geändert werden. Mit der Eigenschaft *GridLineWidth* kann die Stärke der Gitterlinien festgelegt werden. Die Eigenschaft *ScrollBars* ermöglicht das Hinzufügen von Bildlaufleisten.
- Mit den Eigenschaften *FixedCols* und *FixedRows* können Sie Spalten bzw. Zeilen angeben, die immer (also auch bei einem Bildlauf) angezeigt werden. Sie können diesen auch mit *FixedColor* eine bestimmte Farbe zuweisen.
- Mit den Eigenschaften Options, *DefaultColWidth* und *DefaultRowHeight* können weitere Anzeige- und Funktionsoptionen festgelegt werden.

String-Gitter (TStringGrid)

Die von *TDrawGrid* abgeleitete Komponente *TStringGrid* verfügt über spezielle Funktionen für die Anzeige von Strings. Dabei kann jedem String ein Objekt zugeordnet werden. Die Strings werden mit der Eigenschaft *Cells*, die Objekte mit *Objects* verwaltet. Auf die Strings und Objekte einer bestimmten Spalte oder Zeile kann mit der Eigenschaft *Cols* bzw. *Rows* zugegriffen werden.

Grafikkomponenten

Mit Hilfe der folgenden Komponenten können Sie Grafiken zu Ihren Anwendungen hinzufügen.

Komponente:	Anzeige:
Grafik (*TImage)*	Grafikdatei.
Form (*TShape*)	Geometrische Formen.
Rahmen (*TBevel*)	Dreidimensionale Linien und Rahmen.
Zeichenfeld (*TPaintBox*)	Vom Programm zur Laufzeit gezeichnete Grafiken.
Animation (*TAnimate*)	AVI-Dateien.

Grafik (TImage)

Mit Hilfe der Komponente *TImage* können in einem Formular Grafiken wie Bitmaps, Symbole oder Metadateien angezeigt werden. Geben Sie die betreffende Grafik mit

der Eigenschaft *Picture* an. Mit den Eigenschaften *Center*, *AutoSize*, *Stretch* und *Transparent* können verschiedene Anzeigeoptionen festgelegt werden.

Form (TShape)

Mit Hilfe der Komponente *TShape* können verschiedene geometrische Formen angezeigt werden. Sie ist nicht fensterorientiert und kann daher nicht den Eingabefokus erhalten. Die Form des Steuerelements wird mit der Eigenschaft *Shape* angegeben. Mit der Eigenschaft *Brush* können Sie die Farbe der Form ändern oder ein Muster hinzufügen. Sie enthält ein Objekt des Typs *TBrush*. Die Eigenschaften *Color* und *Style* dieses Objekts bestimmen, wie die Form gezeichnet wird.

Rahmen (TBevel)

Die Rahmenkomponente ist eine Linie, die erhöht oder vertieft angezeigt werden kann und dadurch einen räumlichen Eindruck vermittelt. Manche Komponenten (beispielsweise *TPanel*) besitzen spezielle Eigenschaften, um diese Art von Rahmen zu erstellen. Stehen diese nicht zur Verfügung, können Sie mit *TBevel* abgeschrägte Linien, Rechtecke oder Rahmen erstellen.

Zeichenfeld (TPaintBox)

Die Komponente *TPaintBox* stellt einen rechteckigen Bereich zur Verfügung, in dem Grafiken programmseitig erstellt werden können. Die Ausgabeoperationen müssen in der Ereignisbehandlungsroutine für *OnPaint* auf der in Canvas angegebenen Zeichenfläche durchgeführt werden. Außerhalb des Zeichenfeldes sind keine Ausgaben möglich. Weitere Informationen hierzu finden Sie unter »Grafikprogrammierung im Überblick« auf Seite 7-1.

Animationskomponente (TAnimate)

Die Komponente *TAnimate* ist ein Fenster, in dem AVI-Clips (ohne Ton) abgespielt werden können. AVI-Clips bestehen wie ein Film aus einer Folge von Einzelbildern. Sie können auch eine Tonspur enthalten, jedoch wird diese von der Komponente ignoriert. Es können sowohl unkomprimierte als auch mit dem RLE-Verfahren komprimierte Videodateien verwendet werden.

Windows-Standarddialogfelder

Mit Hilfe der Registerkarte *Dialoge* in der Komponentenpalette können Sie die Standarddialogfelder von Windows in Anwendungen einfügen. Auf diese Weise erhalten Ihre Programme eine dem Benutzer vertraute, konsistente Oberfläche für häufig durchgeführte Operationen wie Öffnen oder Speichern von Dateien, Auswählen einer Schriftart und Drucken. Die Dialogfelder müssen zur Laufzeit durch einen Aufruf ihrer Methode *Execute* geöffnet werden.

Eigenschaften von Komponenten festlegen

Als **published** deklarierte Eigenschaften können während des Entwurfs im Objektinspektor bearbeitet werden. In einigen Fällen stehen dazu spezielle Eigenschaftseditoren zur Verfügung.

Um den Wert einer Eigenschaft zur Laufzeit festzulegen, weisen Sie den gewünschten Wert im Quelltext der Anwendung zu.

Genaue Informationen zu den Eigenschaften der einzelnen Komponenten finden Sie in der VCL-Hilfe.

Den Objektinspektor verwenden

Wenn Sie eine Komponente im Formular auswählen, werden ihre Eigenschaften im Objektinspektor angezeigt und können gegebenenfalls bearbeitet werden. Mit der Taste *Tab* können Sie zwischen der Namens- und der Wertespalte einer Eigenschaft wechseln. In der Namensspalte können Sie zu einer anderen Eigenschaft gelangen, indem Sie den ersten Buchstaben ihres Namens eingeben. Bei Booleschen oder Aufzählungseigenschaften können Sie den Wert mit Hilfe einer Dropdown-Liste oder durch Doppelklicken auf die Wertespalte angeben. Befindet sich neben dem Eigenschaftsnamen ein Pluszeichen (+), kann eine Liste mit den enthaltenen Eigenschaften geöffnet werden.

Standardmäßig werden die Eigenschaften in der Kategorie *Legacy* nicht angezeigt. Um diese Einstellung zu ändern, klicken Sie im Objektinspektor mit der rechten Maustaste und wählen *Ansicht*. Weitere Informationen finden Sie unter “Eigenschaftskategorien” in der Online-Hilfe.

Sind mehrere Komponenten ausgewählt, werden die gemeinsamen Eigenschaften (außer *Name*) angezeigt. Unterscheiden sich die Werte einer dieser Eigenschaften, wird entweder der Standardwert oder der Wert der zuerst ausgewählten Komponente angezeigt. Wenn Sie den Wert einer gemeinsamen Eigenschaft ändern, wirkt sich dies auf alle Komponenten aus.

Eigenschaftseditoren verwenden

Manche Eigenschaften (z. B. *Font*) verfügen über einen speziellen Eigenschaftseditor. Für diese wird eine Ellipsen-Schaltfläche (...) in der Wertespalte angezeigt, sobald sie im Objektinspektor ausgewählt werden. Um den Editor zu öffnen, wählen Sie die Schaltfläche oder doppelklicken auf die Wertespalte. Bei bestimmten Komponenten wird auch ein Eigenschaftseditor geöffnet, wenn Sie im Formular auf dem Objekt doppelklicken.

Mit Hilfe eines Eigenschaftseditors können komplexe Eigenschaften in einem einzigen Dialogfeld festgelegt werden. Sie ermöglichen eine Validierung der Eingabe und häufig eine Vorschau auf das Ergebnis einer Wertzuweisung.

Eigenschaften zur Laufzeit festlegen

Sie können jede Eigenschaft, der ein Wert zugewiesen werden kann, zur Laufzeit im Quelltext ändern. So kann beispielsweise folgendermaßen der Titel eines Formulars dynamisch festgelegt werden:

```
Form1.Caption := MyString;
```

Methoden aufrufen

Methoden folgen denselben Aufrufkonventionen wie normale Prozeduren und Funktionen. So verfügen beispielsweise visuelle Steuerelemente über die Methode *Repaint*, mit der die Komponente auf dem Bildschirm aktualisiert werden kann. Bei einem Zeichengitter sieht dann der Aufruf folgendermaßen aus:

```
DrawGrid1.Repaint;
```

Der Gültigkeitsbereich einer Methode entspricht dem einer Eigenschaft. Um beispielsweise ein Formular in einer Ereignisbehandlungsroutine einer seiner untergeordneten Komponenten zu aktualisieren, müssen Sie den Namen des Formulars nicht im Methodenaufruf angeben:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Repaint;
end;
```

Weitere Informationen über Gültigkeitsbereiche finden Sie im Abschnitt »Gültigkeitsbereich und Qualifizierer« auf Seite 2-6.

Mit Ereignissen und Ereignisbehandlungsroutinen arbeiten

In Delphi wird nahezu der gesamte Quelltext Ihrer Anwendungen direkt oder indirekt als Reaktion auf Ereignisse ausgeführt. Ein Ereignis ist eine spezielle Art von Eigenschaft, die einem bestimmten Laufzeitgeschehen (oftmals einer Benutzeraktion) entspricht. Den Quelltext mit den direkten Reaktionen auf ein Ereignis nennt man Ereignisbehandlungsroutine. In diesem Abschnitt erhalten Sie Informationen zu den folgenden Themen:

- Eine neue Ereignisbehandlungsroutine erstellen
- Eine Behandlungsroutine für das Standardereignis einer Komponente erstellen
- Ereignisbehandlungsroutinen suchen
- Ereignissen eine vorhandene Behandlungsroutine zuordnen
- Menüereignissen eine Behandlungsroutine zuordnen
- Ereignisbehandlungsroutinen löschen

Eine neue Ereignisbehandlungsroutine erstellen

Sie können für das Formular oder seine Komponenten mit Hilfe des Objektinspektors eine Ereignisbehandlungsroutine erstellen. Bei dieser Vorgehensweise werden Teile des Quelltextes automatisch generiert und gepflegt. Gehen Sie dazu folgendermaßen vor:

1. Wählen Sie eine Komponente (oder das Formular) aus.
2. Klicken Sie im Objektinspektor auf das Register *Ereignisse*. Es wird dann eine Registerkarte mit allen für die Komponente definierten Ereignissen geöffnet.

3 Wählen Sie das gewünschte Ereignis aus. Doppelklicken Sie anschließend auf die Wertespalte, oder drücken Sie *Strg+Return*. Es wird dann der Quelltexteditor geöffnet. Der Cursor befindet sich bereits an der Stelle, an der Sie den Quelltext eingeben können.

4 Geben Sie in den **begin...end**-Block der Routine den Quelltext ein, der bei Auftreten des Ereignisses ausgeführt werden soll.

Eine Behandlungsroutine für das Standardereignis einer Komponente erstellen

Viele Komponenten verfügen über ein Standardereignis, das von der Komponente zur Laufzeit am häufigsten verarbeitet wird. So ist *OnClick* beispielsweise das Standardereignis einer Schaltfläche. Um die Standard-Ereignisbehandlungsroutine zu generieren, doppelklicken Sie im Formular auf die betreffende Komponente. Dadurch wird automatisch der Quelltexteditor geöffnet und der Cursor an die Stelle gesetzt, an der Sie den Quelltext eingeben können.

Nicht alle Komponenten haben ein Standardereignis. Es gibt auch Steuerelemente wie *TBevel*, die auf kein Ereignis reagieren. Andere Objekte verhalten sich unterschiedlich, wenn auf sie im Formular-Designer doppelgeklickt wird. Bei vielen wird beispielsweise ein Standard-Eigenschaftseditor oder ein anderes Dialogfeld geöffnet, in dem Eigenschaften bearbeitet werden können.

Ereignisbehandlungsroutinen suchen

Eine Standard-Ereignisbehandlungsroutine, die Sie durch Doppelklicken auf eine Komponente im Formular-Designer erstellt haben, können Sie auf dieselbe Weise wieder auffinden. Doppelklicken Sie einfach auf die Komponente. Es wird dann der Quelltexteditor mit der betreffenden Routine geöffnet und der Cursor in die erste Quelltextzeile gesetzt.

So zeigen Sie eine Behandlungsroutine an, die nicht dem Standardereignis zugeordnet ist:

1 Wählen Sie im Formular die betreffende Komponente aus.

2 Klicken Sie im Objektinspektor auf das Register *Ereignisse*.

3 Doppelklicken Sie auf die Wertespalte des gewünschten Ereignisses. Es wird dann der Quelltexteditor mit der betreffenden Routine geöffnet und der Cursor in die erste Quelltextzeile gesetzt.

Ereignissen eine vorhandene Behandlungsroutine zuordnen

Sie können Behandlungsroutinen erstellen, die auf mehrere Ereignisse reagieren. In vielen Anwendungen sind beispielsweise Symbolleisten mit Schaltflächen vorhanden, die Menübefehlen entsprechen. Wenn nun eine Schaltfläche und ein Menüeintrag dieselbe Aktion ausführen, erstellen Sie eine Behandlungsroutine und weisen diese dem Ereignis *OnClick* beider Komponenten zu.

So ordnen Sie einem Ereignis eine vorhandene Behandlungsroutine zu:

1. Wählen Sie im Formular die Komponente aus, deren Ereignis behandelt werden soll.
2. Wählen Sie in der Registerkarte *Ereignisse* des Objektinspektors das Ereignis aus, dem Sie die Routine zuordnen möchten.
3. Klicken Sie in der Wertespalte auf den Dropdown-Pfeil, um eine Liste der bereits erstellten Ereignisbehandlungsroutinen anzuzeigen (es werden nur Routinen aufgeführt, die für gleichnamige Ereignisse in diesem Formular erstellt wurden). Wählen Sie die gewünschte Routine.

Auf diese Weise können Sie Routinen problemlos wiederverwenden. *Aktionslisten* bieten aber eine leistungsfähigere und flexiblere Möglichkeit, Quelltext für die Reaktion auf Benutzeraktionen zentral zu verwalten. Weitere Informationen zu diesen Komponenten finden Sie im Abschnitt »Aktionslisten verwenden« auf Seite 5-39.

Der Parameter Sender

Der Parameter *Sender* einer Ereignisbehandlungsroutine gibt an, welche Komponente das Ereignis erhalten und somit die Routine aufgerufen hat. Auf diese Weise kann die Routine von mehreren Komponenten verwendet werden. Je nach aufrufendem Objekt wird dann eine andere Aktion ausgeführt. Zu diesem Zweck testen Sie den Parameter *Sender* in einer **if...then...else**-Anweisung. Im folgenden Beispiel wird der Name der Anwendung nur im Titel eines Dialogfelds angezeigt, wenn das Ereignis *OnClick* der Komponente *Button1* ausgelöst wurde.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
   AboutBox.Caption := 'Info über ' + Application.Title
else AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

Gemeinsame Ereignisse anzeigen und behandeln

Wenn bestimmte Ereignisse für mehrere Komponenten vorhanden sind, können sie im Objektinspektor angezeigt werden. Wählen Sie zuerst die betreffenden Objekte durch Klicken mit gedrückter Umschalttaste aus, und klicken Sie dann auf das Register *Ereignisse* des Objektinspektors. Erstellen Sie nun für das gewünschte Ereignis eine neue Behandlungsroutine, oder ordnen Sie eine bereits definierte Routine zu.

Menüereignissen eine Behandlungsroutine zuordnen

Sie können die Menüs für Ihre Anwendungen im Menü-Designer mit Hilfe der Komponenten MainMenu und PopupMenu erstellen. Die Menüs sind aber erst funktionstüchtig, wenn die verschiedenen Menüeinträge auf das Ereignis *OnClick* reagieren. Es wird ausgelöst, wenn der Benutzer den jeweiligen Eintrag wählt oder das betreffende Tastenkürzel eingibt. In diesem Abschnitt erfahren Sie, wie den Menüoptionen Ereignisbehandlungsroutinen zugeordnet werden können. Informationen zu Menü-Designer und -komponenten finden Sie unter »Menüs erstellen und verwalten« auf Seite 5-16.

So erstellen Sie eine Ereignisbehandlungsroutine für einen Menüeintrag:

1. Doppelklicken Sie auf ein *MainMenu*- oder *PopupMenu*-Objekt, um den Menü-Designer zu öffnen.
2. Wählen Sie den gewünschten Eintrag im Menü-Designer aus. Vergewissern Sie sich im Objektinspektor, daß die Eigenschaft *Name* einen Wert enthält.
3. Doppelklicken Sie im Menü-Designer auf den Eintrag. Es wird dann der Quelltexteditor geöffnet. Der Cursor befindet sich bereits an der Stelle, an der Sie den Quelltext eingeben können.
4. Geben Sie in den **begin...end**-Block den Quelltext ein, der beim Wählen des Menübefehls ausgeführt werden soll.

So ordnen Sie einem Menüeintrag eine vorhandene *OnClick*-Ereignisbehandlungsroutine zu:

1. Doppelklicken Sie auf ein *MainMenu*- oder *PopupMenu*-Objekt, um den Menü-Designer zu öffnen.
2. Wählen Sie den gewünschten Eintrag im Menü-Designer aus. Vergewissern Sie sich im Objektinspektor, daß die Eigenschaft *Name* einen Wert enthält.
3. Aktivieren Sie die Registerkarte *Ereignisse* des Objektinspektors, und klicken Sie auf den Dropdown-Pfeil neben *OnClick*, um eine Liste der bereits erstellten Ereignisbehandlungsroutinen zu öffnen (es werden nur die Routinen für *OnClick*-Ereignisse in diesem Formular angezeigt). Wählen Sie die gewünschte Routine in der Liste.

Ereignisbehandlungsroutinen löschen

Wenn Sie eine Komponente im Formular-Designer löschen, wird ihre Definition automatisch aus der Typdeklaration des Formulars entfernt. Es werden jedoch nicht die zugehörigen Methoden aus der Unit-Datei entfernt, da diese auch von anderen Komponenten des Formulars aufgerufen werden können. Löschen Sie eine Methode (z. B. eine Ereignisbehandlungsroutine) manuell, müssen Sie unbedingt die Vorwärtsdeklaration (im **interface**-Abschnitt) und die Implementierung (im **implementation**-Abschnitt) der Methode entfernen. Andernfalls wird beim Erstellen des Projekts ein Compiler-Fehler ausgegeben.

Hilfsobjekte verwenden

Die VCL enthält eine Vielzahl nichtvisueller Objekte, mit denen allgemeine Programmieraufgaben auf einfache Weise durchgeführt werden können. Dieser Abschnitt enthält Informationen zu einigen dieser Objekte:

- Mit Listen arbeiten
- Mit Stringlisten arbeiten
- Mit Registrierung und INI-Dateien arbeiten
- Daten auf eine Festplatte oder ein anderes Speichergerät schreiben

Mit Listen arbeiten

Es gibt eine Reihe von VCL-Objekten mit Funktionen zum Erstellen und Verwalten von Listen:

- *TList* verwaltet eine Liste von Zeigern.
- *TObjectList* verwaltet eine Liste von Instanzobjekten im Hauptspeicher.
- *TComponentList* verwaltet eine Liste von Komponenten (Instanzen der von *TComponent* abgeleiteten Klassen) im Hauptspeicher.
- *TQueue* verwaltet eine FIFO-Liste von Zeigern.
- *TStack* verwaltet eine LIFO-Liste von Zeigern.
- *TObjectQueue* verwaltet eine FIFO-Liste von Objekten.
- *TObjectStack* verwaltet eine LIFO-Liste von Objekten.
- *TClassList* verwaltet eine Liste von Klassentypen.
- *TCollection*, *TOwnedCollection* und *TCollectionItem* verwalten indizierte Kollektionen von speziell definierten Elementen.
- *TStringList* verwaltet eine Liste von Strings.

Weitere Informationen zu diesen Objekten können Sie der *VCL-Referenz* in der Online-Hilfe entnehmen.

Mit Stringlisten arbeiten

In Anwendungen müssen häufig Listen von Strings verwaltet werden. Es müssen beispielsweise Einträge in einem Kombinationsfeld angezeigt, Textzeilen in einem Memofeld verwaltet, die installierten Schriftarten aufgelistet oder die Zeilen und Spalten einer Gitterkomponente beschriftet werden. Die VCL stellt dazu eine allgemeine Schnittstelle zu allen Arten von Stringlisten in Form des Objekts *TStrings* und der abgeleiteten Klasse *TStringList* zur Verfügung. Diese Objekte besitzen also Funktionen zum Verwalten von Stringlisten und sind untereinander austauschbar. So können beispielsweise zuerst die Zeilen einer Memokomponente (die Instanzen von *TStrings* sind) bearbeitet und anschließend für die Einträge eines Kombinationsfeldes (ebenfalls eine Instanz von *TStrings*) verwendet werden.

Für eine Stringlisten-Eigenschaft wird im Objektinspektor der Wert *TStrings* angezeigt. Wenn Sie auf diesen doppelklicken, wird ein Editorfenster geöffnet, in dem Sie Textzeilen (Strings) hinzufügen, entfernen und bearbeiten können.

Zur Laufzeit können mit Stringlisten folgende Operationen durchgeführt werden:

- Stringlisten laden und speichern
- Eine neue Stringliste erstellen
- Mit den Strings in einer Liste arbeiten
- Einer Stringliste Objekte zuordnen

Stringlisten laden und speichern

Der Inhalt einer Stringliste kann auf einfache Weise in einer Textdatei gespeichert und später wieder in dieselbe oder eine andere Liste geladen werden. Die Objekte verfügen dazu über die Methoden *SaveToFile* und *LoadFromFile*. Jeder String entspricht einer Zeile in der Textdatei. Mit Hilfe dieser Methoden können Sie beispielsweise aus einer Memokomponente einen einfachen Texteditor erstellen oder für Kombinationsfelder Listen mit Einträgen speichern.

Im folgenden Beispiel wird die Datei C:\WINDOWS\WIN.INI in ein Memofeld geladen und anschließend als Sicherungskopie unter dem Namen WIN.BAK gespeichert.

```
procedure EditWinIni;
var
        FileName: string;{ Variable für Dateinamen }
begin
           FileName := 'C:\WINDOWS\WIN.INI';{ Dateinamen angeben }
   with Form1.Memo1.Lines do
           begin
    LoadFromFile(FileName);{ Aus Datei laden }
    SaveToFile(ChangeFileExt(FileName, '.BAK'));{ In Sicherungskopie speichern }
   end;
end;
```

Eine neue Stringliste erstellen

Sie arbeiten in Ihren Anwendungen meistens mit Stringlisten, die zu einer bestimmten Komponente gehören. In manchen Situationen werden aber unabhängige Listen benötigt, beispielsweise für die Einträge einer Lookup-Tabelle. Das Vorgehen zum Erstellen und Verwalten einer Stringliste hängt davon ab, ob sie kurzlebig (in derselben Routine erstellt, verwendet und freigegeben) oder langlebig (bis zum Beenden der Anwendung verfügbar) ist. Bei beiden Arten müssen Sie unbedingt darauf achten, sie nach ihrer Verwendung wieder freizugeben.

Kurzlebige Stringlisten

Wird eine Stringliste nur in einer Routine benötigt, kann sie in dieser erstellt, verwendet und wieder freigegeben werden. Dies ist die sicherste Art, mit Stringlisten zu arbeiten. Listenobjekte weisen für sich und ihre Strings den benötigten Speicher zu. Es ist daher sehr wichtig, daß Sie die Operationen mit dem Objekt in einem geschützten **try..finally**-Block durchführen, damit der Speicher auch nach dem Auftreten einer Exception wieder freigegeben wird.

1. Erstellen Sie die Stringliste.
2. Verwenden Sie das Objekt im **try**-Abschnitt eines **try...finally**-Blocks.
3. Geben Sie die Stringliste im **finally**-Abschnitt wieder frei.

In der folgenden Behandlungsroutine für das Ereignis *OnClick* einer Schaltfläche wird eine Stringliste erstellt, verwendet und anschließend wieder freigegeben.

```
procedure TForm1.Button1Click(Sender: TObject);
var
        TempList: TStrings;{ Listenobjekt deklarieren }
```

```
begin
   TempList := TStringList.Create;{ Listenobjekt erstellen }
   try
       { Stringliste verwenden }
   finally
            TempList.Free;{ Listenobjekt freigeben }
   end;
end;
```

Langlebige Stringlisten

Wenn Sie eine Stringliste während der gesamten Laufzeit einer Anwendung benötigen, erstellen Sie das Objekt beim Programmstart und geben es vor dem Beenden wieder frei.

1. Fügen Sie in die Deklaration des Hauptformulars ein Feld des Typs *TStrings* ein.
2. Erstellen Sie eine Behandlungsroutine für das Ereignis *OnCreate* des Formulars (da es sich um das Standardereignis eines Formulars handelt, doppelklicken Sie einfach auf die Formularkomponente). Die Routine wird ausgeführt, bevor das Formular auf dem Bildschirm angezeigt wird. Erstellen Sie hier das Listenobjekt, und weisen Sie es dem in Schritt 1 erstellten Feld zu.
3. Erstellen Sie eine Behandlungsroutine für das Ereignis *OnDestroy* des Formulars, und geben Sie mit dieser die Stringliste frei.

In diesem Beispiel wird eine Stringliste erstellt, in die bei jedem Mausklick ein String eingefügt wird. Vor dem Beenden der Anwendung wird die Liste in eine Textdatei geschrieben und dann freigegeben.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
           TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
              Shift: TShiftState; X, Y: Integer);
   private
    { Private-Deklarationen }
   public
    { Public-Deklarationen }
    ClickList: TStrings;{ Feld für Stringliste deklarieren }
   end;

var
        Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
            begin
     ClickList := TStringList.Create;{ Listenobjekt erstellen }
   end;

procedure TForm1.FormDestroy(Sender: TObject);
            begin
     ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.LOG'));{ Liste speichern }
   ClickList.Free;{ Listenobjekt freigeben }
   end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
              Shift: TShiftState; X, Y: Integer);
begin
           ClickList.Add(Format('Mausklicke bei (%d, %d)', [X, Y]));{ String einfügen }
   end;

end.
```

Mit den Strings in einer Liste arbeiten

Folgende Operationen müssen häufig mit den Strings in einem Listenobjekt durchgeführt werden:

- Die Strings in einer Liste zählen
- Auf einen bestimmten String zugreifen
- Die Listenposition eines Strings ermitteln
- Eine Stringliste in einer Schleife verarbeiten
- Einen String einer Liste hinzufügen
- Die Listenposition eines Strings ändern
- Einen String aus einer Liste löschen
- Eine ganze Stringliste kopieren

Die Strings in einer Liste zählen

Mit Hilfe der Nur-Lesen-Eigenschaft *Count* kann die Anzahl der Strings in einem Listenobjekt ermittelt werden. Der Index von Stringlisten beginnt mit Null. Daher ist der Wert von *Count* immer um eins größer als der Index des letzten Strings.

Auf einen bestimmten String zugreifen

Die Array-Eigenschaft *Strings* enthält die Strings des Listenobjektes (die Indizierung beginnt bei Null). Sie ist die Standardeigenschaft einer Stringliste. Aus diesem Grund muß der Bezeichner Strings beim Zugriff auf die Liste nicht angegeben werden. So ist beispielsweise die Anweisung

```
StringList1.Strings[0] := 'Das ist der erste String.';
```

mit folgender identisch:

```
StringList1[0] := 'Das ist der erste String.';
```

Die Listenposition eines Strings ermitteln

Mit Hilfe der Methode *IndexOf* kann die Position eines bestimmten Strings in der Stringliste ermittelt werden. Sie gibt den Index des ersten Strings zurück, der mit dem übergebenen Parameter übereinstimmt. Kann der String nicht gefunden werden, wird –1 zurückgegeben. Mit IndexOf kann nur nach genauen Übereinstimmungen gesucht werden. Um nach einem Teilstring (einer bestimmten Zeichenfolge) zu suchen, müssen Sie das Listenobjekt in einer Schleife verarbeiten.

Im folgenden Beispiel wird mit *IndexOf* geprüft, ob sich ein bestimmter Dateiname in einer Dateiliste befindet:

```
if FileListBox1.Items.IndexOf('WIN.INI') > -1 ...
```

Eine Stringliste in einer Schleife verarbeiten

Um die Strings in einem Listenobjekt nacheinander zu verarbeiten, verwenden Sie eine **for**-Schleife. Geben Sie dabei für den Schleifenzähler als Anfangswert Null und als Endwert Count – 1 an.

Im folgenden Beispiel werden die Strings eines Listenfeldes in Großbuchstaben konvertiert:

```
procedure TForm1.Button1Click(Sender: TObject);
            var
     Index: Integer;
begin
   for Index := 0 to ListBox1.Items.Count - 1 do
             ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
end;
```

Einen String einer Liste hinzufügen

Mit Hilfe der Methode *Add* kann ein String am Ende der Liste hinzugefügt werden. Übergeben Sie dabei den neuen String als Parameter. Um einen String in die Liste einzufügen, rufen Sie die Methode *Insert* mit zwei Parametern auf: dem String und dem Index der Position, an der er eingefügt werden soll. So wird beispielsweise mit folgender Anweisung der String „Drei“ an der dritten Position in die Liste eingefügt:

```
Insert(2, 'Drei');
```

Sie können auch mit der Methode *AddStrings* Strings einer Liste einer anderen hinzufügen:

```
StringList1.AddStrings(StringList2);  { StringList2 wird StringList1 hinzugefügt. }
```

Die Listenposition eines Strings ändern

Mit Hilfe der Methode *Move* können Sie die Position eines Strings in einer Stringliste ändern. Übergeben Sie dabei als Parameter den aktuellen und den neuen Index des betreffenden Strings. So kann beispielsweise folgendermaßen der dritte String in einem Listenobjekt an die fünfte Position verschoben werden:

```
Move(2, 4)
```

Einen String aus einer Liste löschen

Mit der Methode *Delete* können Sie einen bestimmten String aus dem Listenobjekt löschen. Übergeben Sie dabei den Index des betreffenden Strings. Ist die Indexposition nicht bekannt, kann sie mit *IndexOf* ermittelt werden. Um alle Strings aus der Liste zu löschen, verwenden Sie die Methode *Clear*.

Im folgenden Beispiel wird zunächst mit der Methode *IndexOf* die Position eines Strings ermittelt. Anschließend wird dieser String mit *Delete* aus der Liste entfernt:

```
with ListBox1.Items do
            begin
   if IndexOf('bureaucracy') > -1 then
             Delete(IndexOf('bureaucracy'));
end;
```

Eine ganze Stringliste kopieren

Mit Hilfe der Methode *Assign* können ganze Stringlisten kopiert werden. Dabei wird der Inhalt der Zielliste überschrieben. Sollen die Strings hinzugefügt werden, ohne die Zielliste zu überschreiben, verwenden Sie die Methode *AddStrings*. Im folgenden Beispiel werden die Einträge eines Kombinationsfeldes in ein Memofeld kopiert. Der Inhalt der Memokomponente wird dabei überschrieben:

```
Memo1.Lines.Assign(ComboBox1.Items);   { Vorhandene Strings werden überschrieben. }
```

Im nächsten Beispiel werden die Einträge des Kombinationsfeldes an den Inhalt der Memokomponente angefügt:

```
Memo1.Lines.AddStrings(ComboBox1.Items);   { Die neuen Strings werden am Ende hinzugefügt. }
```

Verwenden Sie für lokale Kopien von Stringlisten die Methode *Assign*. Bei einer normalen Zuweisung der Form

```
StringList1 := StringList2;
```

geht das ursprüngliche Listenobjekt verloren. Dies führt oft zu unvorhersehbaren Ergebnissen.

Einer Stringliste Objekte zuordnen

In einer Stringliste können außer den Strings (Eigenschaft *Strings*) auch Verweise auf Objekte verwaltet werden. Diese werden in der Eigenschaft *Objects* gespeichert, die ebenfalls ein Array mit einem nullbasierten Index ist. Am häufigsten wird diese Möglichkeit für Owner-Draw-Steuerelemente verwendet, um den Strings bestimmte Bitmaps zuzuordnen.

Mit der Methode *AddObject* oder *InsertObject* können String und Objekt in einer Operation in die Liste aufgenommen werden. *IndexOfObject* gibt den Index des ersten Strings zurück, der zu einem bestimmten Objekt gehört. Mit Methoden wie *Delete*, *Clear* und *Move* kann mit Strings und mit Objekten gearbeitet werden. So wird beispielsweise beim Löschen eines Strings auch das entsprechende Objekt (falls vorhanden) entfernt.

Um ein Objekt einem vorhandenen String zuzuordnen, geben Sie denselben Index bei der Zuweisung an Objects an. Objekte ohne zugehörigen String sind nicht möglich.

Mit Registrierung und INI-Dateien arbeiten

Die Systemregistrierung von Windows ist eine hierarchische Datenbank, in der die meisten Anwendungen ihre Konfigurationsinformationen speichern. In Delphi können Sie mit dem VCL-Objekt *TRegistry* auf diese Datenbank zugreifen.

Vor Windows 95 wurden die Informationen zumeist in speziellen Dateien mit der Erweiterung INI abgelegt. Die VCL enthält die folgenden Objekte, um mit diesen Dateien zu arbeiten oder die alten Programme auf die Verwendung der Registrierung umzustellen:

- *TRegistry* ermöglicht den Zugriff auf die Registrierung.
- *TIniFile* oder *TMemIniFile* ermöglicht die Arbeit mit INI-Dateien von Windows 3.x.
- *TRegistryIniFile* ermöglicht den Zugriff auf Registrierung und INI-Dateien. Das Objekt hat ähnliche Eigenschaften und Methoden wie *TIniFile*, es liest und beschreibt aber die Registrierung. Mit Hilfe einer Variablen des Typs *TCustomIniFile* (der gemeinsamen Vorfahrklasse von *TIniFile*, *TMemIniFile* und *TRegistryIniFile*) können Sie Quelltext schreiben, in dem je nach Aufruf auf die Systemregistrierung oder auf eine INI-Datei zugegriffen wird.

Streams verwenden

Mit den spezialisierten Stream-Objekten können Informationen auf verschiedenen Speichermedien gelesen und geschrieben werden. Jede von *TStream* abgeleitete Klasse implementiert Methoden für den Zugriff auf ein bestimmtes Medium wie beispielsweise Festplattendateien, dynamischen Speicher usw. Zu diesen Klassen gehören *TFileStream*, *TStringStream*, *TMemoryStream*, *TBlobStream* und *TWinSocketStream*. Neben den Zugriffsmethoden sind auch Routinen vorhanden, um eine beliebige Position im Stream zu suchen. Die Eigenschaften von *TStream* liefern Stream-Informationen wie Größe und aktuelle Position.

Datenmodule und Remote-Datenmodule verwenden

Ein Datenmodul ist ein spezielles Formular für nichtvisuelle Komponenten. Alle Objekte in einem Datenmodul können zusätzlich zu den visuellen Steuerelementen in ein normales Formular eingefügt werden. Wenn Sie aber Gruppen von Datenbank- und Systemobjekten wiederverwenden oder die Teile der Anwendung isolieren wollen, in denen Datenbankkonnektivität und Business Rules behandelt werden, sind Datenmodule die idealen Hilfsmittel für diese Zwecke.

Es gibt zwei Arten von Datenmodulen: Standard und Remote. Standard-Datenmodule werden für ein- und zweischichtige Datenbankanwendungen verwendet. Wenn Sie jedoch mit der Client / Server- oder Enterprise-Version von Delphi mehrschichtige

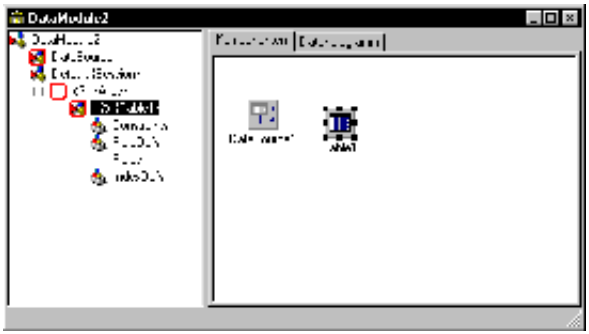
Anwendungen erstellen, können Sie Ihrem Anwendungsserver ein Remote-Datenmodul hinzufügen (siehe »Ein Remote-Datenmodul einem Anwendungsserver hinzufügen« auf Seite 2-40).

Datenmodule erstellen und bearbeiten

Sie können ein neues Datenmodul erstellen, indem Sie *Datei / Neu* wählen und dann auf den Eintrag *Datenmodul* doppelklicken. Dadurch wird ein leeres Modulfenster im Datenmodul-Designer geöffnet, die Unit des neuen Moduls im Quelltexteditor angezeigt und das Datenmodul in das aktuelle Projekt aufgenommen. Wenn Sie ein vorhandenes Datenmodul erneut öffnen, werden seine Komponenten im Designer-Fenster angezeigt.

Das Designer-Fenster ist in zwei Bereiche unterteilt. Der linke Bereich zeigt eine hierarchische Baumansicht der Komponenten des Moduls. Die rechte Ansicht enthält die zwei Registerkarten *Komponenten* und *Datendiagramm*. Erstere zeigt die Komponenten so an, wie sie in einem Formular zu sehen sind. Die Registerkarte *Datendiagramm* enthält eine grafische Darstellung der internen Beziehungen zwischen den Komponenten (z. B. Haupt / Detail-Verknüpfungen und Lookup-Felder).

Abbildung 2.5 Ein einfaches Datenmodul



Sie können Komponenten einem Datenmodul hinzufügen, indem Sie zuerst das gewünschte Objekt in der Komponentenpalette auswählen und dann in einer der beiden Ansichten des Datenmodul-Designers klicken. Ist eine Komponente im Designer ausgewählt, können ihre Eigenschaften wie bei einem Objekt in einem Formular mit Hilfe des Objektinspektors bearbeitet werden. Weitere Informationen über den Designer finden Sie in der Online-Hilfe .

Business Rules in einem Datenmodul erstellen

Sie können in der Unit-Datei nicht nur Ereignisbehandlungsroutinen für die Komponenten in einem Datenmodul erstellen, sondern auch Methoden und globale Routinen zur Implementierung von Business Rules. Sie können beispielsweise monatliche, vierteljährliche oder jährliche Abrechnungen in einer Prozedur durchführen und diese in einer Ereignisbehandlungsroutine aufrufen.

In einem Formular auf ein Datenmodul zugreifen

Um die visuellen Steuerelemente in einem Formular mit einem Datenmodul zu verbinden, müssen Sie dieses in die **uses**-Klausel der Formular-Unit aufnehmen. Dazu stehen Ihnen folgende Möglichkeiten zur Verfügung:

- Öffnen Sie die Unit des Formulars im Quelltexteditor, und fügen Sie das Datenmodul der **uses**-Klausel im **interface**-Abschnitt hinzu.
- Wählen Sie *Datei* / *Unit verwenden*, um das Dialogfeld *Unit verwenden* zu öffnen. Geben Sie anschließend den Namen des Moduls ein, oder wählen Sie den gewünschten Eintrag in der Liste.
- Doppelklicken Sie im Datenmodul auf eine *TTable*- oder *TQuery*-Komponente, um den Felder-Editor zu öffnen. Ziehen Sie anschließend die gewünschten Felder in das Formular. Es wird dann ein Meldungsfenster angezeigt, in dem Sie das Einfügen in die **uses**-Klausel des Formulars bestätigen müssen. Anschließend wird aufgrund der Informationen im Daten-Dictionary für jedes Feld automatisch ein Steuerelement (z. B. ein Eingabefeld) erstellt.

Ein Remote-Datenmodul einem Anwendungsserver hinzufügen

In manchen Versionen von Delphi können Remote-Datenmodule in Anwendungsserver-Projekten eingefügt werden. Diese Art von Datenmodul verfügt über eine Schnittstelle, auf die Client-Anwendungen in mehrschichtigen Anwendungen über das Netzwerk zugreifen können. Sie können ein Remote-Modul einem Projekt hinzufügen, indem Sie *Datei* / *Neu* wählen, die Registerkarte *Mehrschichtig* öffnen und auf das gewünschte Modul (Remote- MTS- oder CORBA-Datenmodul) doppelklicken, um den Modulexperten zu starten. Nachdem das Remote-Modul dem Projekt hinzugefügt wurde, kann es wie ein Standardmodul verwendet werden.

Weitere Informationen zu mehrschichtigen Datenbankanwendungen finden Sie in Kapitel 14, »Mehrschichtige Anwendungen erstellen«.

Die Objektablage verwenden

Mit Hilfe der Objektablage (*Tools* / *Objektablage*) können Formulare, Dialogfelder, Frames und Datenmodule auf einfache Weise projektübergreifend oder in einem Projekt gemeinsam verwendet werden. Die Ablage enthält auch Projektvorlagen, die als Vorlagen für neue Anwendungen dienen. Desweiteren sind Vorlagen für Experten gespeichert. Diese kleinen Anwendungen führen den Benutzer mit Hilfe einer Reihe von Dialogfeldern durch das Erstellen von Formularen und Projekten. Die Objektablage ist eine Textdatei mit dem Namen DELPHI32.DRO (Standardverzeichnis BIN). Sie enthält Verweise auf die Elemente, die in den Dialogfeldern Objektablage und Objektgalerie angezeigt werden.

Elemente in einem Projekt gemeinsam verwenden

Sie können Elemente in einem Projekt mehrmals verwenden, ohne sie in die Objektablage aufzunehmen. Das Dialogfeld *Objektgalerie* (*Datei* / *Neu*) enthält dazu eine Registerkarte mit dem Namen des aktuellen Projekts. Hier sind alle im Projekt erstellten Formulare, Dialogfelder, Frames und Datenmodule aufgeführt. Leiten Sie einfach ein neues Objekt von einem vorhandenen Element ab, und nehmen Sie anschließend die gewünschten Änderungen vor.

Elemente in die Objektablage aufnehmen

Sie können den bereits in der Objektablage vorhandenen Elementen jederzeit eigene Projekte, Formulare, Frames und Datenmodule hinzufügen. Gehen Sie dabei folgendermaßen vor:

1. Handelt es sich um ein Projekt oder eines seiner Elemente, öffnen Sie das betreffende Projekt.
2. Wählen Sie für ein Projekt *Projekt/Der Objektablage hinzufügen*. Bei einem Formular oder Datenmodul klicken Sie mit der rechten Maustaste auf das Element und wählen *Der Objektablage hinzufügen*.
3. Geben Sie Beschreibung, Titel und Autor in die entsprechenden Felder ein.
4. Geben Sie im Dialogfeld *Seite* an, in welche Registerkarte des Dialogfelds *Objektgalerie* das neue Element aufgenommen werden soll. Ist die angegebene Seite noch nicht vorhanden, wird sie automatisch erstellt.
5. Klicken Sie auf *Durchsuchen*, und wählen Sie ein Symbol für das neue Objekt.
6. Bestätigen Sie mit *OK*.

Objekte in einer Team-Umgebung gemeinsam verwenden

Objekte können von den Mitgliedern einer Arbeitsgruppe oder eines Entwicklungs-Teams gemeinsam verwendet werden, indem die Objektablage im Netzwerk freigegeben wird. Dazu muß für alle Personen dasselbe Verzeichnis mit der Option Gemeinsame Objektablage im Dialogfeld *Umgebungsoptionen* angegeben werden:

1. Wählen Sie *Tools* / *Umgebungsoptionen*.
2. Wechseln Sie in der Registerkarte *Präferenzen* in das Feld *Gemeinsame Objektablage*. Geben Sie das gemeinsame Verzeichnis mit der Option *Ordner* an. Vergewissern Sie sich, daß dieses Verzeichnis für alle Team-Mitglieder freigegeben ist.

Wenn Sie erstmalig ein Element in die Objektablage aufnehmen, wird automatisch die Datei DELPHI32.DRO im gemeinsamen Verzeichnis erstellt, falls sie noch nicht vorhanden ist.

Ein Element der Objektablage in einem Projekt verwenden

Sie können auf die Elemente in der Objektablage zugreifen, indem Sie *Datei / Neu* wählen. Es wird dann das Dialogfeld *Objektgalerie* mit den verfügbaren Elementen angezeigt. Sie haben je nach Objektart bis zu drei Möglichkeiten, einen Eintrag in das Projekt einzufügen:

- Kopieren
- Vererben
- Verwenden

Ein Objekt kopieren

Mit *Kopieren* können Sie eine Kopie des ausgewählten Objekts in Ihr Projekt einfügen. Wenn Sie das Objekt später in der Objektablage ändern, wirkt sich dies nicht auf die Kopie aus. Entsprechend betreffen Änderungen an der Kopie auch nicht das Original in der Objektablage.

Für Projektvorlagen ist *Kopieren* die einzige verfügbare Option.

Ein Objekt vererben

Mit *Vererben* können Sie eine neue Klasse von dem ausgewählten Objekt ableiten und in Ihr Projekt einfügen. Beim Kompilieren werden alle späteren Änderungen in der Objektablage in die abgeleitete Klasse übernommen. Änderungen der Klasse beeinflussen jedoch nicht das gemeinsame Element in der Ablage.

Vererben kann für Formulare, Dialogfelder und Datenmodule, nicht aber für Projektvorlagen verwendet werden. *Vererben* steht als einzige Option zur Wiederverwendung von Objekten in einem Projekt zur Verfügung.

Ein Objekt verwenden

Wählen Sie *Verwenden*, wenn das ausgewählte Objekt in Ihr Projekt aufgenommen werden soll. Wird das Objekt später im Projekt geändert, wirkt sich dies auf alle Elemente aus, die mit der Option *Vererben* oder *Verwenden* in andere Projekte eingefügt wurden. Setzen Sie diese Option entsprechend vorsichtig ein.

Verwenden steht für Formulare, Dialogfelder und Datenmodule zur Verfügung.

Projektvorlagen verwenden

Projektvorlagen sind vordefinierte Projekte, die Sie als Ausgangspunkt für eigene Anwendungen verwenden können. So erstellen Sie aus einer Vorlage ein neues Projekt:

1. Wählen Sie *Datei/Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.
2. Klicken Sie auf das Register *Projekte*.
3. Wählen Sie die gewünschte Projektvorlage, und bestätigen Sie mit *OK*.

4 Geben Sie im Dialogfeld *Verzeichnis auswählen* ein Verzeichnis für die Dateien des neuen Projekts an.

Die Vorlagendateien werden in das angegebene Verzeichnis kopiert, in dem sie beliebig geändert werden können. Dies wirkt sich aber nicht auf die Originalvorlage aus.

Freigegebene Objekte ändern

Wenn Sie ein Objekt in der Objektablage ändern, wirkt sich dies auf alle künftigen Projekte sowie auf die vorhandenen Projekte aus, in die das Objekt mit *Verwenden* oder *Vererben* eingefügt wurde. Sie haben folgende Möglichkeiten, dies zu verhindern:

- Fügen Sie das Objekt mit *Kopieren* ein, und ändern Sie es nur im aktuellen Projekt.
- Fügen Sie das Objekt mit *Kopieren* ein, ändern Sie es, und nehmen Sie es anschließend unter einem anderen Namen in die Objektablage auf.
- Erstellen Sie aus dem Objekt eine Komponente, DLL, Komponentenvorlage oder einen Rahmen. Eine Komponente oder DLL kann von mehreren Entwicklern gemeinsam genutzt werden.

Ein Standardelement für neue Projekte, Formulare und Hauptformulare angeben

Wenn Sie *Datei/Neue Anwendung* oder *Datei/Neues Formular* wählen, wird standardmäßig ein leeres Formular angezeigt. Diese Vorgabe kann folgendermaßen geändert werden:

1 Wählen Sie *Tools/Objektablage*.

2 Um ein Standardprojekt festzulegen, klicken Sie auf das Register *Projekte*, und wählen Sie unter *Objekte* das gewünschte Element. Aktivieren Sie anschließend das Kontrollkästchen *Neues Projekt*.

3 Um ein Standardformular festzulegen, klicken Sie zuerst auf das betreffende Register (z. B. *Formulare*), und wählen Sie anschließend unter *Objekte* das gewünschte Formular. Aktivieren Sie anschließend das Kontrollkästchen *Neues Formular*, wenn es automatisch für neue Formulare (*Datei / Neues Formular*) verwendet werden soll. Um es als Standard-Hauptformular zu verwenden, aktivieren Sie *Hauptformular*.

4 Klicken Sie auf *OK*.

Benutzerdefinierte Komponenten der IDE hinzufügen

Sie können von Ihnen oder anderen Entwicklern erstellte Komponenten in die Komponentenpalette aufnehmen und in Ihren Anwendungen verwenden. Informationen über das Erstellen von Komponenten finden Sie in Teil IV, »Benutzerdefinierte Komponenten erzeugen«. Informationen zum Installieren vorhandener Komponenten finden Sie unter »Komponenten-Packages installieren« auf Seite 9-6.

Kapitel

3

Typische Programmieraufgaben

Dieses Kapitel erläutert Grundlagen für einige der typischen Programmieraufgaben in Delphi:

- Exception-Behandlung
- Schnittstellen verwenden
- Mit Strings arbeiten
- Mit Dateien arbeiten
- Neue Datentypen definieren

Exception-Behandlung

Delphi besitzt einen Mechanismus zur konsistenten Verarbeitung von Fehlern, um die Stabilität der Anwendungen zu gewährleisten. Die Exception-Behandlung unterstützt die Wiederherstellung von Anwendungen nach Fehlern bzw. ihre Beendigung, ohne daß Daten oder Ressourcen verloren gehen. Fehlerbedingungen werden in Delphi durch Exceptions gemeldet. Dieser Abschnitt beschreibt die folgenden Aufgaben, die dazu dienen, sichere Anwendungen zu erstellen:

- Quelltextblöcke schützen
- Ressourcenzuweisungen schützen
- RTL-Exceptions behandeln
- Komponenten-Exceptions behandeln
- TApplication.HandleException verwenden
- Stille Exceptions
- Exceptions definieren

Quelltextblöcke schützen

Damit Anwendungen stabil arbeiten, muß das Auftreten von Exceptions erkannt werden. Außerdem müssen geeignete Operationen eingeleitet werden. Wenn Sie keine Reaktion definieren, zeigt die Anwendung ein Meldungsfenster an, das den Fehler beschreibt. Sie müssen die Bedingungen ermitteln, unter denen der Fehler auftritt, und geeignete Reaktionen definieren. Dies gilt insbesondere, wenn Fehler zu Datenverlusten oder zum Verlust von Systemressourcen führen können.

Die Reaktionen auf Exceptions werden in Quelltextblöcken definiert. Wenn mehrere Anweisungen eine identische Reaktion auf Fehler erfordern, können Sie diese Anweisungen in einem Block zusammenfassen und Reaktionen definieren, die für den gesamten Block gelten.

Blöcke mit spezifischen Reaktionen auf Exceptions werden als geschützte Blöcke bezeichnet, da sie einen Schutz vor Fehlern bilden, die andernfalls zum Abbruch der Anwendung oder zu Datenverlusten führen können.

Das Schützen von Quelltextblöcken setzt folgende Kenntnisse voraus:

- Auf Exceptions reagieren
- Exceptions und die Ablaufsteuerung
- Exception-Reaktionen verschachteln

Auf Exceptions reagieren

Wenn eine Fehlerbedingung auftritt, löst die Anwendung eine Exception aus, erstellt also ein Exception-Objekt. Nach dem Auslösen einer Exception kann die Anwendung Code zur Durchführung von Bereinigungen und / oder zur Behandlung der Exception ausführen.

- **Bereinigungscode ausführen**: Die einfachste Möglichkeit der Reaktion auf eine Exception ist die Ausführung von Bereinigungscode. Diese Reaktion beseitigt zwar nicht die Bedingung, die den Fehler verursacht hat, stellt aber sicher, daß die Anwendung die Betriebssystemumgebung nicht beeinträchtigt. Normalerweise werden also die zugewiesenen Ressourcen unabhängig von der Art des aufgetretenen Fehlers freigegeben.
- **Exception-Behandlung**: Dieser Begriff beschreibt die Reaktion auf eine bestimmte Exception. Bei der Behandlung einer Exception wird die Fehlerbedingung beseitigt und das Exception-Objekt freigegeben, damit die Ausführung der Anwendung fortgesetzt werden kann. Sie definieren normalerweise Exception-Behandlungsroutinen, welche die weitere Ausführung der Anwendung ermöglichen. Auf diese Weise können Sie beispielsweise Exception-Typen, wie das Öffnen nicht vorhandener Dateien, das Schreiben auf volle Datenträger oder das Berechnen von Werten außerhalb des zulässigen Bereichs verarbeiten. Während das Öffnen einer nicht vorhandenen Datei relativ problemlos abzufangen ist, können andere Exceptions wie das Schreiben auf volle Datenträger umfangreichere Korrekturmaßnahmen durch die Anwendung oder sogar durch den Benutzer erforderlich machen.

Exceptions und die Ablaufsteuerung

Object Pascal erleichtert die Fehlerbehandlung in Anwendungen, da Exceptions nicht in den normalen Ablauf der Anwendung eingreifen. Tatsächlich wird der Quelltext vereinfacht, indem die Fehlerprüfung und -behandlung außerhalb des normalen Programmablaufs implementiert wird.

Wenn Sie einen geschützten Quelltextblock deklarieren, definieren Sie spezielle Reaktionen auf Exceptions, die in diesem Block auftreten können. Tritt in diesem Block eine Exception auf, wird die Programmausführung sofort mit der definierten Reaktion fortgesetzt. Anschließend wird der Block verlassen.

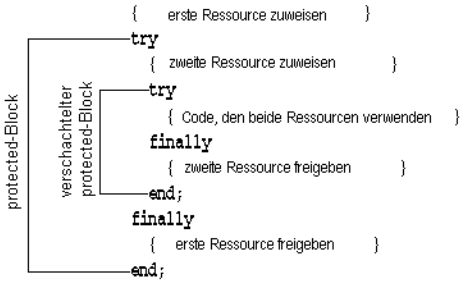
Beispiel Der folgende Quelltext enthält einen geschützten Block. Tritt in diesem geschützten Block eine Exception auf, wird die Ausführung im Abschnitt der Exception-Behandlung fortgesetzt. Die hier verwendete Anweisung gibt einen Warnton aus. Anschließend wird die Ausführung außerhalb des Blocks fortgesetzt.

```
...
try   { Beginn des geschützten Blocks }
  Font.Name := 'Courier';{ Tritt eine Exception... }
 Font.Size := 24;{ ...in einer dieser Anweisungen auf,... }
 Color := clBlue;
except   { ...wird die Ausführung hier fortgesetzt. }
  on Exception do MessageBeep(0);{ Als Reaktion auf die Exception ertönt ein Warnton }
end;
...   { Anschließend wird die Ausführung außerhalb des geschützten Blocks fortgesetzt. }
```

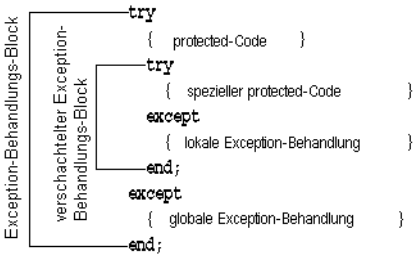
Exception-Reaktionen verschachteln

Ihr Quelltext definiert Reaktionen auf Exceptions, die in Blöcken auftreten. Da Object Pascal verschachtelte Blöcke unterstützt, können Sie Reaktionen auf Exceptions in Blöcken definieren, die sich wiederum in Blöcken mit eigenen Reaktionen auf Exceptions befinden.

Ein einfaches Beispiel ist der Schutz einer Ressourcenzuweisung. In diesem geschützten Block können Sie Blöcke definieren, die ihrerseits Ressourcen zuweisen und schützen. Daraus ergibt sich folgende Struktur:



Mit Hilfe verschachtelter Blöcke können Sie auch eine lokale Behandlung für bestimmte Exceptions definieren, die anstelle der allgemeinen Behandlung im umgebenden Block zum Tragen kommt. Daraus ergibt sich folgende Struktur:



Die verschiedenen Arten von Blöcken zur Exception-Behandlung können auch gleichzeitig eingesetzt werden, indem Sie geschützte Blöcke für Ressourcen mit Blökken zur Exception-Behandlung verschachteln (und umgekehrt).

Ressourcenzuweisungen schützen

Ein Schlüsselmerkmal stabiler Anwendungen ist die Freigabe von zugewiesenen Ressourcen auch nach dem Auftreten einer Exception. Weist die Anwendung beispielsweise Speicher zu, müssen Sie sicherstellen, daß dieser Speicher auch wieder freigegeben wird. Wird eine Datei geöffnet, muß sie später auch wieder geschlossen werden.

Beachten Sie, daß die Ursache einer Exception nicht in der Anwendung liegen muß. Der Aufruf einer Routine in einer Laufzeitbibliothek oder einer anderen Komponente kann ebenfalls eine Exception in der Anwendung auslösen. Der Quelltext der Anwendung muß die Freigabe der Ressourcen auch unter diesen Fehlerbedingungen gewährleisten.

Um Ressourcen zu schützen, benötigen Sie folgende Kenntnisse:

- Zu schützende Ressourcen
- Ressourcen-Schutzblöcke erstellen

Zu schützende Ressourcen

Unter normalen Umständen können Sie die Freigabe zugewiesener Ressourcen durch eine Anwendung gewährleisten, indem Sie Quelltext zur Zuweisung und zur Freigabe dieser Ressourcen schreiben. Sie müssen jedoch sicherstellen, daß der Code zur Freigabe der Ressourcen auch ausgeführt wird, nachdem eine Exception aufgetreten ist.

Insbesondere die folgenden Ressourcen müssen immer freigegeben werden:

- Dateien
- Speicher
- Windows-Ressourcen

- Objekte

Beispiel Die folgende Ereignisbehandlungsroutine weist Speicher zu und generiert dann einen Fehler. Der Code zur Freigabe des Speichers wird also nicht ausgeführt:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
 GetMem(APointer, 1024);    { 1 KB Speicher wird zugewiesen }
 AnInteger := 10 div ADividend;  { Löst einen Fehler aus }
 FreeMem(APointer, 1024);  { Diese Anweisung wird nie ausgeführt }
end;
```

Die meisten Fehler sind nicht so offensichtlich wie dieser hier. Trotzdem illustriert das Beispiel einen wichtigen Punkt: Wenn infolge der Division durch Null ein Fehler auftritt, wird die Ausführung außerhalb des Blocks fortgesetzt. Die *FreeMem*-Anweisung wird also nie ausgeführt.

Damit *FreeMem* den mit *GetMem* zugewiesenen Speicher freigeben kann, muß der Quelltext in einem geschützten Block stehen.

Ressourcen-Schutzblöcke erstellen

Damit zugewiesene Ressourcen auch nach dem Auftreten einer Exception freigegeben werden, müssen Sie den auf die Ressource zugreifenden Quelltext in einen geschützten Block einfügen. Die Freigabe der Ressourcen erfolgt in einem speziellen Teil dieses Blocks. Die Struktur einer geschützten Ressourcenzuweisung stellt sich folgendermaßen dar:

```
{ Ressource zuweisen }
try
  { Anweisungen, die auf die Ressource zugreifen }
finally
  { Ressource freigeben }
end;
```

Das wichtigste Merkmal des **try..finally**-Blocks ist die Tatsache, daß die Anwendung auch nach dem Auftreten einer Exception im geschützten Block die Anweisungen im **finally**-Abschnitt des Blocks ausführt. Löst eine Anweisung oder eine Routine im **try**-Abschnitt des Blocks eine Exception aus, wird die Ausführung unterbrochen. Sobald eine Exception-Behandlungsroutine gefunden wird, setzt die Anwendung die Ausführung im **finally**-Abschnitt fort. Dieser Abschnitt wird auch als Bereinigungscode bezeichnet. Nachdem dieser Abschnitt verarbeitet wurde, wird die Exception-Behandlungsroutine aufgerufen. Tritt keine Exception auf, wird der Bereinigungscode in der angegebenen Reihenfolge ausgeführt, also nach den Anweisungen im **try**-Abschnitt.

Beispiel Das folgende Beispiel zeigt eine Ereignisbehandlungsroutine, die Speicher zuweist und einen Fehler generiert, den zugewiesenen Speicher aber anschließend wieder freigibt:

```
procedure TForm1.Button1Click(Sender: TComponent);
```

```
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ 1 KB Speicher zuweisen }
  try
    AnInteger := 10 div ADividend;{ Diese Anweisung generiert einen Fehler }
  finally
    FreeMem(APointer, 1024);{ Ausführung wird trotz Fehler an dieser Stelle fortgesetzt }
  end;
end;
```

Die Anweisungen im Bereinigungscode sind nicht vom Auftreten einer Exception abhängig. Auch wenn keine der Anweisungen im **try**-Abschnitt eine Exception auslöst, wird der Bereinigungscode ausgeführt.

RTL-Exceptions behandeln

Wenn Sie Quelltext schreiben, der Routinen in der Laufzeitbibliothek (Runtime Library = RTL) aufruft (beispielsweise mathematische Funktionen oder Prozeduren zur Dateiverarbeitung), meldet die RTL Fehler in Form von Exceptions an die Anwendung. Normalerweise generieren RTL-Exceptions eine Meldung, die dem Benutzer von der Anwendung angezeigt wird. Sie können Exception-Behandlungsroutinen definieren, die RTL-Exceptions auf andere Art verarbeiten.

Neben den genannten Exceptions können sogenannte »stille« Exceptions auftreten, die standardmäßig keine Meldung anzeigen.

Zur Behandlung von RTL-Exceptions benötigen Sie folgende Kenntnisse:

- RTL-Exceptions
- Exception-Behandlungsroutine erstellen
- Exception-Behandlungsanweisungen
- Exception-Instanz verwenden
- Gültigkeitsbereich von Exception-Behandlungsroutinen
- Standard-Exception-Behandlungsroutinen bereitstellen
- Exception-Klassen verarbeiten
- Exception erneut auslösen

RTL-Exceptions

Die Exceptions der Laufzeitbibliothek sind in der Unit *SysUtils* definiert und vom generischen Exception-Objekt *Exception* abgeleitet. Exception stellt einen String bereit, der standardmäßig von RTL-Exceptions angezeigt wird.

Die RTL kann verschiedene Typen von Exceptions auslösen, die in der folgenden Tabelle beschrieben werden.

Tabelle 3.1 RTL-Exceptions

Fehlertyp	Ursache	Beschreibung
Eingabe / Ausgabe	Fehler beim Zugriff auf eine Datei oder ein E / A-Gerät	Die meisten E / A-Fehler beziehen sich auf Fehlercodes, die von Windows beim Zugriff auf eine Datei zurückgegeben werden.
Heap	Fehler beim Zugriff auf den dynamischen Speicher	Heap-Fehler können auftreten, wenn nicht genügend Speicher verfügbar ist oder die Anwendung einen Zeiger verwendet, der auf eine Position außerhalb des Heap zeigt.
Integer-Berechnung	Unzulässige Operationen in Integer-Ausdrücken	Beispiele sind die Division durch Null, Werte oder Ausdrücke außerhalb des zulässigen Bereichs und Überläufe.
Gleitkomma-Berechnung	Unzulässige Operationen in Real-Ausdrükken	Gleitkommafehler können vom Coprozessor (Hardware) oder vom Software-Emulator ausgelöst werden. Beispiele sind ungültige Anweisungen, die Division durch Null sowie Über- und Unterläufe.
Typumwandlung	Unzulässige Typumwandlung mit dem Operator **as**	Die Typumwandlung für Objekte ist nur in kompatible Typen zulässig.
Konvertierung	Unzulässige Typkonvertierung	Funktionen zur Typkonvertierung wie *IntToStr*, *StrToInt* und *StrToFloat* lösen eine Konvertierungs-Exception aus, wenn der Parameter nicht in den gewünschten Typ konvertiert werden kann.
Hardware	Systembedingung	Hardware-Exceptions weisen auf Fehlerbedingungen oder Unterbrechungen hin, die vom Prozessor oder vom Benutzer generiert wurden, z. B. Zugriffsverletzungen, Stack-Überläufe oder Tastatureingriffe.
Variante	Unzulässige Typzuweisung	Bei der Referenzierung von Varianten in Ausdrücken können Fehler auftreten, wenn die Zuweisung eines kompatiblen Typs an die Variante nicht möglich ist.

Eine Liste der RTL-Exceptions finden Sie in der Unit *SysUtils*.

Exception-Behandlungsroutine erstellen

Eine Exception-Behandlungsroutine besteht aus Quelltext, der eine bestimmte Exception oder die in einem geschützten Quelltextblock auftretenden Exceptions verarbeitet.

Die Definition einer Exception-Behandlungsroutine erfolgt durch Einbetten des zu schützenden Quelltextes in einen Exception-Behandlungsblock. Die Anweisungen zur Exception-Behandlung werden in den **except**-Abschnitt des Blocks eingefügt. Die Struktur eines typischen Exception-Behandlungsblocks stellt sich also folgendermaßen dar:

```
   try
  { Zu schützende Anweisungen }
except
  { Anweisungen zur Exception-Behandlung }
end;
```

Die Anwendung führt die Anweisungen im **except**-Abschnitt nur aus, wenn während der Ausführung der Anweisungen im **try**-Abschnitt eine Exception auftritt. Zur Ausführung der Anweisungen im **try**-Abschnitt gehört hier auch die Ausführung der Routinen, die von diesen Anweisungen aufgerufen werden. Rufen Anweisungen im **try**-Abschnitt also eine Routine auf, für die keine Exception-Behandlungsroutine definiert ist, wird die Ausführung zur Behandlung der Exception an diesen Exception-Behandlungsblock zurückgegeben.

Löst eine Anweisung im **try**-Abschnitt eine Exception aus, wird die Ausführung mit dem except-Abschnitt fortgesetzt. Die enthaltenen Anweisungen zur Exception-Behandlung bzw. die Exception-Behandlungsroutinen werden durchsucht, bis eine für die aktuelle Exception geeignete Behandlungsroutine gefunden wird.

Sobald die Anwendung eine Exception-Behandlungsroutine findet, die zur Behandlung der Exception geeignet ist, wird diese Anweisung ausgeführt. Anschließend gibt die Anwendung das Exception-Objekt automatisch frei. Die Ausführung der Anwendung wird dann am Ende des aktuellen Blocks fortgesetzt.

Exception-Behandlungsanweisungen

Jede **on**-Anweisung im **except**-Abschnitt eines **try..except**-Blocks definiert die Behandlung eines bestimmten Exception-Typs. Die Struktur dieser Anweisungen zur Exception-Behandlung sieht folgendermaßen aus:

```
on <Exception-Typ> do <Anweisung>;
```

Beispiel Sie können eine Exception-Behandlungsroutine bereitstellen, die für eine Division durch Null ein Standardergebnis bereitstellt:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
 try
    Result := Sum div NumberOfItems;{ Normalfall }
 except
    on EDivByZero do Result := 0;{ Exception-Behandlung, sofern erforderlich }
 end;
end;
```

Dieses Vorgehen ist übersichtlicher als die Abfrage, ob der Divisor Null ist, sobald die Funktion aufgerufen wird. Die folgende äquivalente Funktion nutzt die strukturellen Vorteile der Exceptions nicht:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
 if NumberOfItems <> 0 then{ Immer testen }
    Result := Sum div NumberOfItems{ Normalfall }
  else Result := 0;{ Exception-Behandlung }
end;
```

Diese beiden Funktionen verdeutlichen den Unterschied zwischen der Programmierung mit und ohne Exceptions. Das Beispiel ist sehr einfach, die Struktur bleibt jedoch identisch, wenn komplexe Berechnungen mit hunderten von Schritte durchgeführt werden, bei denen Exceptions auftreten können, wenn nur eine von Dutzenden Eingaben ungültig ist.

Mit Exceptions können Sie den eigentlichen Ausdruck für einen Algorithmus definieren und anschließend den Code für die Ausnahmefälle bereitstellen, in denen dieser Code nicht zur Verarbeitung ausreicht. Ohne Exceptions müssen Sie dagegen jedesmal die Eingabe testen, um sicherzustellen, daß der nächste Schritt in der Berechnung ausgeführt werden kann, ohne Fehler auszulösen.

Exception-Instanz verwenden

Normalerweise benötigt eine Exception-Behandlungsroutine keinerlei Informationen über Exceptions, für deren Behandlung andere Routinen definiert wurden. Die Anweisungen nach on..do sind also für den jeweiligen Exception-Typ spezifisch. In bestimmten Fällen werden jedoch einige dieser Informationen in der Exception-Instanz benötigt.

Mit einer speziellen Form der on..do-Struktur erhalten Sie Zugriff auf die Exception-Instanz und können diese Informationen in einer Exception-Behandlungsroutine lesen. Diese spezielle Form setzt eine temporäre Variable voraus, in der die Instanz gespeichert werden kann.

Beispiel Wenn Sie ein neues Projekt mit einem einzelnen Formular erstellen, können Sie eine Bildlaufleiste und eine Schaltfläche in das Formular einfügen. Klicken Sie doppelt auf diese Schaltfläche, und fügen Sie die folgende Zeile in deren Ereignisbehandlungsroutine für Mausklicks ein:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

Diese Zeile löst eine Exception aus, weil der Maximalwert einer Bildlaufleiste immer größer als der Minimalwert sein muß. Die Standard-Exception-Behandlungsroutine dieser Anwendung öffnet ein Dialogfeld mit der Meldung, die das Exception-Objekt enthält. Sie können die Exception-Behandlung in dieser Routine überschreiben und ein eigenes Meldungsfenster erstellen, das den Meldungsstring der Exception anzeigt:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

Die temporäre Variable (*E* in diesem Beispiel) besitzt den Typ, der hinter dem Doppelpunkt angegeben wird (*EInvalidOperation* in diesem Beispiel). Sie können den Operator as verwenden, um der Exception gegebenenfalls einen spezifischeren Typ zuzuweisen.

Hinweis Geben Sie das temporäre Exception-Objekt nicht explizit frei. Die Freigabe erfolgt automatisch bei der Behandlung der Exception. Wenn Sie das Objekt explizit freigeben,

tritt bei der automatischen Freigabe des (nicht mehr vorhandenen) Objekts durch die Anwendung eine Zugriffsverletzung auf.

Gültigkeitsbereich von Exception-Behandlungsroutinen

Sie müssen nicht in jedem Block Behandlungsroutinen für jeden Exception-Typ bereitstellen. Normalerweise stellen Sie nur Behandlungsroutinen für die Exceptions bereit, die Sie explizit im jeweiligen Block verarbeiten wollen.

Kann ein Block eine bestimmte Exception nicht behandeln, wird die Ausführung an den umgebenden Block übergeben (bzw. an den Code, der diesen Block aufgerufen hat). Die Exception liegt dabei weiterhin vor. Dieser Prozeß der Erweiterung des Gültigkeitsbereichs wird fortgesetzt, bis entweder der höchste Punkt in der Anwendungshierarchie erreicht ist oder ein Block auf einer beliebigen übergeordneten Ebene die Exception behandeln kann.

Standard-Exception-Behandlungsroutinen bereitstellen

Sie können eine einzelne Standard-Behandlungsroutine für alle Exceptions bereitstellen, für die Sie keine speziellen Behandlungsroutinen definieren. Zu diesem Zweck fügen Sie einen else-Abschnitt in den **except**-Abschnitt des Exception-Behandlungsblocks ein:

```
try
  { Anweisungen }
except
  on ESomething do { Spezifischer Code zur Exception-Behandlung };
 else { Code zur Standard-Exception-Behandlung };
end;
```

Indem Sie eine Standard-Exception-Behandlungsroutine in einen Block einfügen, wird sichergestellt, daß dieser Block jede auftretende Exception behandelt. Alle Behandlungsroutinen des umgebenden Blocks werden also überschrieben.

Achtung Der Einsatz dieser »allgemeinen« Exception-Behandlungsroutinen ist nicht empfehlenswert. Die **else**-Klausel verarbeitet alle Exceptions, also auch die, zu denen keinerlei Informationen vorliegen. Grundsätzlich sollten Anwendungen nur Exceptions behandeln, deren Behandlung dem Programmierer geläufig ist. Wollen Sie nur Bereinigungsarbeiten durchführen und die Behandlung dem Quelltext überlassen, der weitere Informationen zur Exception und deren Behandlung besitzt, können Sie einen umgebenden **try..finally**-Block verwenden:

```
try
      try
  { Anweisungen }
      except
    on ESomething do { Spezifischer Code zur Exception-Behandlung };
      end;
finally
{ Bereinigungscode };
end;
```

Ein anderer Ansatz der Exception-Behandlung wird weiter unten im Abschnitt »Exception erneut auslösen« beschrieben.

Exception-Klassen verarbeiten

Da Exception-Objekte Teil einer Hierarchie sind, können Sie spezielle Behandlungsroutinen für ganze Teile dieser Hierarchie erstellen. Zu diesem Zweck stellen Sie eine Behandlungsroutine für die Exception-Klasse bereit, von der der betreffende Teil der Hierarchie abgeleitet ist.

Beispiel Der folgende Block zeigt die Struktur einer Routine zur Behandlung von Exceptions, die von Integer-Operationen ausgelöst werden:

```
try
  { Anweisungen mit Integer-Operationen }
except
  on EIntError do { Behandlung von Fehlern bei Integer-Operationen };
end;
```

Sie können natürlich weiterhin für einzelne Exceptions spezifische Behandlungsroutinen erstellen. Diese Routinen müssen sich aber vor der allgemeinen Behandlungsroutine befinden, da die Anwendung die Behandlungsroutinen in der angegebenen Reihenfolge durchsucht und die erste geeignete Behandlungsroutine ausführt. Der folgende Block definiert beispielsweise eine spezielle Behandlung für Bereichsfehler. Alle anderen Fehler bei Integer-Operationen werden mit einer allgemeinen Behandlungsroutine abgefangen:

```
try
  { Anweisungen mit Integer-Operationen }
except
  on ERangeError do { Bereichsfehlerbehandlung };
 on EIntError do { Behandlung anderer Fehler bei Integer-Operationen };
end;
```

Wird die Behandlungsroutine für *EIntError* vor der Behandlungsroutine für *ERangeError* definiert, erreicht die Anwendungsausführung nie die spezifische Behandlungsroutine für *ERangeError*.

Exception erneut auslösen

In bestimmten Fällen ist es auch bei einer lokalen Behandlung von Exceptions erforderlich, die Behandlung an den umgebenden Block weiterzugeben. Nachdem die lokale Behandlungsroutine die Behandlung beendet hat, gibt sie die Exception-Instanz frei. Die Behandlungsroutine des umgebenden Blocks kann also nicht eingreifen. Sie können jedoch verhindern, daß die Behandlungsroutine die Exception freigibt und so sicherstellen, daß auch die Behandlungsroutine des umgebenden Blocks auf die Exception reagieren kann.

Beispiel Wenn eine Exception auftritt, können Sie dem Benutzer eine Meldung anzeigen und anschließend mit der Standardbehandlung fortfahren. Zu diesem Zweck deklarieren Sie eine lokale Exception-Behandlungsroutine, die diese Meldung anzeigt und anschließend das reservierte Wort raise verwendet. Diese Operation wird als »erneutes Auslösen« der Exception bezeichnet (siehe Beispiel):

```
try
  { Anweisungen }
 try
```

```
    { Spezielle Anweisungen }
  except
    on ESomething do
    begin
      { Behandlung nur der speziellen Anweisungen }
     raise; { Erneutes Auslösen der Exception }
    end;
      end;
except
  on ESomething do ...;{ Die für alle verbleibenden Fälle gewünschte Behandlung }
end;
```

Wenn Quelltext im Abschnitt { Anweisungen } eine *ESomething*-Exception auslöst, wird nur die Behandlungsroutine im äußeren **except**-Abschnitt ausgeführt. Löst jedoch der Abschnitt { Spezielle Anweisungen } eine *ESomething*-Exception aus, wird zunächst die Behandlung im inneren **except**-Abschnitt und anschließend die allgemeinere Behandlung im äußeren **except**-Abschnitt ausgeführt.

Durch erneutes Auslösen von Exceptions können problemlos spezielle Behandlungen für Exceptions in Sonderfällen definiert werden, ohne daß die existierenden Behandlungsroutinen überschrieben oder dupliziert werden.

Komponenten-Exceptions behandeln

Delphi-Komponenten lösen Exceptions aus, um Fehlerbedingungen anzuzeigen. Die meisten Komponenten-Exceptions weisen auf Programmierfehler hin, die andernfalls Laufzeitfehler generieren würden. Grundsätzlich besteht kein Unterschied zwischen der Behandlung von Komponenten-Exceptions und der Behandlung von RTL-Exceptions.

Beispiel Eine häufige Fehlerursache in Komponenten sind Bereichsfehler bei indizierten Eigenschaften. Enthält ein Listenfeld beispielsweise drei Elemente (0..2) und greift die Anwendung auf das Element mit dem Index 3 zu, löst das Listenfeld wegen der Bereichsüberschreitung eine Exception aus.

Die folgende Ereignisbehandlungsroutine enthält eine Exception-Behandlungsroutine, die den Benutzer in einem Fenster auf den ungültigen Indexzugriff im Listenfeld hinweist:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('Ein String');{ String in Listenfeld einfügen }
      ListBox1.Items.Add('Noch ein String');{ Zweiten String einfügen... }
  ListBox1.Items.Add('und noch ein String');{ ...Dritten String einfügen }
 try
    Caption := ListBox1.Items[3];{ Der Überschrift wird der vierte String zugewiesen }
 except
    on EStringListError do
      MessageDlg('Listenfeld enthält weniger als vier Strings', mtWarning, [mbOK], 0);
 end;
end;
```

Wenn Sie einmal auf die Schaltfläche klicken, enthält die Liste nur drei Strings. Der Zugriff auf den vierten String (*Items[3]*) löst deshalb eine Exception aus. Sobald Sie

ein zweites Mal auf die Schaltfläche klicken, werden weitere Strings in die Liste eingefügt. Es kommt also nicht mehr zu einer Exception.

TApplication.HandleException verwenden

HandleException stellt die Standardbehandlung für Exceptions in einer Anwendung bereit. Wenn die Exception durch alle **try**-Blöcke im Anwendungsquelltext weitergegeben wird, ruft die Anwendung automatisch die Methode *HandleException* auf. Diese Methode zeigt ein Fenster mit dem Hinweis an, daß ein Fehler aufgetreten ist. Sie können *HandleException* folgendermaßen einsetzen:

```
try
  { Anweisungen }
except
  Application.HandleException(Self);
end;
```

Für alle Exceptions mit Ausnahme von *EAbort* ruft *HandleException* die Ereignisbehandlungsroutine für *OnException* auf, wenn diese vorhanden ist. Soll die Exception also behandelt und gleichzeitig dieses Standardverhalten bereitgestellt werden (wie bei der VCL), können Sie einen Aufruf der Methode *HandleException* in Ihren Quelltext einfügen:

```
try
   { Spezielle Anweisungen }
except
   on ESomething do
   begin
     { Nur die speziellen Anweisungen werden verarbeitet }
     Application.HandleException(Self);{ Aufruf von HandleException }
   end;
end;
```

Weitere Informationen finden Sie im Index der Online-Hilfe unter Exception-Behandlungsroutinen.

Stille Exceptions

Delphi-Anwendungen zeigen für die meisten Exceptions, die im Quelltext nicht speziell berücksichtigt wurden, ein Meldungsfenster mit dem Meldungsstring aus dem Exception-Objekt an. Sie können aber auch »stille« Exceptions definieren, die standardmäßig nicht zur Anzeige einer Fehlermeldung führen.

Stille Exceptions sind hilfreich, wenn eine Exception nicht behandelt, die betreffende Operation aber abgebrochen werden soll. Das Abbrechen einer Operation entspricht der Verwendung der Prozeduren *Break* bzw. *Exit* in einem Block, kann jedoch zur Fortsetzung des Abbruchs durch mehrere Ebenen verschachtelter Blöcke führen.

Stille Exceptions sind vom Exception-Standardtyp *EAbort* abgeleitet. Die Standard-Exception-Behandlungsroutine für VCL-Anwendungen zeigt das Dialogfeld mit der Fehlermeldung für alle eingehenden Exceptions außer denen an, die von *EAbort* abgeleitet wurden.

Hinweis In Konsolenanwendungen wird für eine *EAbort*-Exception ein Dialogfeld mit einer Fehlermeldung angezeigt.

Zum Auslösen stiller Exceptions können Sie eine schnellere Methode verwenden. In diesem Fall wird das Objekt nicht manuell erstellt. Statt dessen rufen Sie die Prozedur *Abort* auf. *Abort* löst automatisch eine *EAbort*-Exception aus, die ohne Anzeige einer Fehlermeldung zum Abbruch der aktuellen Operation führt.

Beispiel Der folgende Code zeigt ein einfaches Beispiel zum Abbruch einer Operation. In einem Formular mit einem leeren Listenfeld und einer Schaltfläche kann der folgende Quelltext für das Ereignis *OnClick* der Schaltfläche verwendet werden:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
 for I := 1 to 10 do{ Schleife zehnmal verarbeiten }
  begin
   ListBox1.Items.Add(IntToStr(I));{ Zahl in die Liste einfügen }
   if I = 8 then Abort;{ Nach der achten Zahl abbrechen }
 end;
end;
```

Exceptions definieren

Sie können Code nicht nur vor Exceptions schützen, die von der Laufzeitbibliothek und verschiedenen Komponenten generiert werden, sondern diesen Mechanismus auch nutzen, um Ausnahmebedingungen in Ihrem eigenen Quelltext zu verarbeiten.

Zum Einsatz von Exceptions im Quelltext benötigen Sie die folgenden Kenntnisse:

- Exception-Objekttyp deklarieren
- Exception auslösen

Exception-Objekttyp deklarieren

Da Exceptions als Objekte vorliegen, können neue Exceptions ebenso wie neue Objekttypen deklariert werden. Obwohl Sie jede Objektinstanz als Exception verwenden können, verarbeiten die Standard-Behandlungsroutinen für Exceptions nur die von *Exception* abgeleiteten Exceptions.

Aus diesem Grund bietet es sich an, neue Exception-Typen von *Exception* oder einer der anderen Standard-Exceptions abzuleiten. So wird gewährleistet, daß auch beim Auslösen einer neuen Exception in einem nicht durch eine spezielle Behandlungsroutine geschützten Block eine der Standard-Behandlungsroutinen diese Exception verarbeiten kann.

Beispiel Als Beispiel dient die folgende Deklaration:

```
type
  EMyException = class(Exception);
```

Wenn Sie *EMyException* auslösen, ohne eine spezifische Behandlungsroutine bereitzustellen, wird diese Exception von der Behandlungsroutine für *Exception* (oder einer

Standard-Exception-Behandlungsroutine) verarbeitet. Da die Standardbehandlung für Exception in der Anzeige des Namens der ausgelösten Exception besteht, können Sie die ausgelöste Exception problemlos ermitteln.

Exception auslösen

Sie können eine Exception auslösen, um eine Fehlerbedingung in einer Anwendung zu signalisieren. Zu diesem Zweck wird eine Instanz des gewünschten Typs erstellt und das reservierte Wort **raise** verwendet.

Das Auslösen einer Exception erfolgt mit dem reservierten Wort **raise** und der Angabe einer Instanz des Exception-Objekts. Wenn eine Exception-Behandlungsroutine diese Exception verarbeitet, wird die Exception-Instanz nach der Behandlung freigegeben. Die Freigabe muß also nicht manuell erfolgen.

Die Exception-Adresse wird mit der Variablen *ErrorAddr* in der Unit *System* bereitgestellt. Beim Auslösen einer Exception wird dieser Variablen die Adresse zugewiesen, an der die Anwendung die Exception ausgelöst hat. Sie können in Exception-Behandlungsroutinen auf *ErrorAddr* zugreifen, um beispielsweise dem Benutzer mitzuteilen, wo der Fehler aufgetreten ist. Sie können also einen Wert für *ErrorAddr* angeben, wenn Sie eine Exception auslösen.

Die Angabe der Fehleradresse für eine Exception erfolgt mit dem reservierten Wort **at** hinter der Exception-Instanz sowie einem Adreßausdruck (beispielsweise einem Bezeichner).

Ein Beispiel bildet die folgende Deklaration:

```
type
  EPasswordInvalid = class(Exception);
```

Sie können jederzeit eine Exception wegen eines ungültigen Kennworts mit einer Instanz von *EPasswordInvalid* auslösen:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Falsches Kennwort');
```

Schnittstellen verwenden

Das Delphi-Schlüsselwort **interface** erlaubt in Anwendungen das Erstellen und Verwenden von Schnittstellen. Schnittstellen erweitern das Modell der Einfachvererbung der VCL, indem einzelne Klassen mehr als eine Schnittstelle implementieren können. Gleichzeitig können mehrere Klassen mit unterschiedlichen Basisklassen gemeinsam auf eine Schnittstelle zugreifen. Schnittstellen sind hilfreich, wenn beispielsweise Stream-Operationen für viele verschiedene Objekte erforderlich sind. Außerdem bilden Schnittstellen eine der Grundlagen der verteilten Objektmodelle COM (Component Object Model) und CORBA (Common Object Request Broker Architecture).

Schnittstellen als Sprachmerkmal

Eine Schnittstelle ähnelt einer Klasse, die nur abstrakte Methoden und eine genaue Definition ihrer Funktionalität enthält. Die Definitionen der Schnittstellenmethoden enthalten also Anzahl und Typ der Parameter, den Rückgabetyp und das erweiterte Verhalten. Schnittstellenmethoden stehen in einem logischen Verhältnis zum Zweck der Schnittstelle. Per Konvention werden Schnittstellen nach Maßgabe ihres Verhaltens benannt. Dem Namen wird ein *I* vorangestellt. Eine Schnittstelle namens *IMalloc* weist beispielsweise Speicher zu, verwaltet ihn und gibt ihn wieder frei. Eine Schnittstelle namens *IPersist* könnte dagegen die Basisschnittstelle für eine Reihe von Nachkommen bilden, die ihrerseits spezifische Methodenprototypen zum Laden und Speichern des Status von Objekten im Speicher, in Streams oder in Dateien definieren. Nachstehend finden Sie ein einfaches Beispiel für die Deklaration einer Schnittstelle:

```
type
IEdit = interface
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

Schnittstellen können ebensowenig wie abstrakte Klassen instantiiert werden. Der Einsatz einer Schnittstelle setzt also deren Ableitung aus einer implementierenden Klasse voraus.

Zur Implementierung einer Schnittstelle definieren Sie eine Klasse, in deren Vorfahrenliste die Schnittstelle deklariert wird. Dadurch werden alle Methoden dieser Schnittstelle implementiert:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

Schnittstellen definieren das Verhalten und die Signaturen der enthaltenen Methoden, nicht jedoch deren Implementierungen. Wenn die Implementierung der Klasse der Schnittstellendefinition entspricht, ist die Schnittstelle vollständig polymorph: Zugriff und Verwendung der Schnittstelle ist also für alle Implementierungen dieser Schnittstelle identisch.

Schnittstellen in mehreren Klassen nutzen

Mit Schnittstellen eröffnet sich eine neue Möglichkeit zur Trennung von Einsatz und Implementierung einer Klasse. Zwei Klassen können dieselbe Schnittstelle verwenden, auch wenn sie keine Nachkommen derselben Basisklasse sind. Dieser polymorphe Aufruf von Methoden für voneinander unabhängige Objekte ist möglich, wenn diese Objekte dieselbe Schnittstelle implementieren. Als Beispiel dient die folgende Deklaration einer Schnittstelle

```
IPaint = interface
  procedure Paint;
end;
```

und zweier Klassen:

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Auch wenn die beiden Klassen unterschiedliche Vorfahren besitzen, sind sie doch mit der Variablen *IPaint* zuweisungskompatibel:

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

Sie können dies beispielsweise auch durch Ableitung von *TCircle* und *TSquare* aus der Klasse *TFigure* erreichen, die eine virtuelle Methode namens *Paint* implementiert. *TCircle* und *TSquare* würden in diesem Fall die Methode *Paint* überschreiben. *IPaint* im Beispiel müßte durch *TFigure* ersetzt werden. Beachten Sie aber die folgende Schnittstelle:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

Hier ist es sinnvoll, das Rechteck, nicht aber den Kreis zu unterstützen. Die Klasse würde so aussehen:

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Anschließend können Sie eine Klasse namens *TFilledCircle* erstellen, welche die Schnittstelle *IRotate* implementiert, um für den Kreis eine Drehung des Füllmusters zu ermöglichen, ohne daß der Kreis selbst gedreht werden muß.

Hinweis In diesen Beispielen ist unterstellt, daß die Basisklasse oder ein Vorfahr die Methoden von *IUnknown* zur Referenzzählung implementiert. Weitere Informationen finden Sie unter »IUnknown implementieren« auf Seite 3-18 und »Speicherverwaltung für Schnittstellenobjekte« auf Seite 3-22.

Schnittstellen mit Prozeduren verwenden

Schnittstellen ermöglichen das Schreiben generischer Prozeduren zur Verarbeitung von Objekten, die keine Nachkommen derselben Basisklasse sein müssen. Mit den oben verwendeten Schnittstellen *IPaint* und *IRotate* können die folgenden Prozeduren geschrieben werden:

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects setzt nicht voraus, daß Objekte die Informationen für ihre Bildschirmdarstellung besitzen, PaintObjects setzt nicht voraus, daß Objekte die zum Drehen erforderlichen Informationen besitzen. Dadurch können diese Objekte wesentlich öfter genutzt werden, als dies der Fall ist, wenn sie explizit für eine *TFigure*-Klasse geschrieben werden.

Weitere Informationen über Syntax, Sprachdefinitionen und Regeln für Schnittstellen finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Objektschnittstellen«.

IUnknown implementieren

Alle Schnittstellen werden direkt oder indirekt von der Schnittstelle *IUnknown* abgeleitet. Diese Schnittstelle stellt die grundlegende Funktionalität von Schnittstellen bereit, also die dynamischen Abfragen und die Referenzverwaltung. Diese Funktion wird in den drei Methoden von *IUnknown* implementiert:

- *QueryInterface* ist eine Methode zur dynamischen Abfrage von Objekten und zum Abrufen von Schnittstellenreferenzen für die vom Objekt unterstützten Schnittstellen.
- *AddRef* ist eine Methode zur Referenzzählung, deren Wert mit jedem erfolgreichen Aufruf von *QueryInterface* erhöht wird. Solange der Referenzzähler einen Wert ungleich Null aufweist, darf das Objekt nicht aus dem Speicher entfernt werden.
- *Release* wird in Verbindung mit *AddRef* verwendet, damit Objekte ermitteln können, ob sie aus dem Speicher entfernt werden dürfen. Sobald der Referenzzähler Null erreicht, gibt die Schnittstellenimplementierung das Objekt frei.

Wenn eine Klasse Schnittstellen implementiert, muß sie auch die drei *IUnknown*-Methoden und alle Methoden implementieren, die von anderen Vorfahrenschnittstellen

und den betreffenden Schnittstellen selbst deklariert werden. Sie können die Implementierungen der Schnittstellenmethoden in einer Klasse auch vererben.

TInterfacedObject

Die VCL definiert eine einfache Klasse namens *TInterfacedObject*, die hervorragend als Basisklasse eingesetzt werden kann, da sie die Methoden der Schnittstelle *IUnknown* implementiert. Die Klasse *TInterfacedObject* ist in der Unit *System* folgendermaßen deklariert:

```
type
  TInterfacedObject = class(TObject, IUnknown)
  private
    FRefCount: Integer;
 protected
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    property RefCount: Integer read FRefCount;
  end;
```

Das Ableiten eigener Schnittstellen von *TInterfacedObject* ist problemlos möglich. In der folgenden Beispieldeklaration ist *TDerived* als direkter Nachkomme von *TInterfacedObject* abgeleitet und implementiert die hypothetische Schnittstelle *IPaint*.

```
type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;
```

Da die Klasse *TInterfacedObject* die Methoden von *IUnknown* implementiert, führt sie automatisch die Referenzzählung und die Speicherverwaltung für Schnittstellenobjekte durch. Weitere Informationen finden Sie im Abschnitt »Speicherverwaltung für Schnittstellenobjekte« auf Seite 3-22. In diesem Abschnitt wird auch beschrieben, wie Sie eigene Klassen erstellen, die Schnittstellen implementieren, aber nicht die Referenzzählung von *TInterfacedObject* nutzen.

Operator as verwenden

Wenn Klassen Schnittstellen implementieren, können sie den Operator **as** für die dynamische Bindung an die Schnittstelle verwenden. Ein Beispiel:

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;

begin
  X := P as IPaint;
{ statements }
end;
```

Die Variable *P* des Typs *TInterfacedObject* kann der Variablen *X* zugewiesen werden, bei der es sich um eine *IPaint*-Schnittstellenreferenz handelt. Die Zuweisung wird durch die dynamische Bindung ermöglicht. Für diese Zuweisung generiert der Compiler Code, der die Methode *QueryInterface* der Schnittstelle *IUnknown* von *P* aufruft, da der Compiler dem deklarierten Typ *P* nicht entnehmen kann, ob die Instanz von *P* tatsächlich *IPaint* unterstützt. Zur Laufzeit wird entweder *P* als *IPaint*-Referenz aufgelöst, oder es wird eine Exception ausgelöst. In beiden Fällen wird durch die Zuweisung von *P* an *X* kein Compiler-Fehler generiert. Es wird jedoch ein Compiler-Fehler generiert, wenn *P* die Klasse eines Typs darstellt, der *IUnknown* nicht implementiert.

Wenn Sie den Operator as für die dynamische Bindung an eine Schnittstelle verwenden, sollten Sie die folgenden Anforderungen berücksichtigen:

- Explizite Deklaration von *IUnknown*: Obwohl alle Schnittstellen von *IUnknown* abgeleitet sind, reicht es für den Einsatz des Operators **as** nicht aus, wenn eine Klasse die Methoden von *IUnknown* implementiert. Dies gilt auch, wenn zusätzlich die explizit deklarierten Schnittstellen implementiert werden. Die Klasse muß *IUnknown* vielmehr explizit in der Vorfahrenliste deklarieren.
- Verwenden einer IID: Schnittstellen können einen Bezeichner verwenden, der auf einer GUID (Globally Unique Identifier = global eindeutiger Bezeichner) basiert. GUIDs zur Bezeichnung von Schnittstellen werden Schnittstellenbezeichner (Interface Identifiers = IIDs) genannt. Wenn Sie den Operator **as** mit einer Schnittstelle einsetzen, muß diese eine zugeordnete IID besitzen. Mit dem Editor-Tastenkürzel *Strg+Umschalt+G* können Sie im Quelltext eine neue GUID erstellen.

Wiederverwendung von Quelltext und Delegation

Eine Möglichkeit der Wiederverwendung von Quelltext in Schnittstellen besteht darin, ein Objekt in die Schnittstelle aufzunehmen bzw. die Schnittstelle als Objekt in eine andere einzufügen. Die VCL verwendet zu diesem Zweck Eigenschaften, die Objekttypen darstellen. Zur Unterstützung dieser Struktur für Schnittstellen stellt Delphi das Schlüsselwort **implements** bereit, mit dem die Implementierung einer Schnittstelle ganz oder teilweise einem untergeordneten Objekt übertragen werden kann (Delegation). Die Aggregation ist eine andere Möglichkeit, Quelltext über diese Mechanismen wiederzuverwenden. Bei der Aggregation enthält ein übergeordnetes Objekt ein untergeordnetes. Das untergeordnete Objekt implementiert Schnittstellen, die nur für dieses übergeordnete Objekt verfügbar sind. Die VCL enthält beispielsweise Klassen, welche die Aggregation unterstützen.

implements für die Delegation verwenden

Viele Klassen der VCL besitzen Eigenschaften, die untergeordnete Objekte darstellen. Sie können auch Schnittstellen als Eigenschaftstyp verwenden. Wird einer Eigenschaft der Typ **interface** zugewiesen (oder der Typ einer Klasse, in der die Methoden einer Schnittstelle implementiert werden), können Sie mit dem Schlüsselwort implements angeben, daß die Methoden dieser Schnittstelle an die Objekt- oder Schnittstellenreferenz delegiert werden, welche die Eigenschaftsinstanz repräsentiert. Das Objekt bzw. die Schnittstelle muß nur die Implementierung der Methoden bereitstellen, nicht die Deklaration der Schnittstellenunterstützung. Die Klasse mit der Eigen-

schaft muß die Schnittstelle in der Vorfahrenliste aufführen. Standardmäßig führt die Verwendung des Schlüsselwortes **implements** zur Delegation aller Schnittstellenmethoden. Sie können aber Klauseln zur Methodenauflösung verwenden oder Methoden in der Klasse implementieren, um einzelne dieser Methoden zu überschreiben.

Das folgende Beispiel verwendet das Schlüsselwort **implements** zur Bereitstellung eines Farbadapter-Objekts, mit dem ein RGB-Farbwert (8 Bit) in eine *Color*-Referenz konvertiert wird:

```
type
IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;

IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;

{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
  TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

Weitere Informationen über Syntax, Implementierung und Sprachregeln des Schlüsselwortes **implements** finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Objektschnittstellen«.

Aggregation

Die Aggregation stellt einen modularen Ansatz zur Wiederverwendung von Quelltext dar. Untergeordnete Objekte definieren die Funktionalität des umgebenden Objekts, verbergen aber die Implementierungsdetails vor diesem Objekt. Bei der Aggregation implementiert ein umgebendes Objekt eine oder mehrere Schnittstellen. *IUnknown* muß implementiert werden. Das enthaltene Objekt (bzw. die Objekte) implementiert eine oder mehrere Schnittstellen. Nur das umgebende Objekt macht jedoch diese Schnittstellen verfügbar. Dies gilt sowohl für die vom umgebenden Objekt selbst als auch für die vom enthaltenen Objekt implementierten Schnittstellen. Clients besitzen keinerlei Informationen über die enthaltenen Objekte. Da das umgebende Objekt den Zugriff auf die Schnittstellen der enthaltenen Objekte ermöglicht, sind deren Implementierungen vollständig transparent. Aus diesem Grund kann im umgebenden Objekt der Klassentyp des enthaltenen Objekts jederzeit gegen den Typ einer Klasse ausgetauscht werden, die dieselbe Schnittstelle implementiert. Dementsprechend kann der Quelltext der Klassen der enthaltenen Objekte auch von anderen Klassen genutzt werden.

Das Implementierungsmodell der Aggregation definiert explizite Regeln zur Implementierung von *IUnknown* über den Mechanismus der Delegation. Das enthaltene Objekt muß eine eigene *IUnknown*-Instanz implementieren, die den Referenzzähler für das enthaltene Objekt bereitstellt. Diese Implementierung von *IUnknown* überwacht die Interaktion zwischen dem umgebenden und dem enthaltenen Objekt. Wird ein Objekt mit dem Typ des enthaltenen Objekts erstellt, kann dies nur erfolgreich durchgeführt werden, wenn eine Schnittstelle mit dem Typ *IUnknown* angefordert wird. Das enthaltene Objekt implementiert außerdem eine zweite *IUnknown*-Instanz für alle in diesem Objekt implementierten Schnittstellen. Diese Schnittstellen werden dem umgebenden Objekt verfügbar gemacht. Die zweite IUnknown-Instanz delegiert alle Aufrufe der Methoden *QueryInterface*, *AddRef* und *Release* an das umgebende Objekt. Die *IUnknown*-Instanz des umgebenden Objekts wird als »steuernde Unknown-Schnittstelle« bezeichnet.

In der Online-Hilfe finden Sie Informationen zu den Regeln für die Erstellung einer Aggregation. Wenn Sie eigene Aggregationsklassen erstellen, sollten Sie außerdem die Implementierung von *IUnknown* in *TComObject* beachten. *TComObject* ist eine COM-Klasse, die den Mechanismus der Aggregation unterstützt. Wenn Sie COM-Anwendungen schreiben, können Sie *TComObject* direkt als Basisklasse einsetzen.

Speicherverwaltung für Schnittstellenobjekte

Eines der grundlegenden Konzepte beim Entwurf von Schnittstellen besteht in der Sicherstellung der Referenzverwaltung für die Objekte, welche die Schnittstellen implementieren. Die *IUnknown*-Methoden *AddRef* und *Release* ermöglichen die Implementierung dieser Funktionalität. Sie überwachen die Existenz eines Objekts mit Hilfe eines Referenzzählers. Dieser wird jedesmal erhöht, wenn eine Schnittstellenreferenz an einen Client übergeben wird. Sobald der Referenzzähler den Wert Null erreicht, wird das Objekt freigegeben.

Wenn Sie COM-Objekte für verteilte Anwendungen erstellen, müssen Sie die Regeln zur Referenzzählung genauestens beachten. Werden die Schnittstellen dagegen nur

anwendungsintern eingesetzt, können Sie in Abhängigkeit vom Objekt und seinem Einsatzzweck auch anders vorgehen.

Referenzzählung einsetzen

Delphi stellt den wesentlichen Teil der *IUnknown*-Speicherverwaltung in Form der Implementierung der Schnittstellenabfrage und des Referenzzählers bereit. Ein Objekt, dessen Existenz von seinen Schnittstellen abhängig ist, kann also einfach von diesen Klassen abgeleitet werden, da diese die Referenzzählung zur Verfügung stellen. *TInterfacedObject* ist die Nicht-*CoClass*, die dieses Verhalten implementiert. Wenn Sie die Referenzzählung nutzen wollen, dürfen Sie das Objekt nur als Schnittstellenreferenz verwalten. Die Referenzzählung muß konsistent erfolgen. Ein Beispiel:

```
procedure beep(x: ITest);
function test_func()
var
      y: ITest;
begin
      y := TTest.Create; // Da der Typ ITest ist, beträgt der Referenzzähler 1
beep(y); // Das Aufrufen der Funktion beep erhöht den Referenzzähler
// Nach Ausführung der Funktion wird der Zähler wieder verringert
      y.something; // Das Objekt existiert weiterhin mit einem Referenzzähler von 1
end;
```

Dies ist die sicherste Methode der Speicherverwaltung, die bei Verwendung von *TInterfacedObject* automatisch erfolgt. Wenn Sie diese Regel nicht beachten, kann das Objekt ungewollt freigegeben werden:

```
function test_func()
var
      x: TTest;
begin
      x := TTest.Create; // Referenzzähler des Objekts beträgt 0
      beep(x as ITest); // Zähler wird durch Aufruf der Funktion beep erhöht
// Nach Ausführung der Funktion wird der Zähler wieder verringert
      x.something; // Das Objekt existiert nicht mehr
end;
```

Hinweis In den beschriebenen Beispielen erhöht die Prozedur *beep* der Deklaration entsprechend den Referenzzähler im Parameter durch einen Aufruf von *AddRef*. Dies geschieht bei den folgenden Deklarationen nicht:

```
procedure beep(const x: ITest);
```

oder

```
procedure beep(var x: ITest);
```

Diese Deklarationen generieren schnelleren und weniger umfangreichen Code.

In einem Sonderfall kann die Referenzzählung nicht genutzt werden, da keine konsistente Durchführung der Zählung möglich ist: Wenn das Objekt eine Komponente bzw. ein Steuerelement ist, das zu einer anderen Komponente gehört, sollten Sie keine Referenzzählung einsetzen, da die Lebensdauer des Objekts nicht von den Schnittstellen diktiert wird.

Keine Referenzzählung einsetzen

Wenn es sich bei dem Objekt um eine VCL-Komponente oder ein Steuerelement handelt, die bzw. das einer anderen Komponente gehört, unterliegt dieses Objekt einem anderen Speicherverwaltungssystem, das auf *TComponent* basiert. Sie sollten die Ansätze zur Referenzverwaltung - VCL-Komponenten im Unterschied zur COM-Referenzzählung - nicht mischen. Wenn Sie eine Komponente mit Schnittstellenunterstützung erstellen, können Sie die *IUnknown*-Methoden *AddRef* und *Release* als leere Funktionen implementieren, um den COM-Referenzzählungsmechanismus zu umgehen:

```
function TMyObject.AddRef: Integer;
begin
Result := -1;
end;

function TMyObject.Release: Integer;
begin
Result := -1;
end;
```

Auch in diesem Fall kann *QueryInterface* implementiert werden, um die dynamische Abfrage für das Objekt zu ermöglichen.

Da Sie *QueryInterface* implementieren, können Sie weiterhin den Operator as für die Schnittstellen von Komponenten verwenden, sofern Sie einen Schnittstellenbezeichner erstellen (IID). Ebenso können Sie die Aggregation einsetzen. Handelt es sich bei dem umgebenden Objekt um eine Komponente, implementiert das enthaltene Objekt die Referenzzählung durch Delegation an die »steuernde Unknown-Schnittstelle«. Auf der Ebene des umgebenden Komponentenobjekts wird über das Umgehen der Methoden *AddRef* und *Release* und über die Durchführung der Speicherverwaltung auf VCL-Art entschieden. Sie können also *TInterfacedObject* als Basisklasse für ein enthaltenes Objekt einer Aggregation verwenden, das seinerseits eine Komponente als umgebendes Objekt besitzt.

Hinweis Die »steuernde Unknown-Schnittstelle« ist die vom umgebenden Objekt implementierte *IUnknown*-Instanz, mit deren Hilfe die Referenzzählung für das gesamte Objekt verwaltet wird. Weitere Informationen zu den unterschiedlichen Implementierungen der Schnittstelle *IUnknown* durch das enthaltene bzw. das umgebende Objekt finden Sie im Abschnitt »Aggregation« auf Seite 3-22 und in der Online-Hilfe von Microsoft zur »steuernden Unknown-Schnittstelle«.

Schnittstellen in verteilten Anwendungen einsetzen

Schnittstellen sind die fundamentalen Elemente der beiden Objektmodelle COM und CORBA. Delphi besitzt für diese Technologien Basisklassen, die über die grundlegende Funktionalität in *TInterfacedObject* hinausgehen. *TInterfacedObject* implementiert die Methoden der Schnittstelle *IUnknown*.

COM-Klassen erweitern die Funktionalität von Klassengeneratoren und Klassenbezeichnern (CLSIDs). Klassengeneratoren dienen der Erstellung von Klasseninstanzen über CLSIDs. Die CLSIDs werden zur Registrierung und Bearbeitung von COM-

Klassen eingesetzt. COM-Klassen mit Klassengeneratoren und Klassenbezeichnern werden als *CoClasses* bezeichnet. *CoClasses* nutzen die Funktion zur Versionsüberwachung von *QueryInterface*. Wird also ein Softwaremodul aktualisiert, kann *QueryInterface* zur Laufzeit aufgerufen werden, um die aktuellen Merkmale eines Objekts zu ermitteln. Neue Versionen alter Schnittstellen werden wie neue Schnittstellen oder Merkmale eines Objekts sofort für neue Clients verfügbar. Gleichzeitig bleiben die Objekte vollständig zum vorhandenen Client-Code kompatibel. Eine erneute Compilierung ist nicht erforderlich, da die Schnittstellenimplementierungen verborgen sind (Methoden und Parameter bleiben erhalten). In COM-Anwendungen können Entwickler die Implementierung ändern, um beispielsweise die Leistung zu verbessern (oder um interne Anforderungen zu erfüllen), ohne den Client-Code zu beeinflussen, der auf dieser Schnittstelle basiert. Weitere Informationen zu COM-Schnittstellen finden Sie in Kapitel 44, »COM-Technologien im Überblick« .

Die zweite Technologie für verteilte Anwendungen heißt CORBA. Der Einsatz von Schnittstellen in CORBA-Anwendungen wird durch die *Stub*-Klassen des Clients und die Skeleton-Klassen des Servers vermittelt. Diese Stub- und Skeleton-Klassen verwalten die Ausführung von Schnittstellenaufrufen, um die richtige Übergabe der Parameter- und Rückgabewerte sicherzustellen. Anwendungen müssen eine Stub- bzw. eine Skeleton-Klasse verwenden oder die DII (Dynamic Invocation Interface) nutzen. Diese Schnittstelle konvertiert alle Parameter in spezielle Varianten, die eigene Typinformationen enthalten. Obwohl dies nicht zu den erforderlichen Merkmalen der CORBA-Technologie gehört, implementiert Delphi Klassengeneratoren in einer Weise, die mit der Verwendung von Klassengeneratoren und CoClasses durch COM vergleichbar ist. Durch diese Vereinheitlichung der beiden Architekturmodelle für verteilte Anwendungen unterstützt Delphi kombinierte COM / CORBA-Server, die gleichzeitig Anforderungen von COM- und von CORBA-Clients verarbeiten können. Weitere Informationen zum Einsatz von Schnittstellen mit CORBA finden Sie in Kapitel 28, »CORBA-Anwendungen«.

Mit Strings arbeiten

In Delphi gibt es eine Reihe unterschiedlicher Zeichen- und String-Typen, die im Rahmen der Entwicklung der Sprache Object Pascal eingeführt wurden. Dieser Abschnitt bietet eine Übersicht zu diesen Typen, ihrem Zweck und ihrer Verwendung. Umfassende Syntaxinformationen finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz«

Zeichentypen

Delphi besitzt drei Zeichentypen: *Char, AnsiChar und WideChar.*

Der Zeichentyp *Char* wurde aus Standard-Pascal in Turbo Pascal und später in Object Pascal übernommen. Object Pascal implementiert zusätzlich die speziellen Zeichentypen *AnsiChar* und *WideChar*, um die Standards des Betriebssystems Windows für die Zeichendarstellung zu unterstützen. *AnsiChar* wurde zur Unterstützung des 8-Bit Zeichenstandards ANSI und *WideChar* zur Unterstützung des 16-Bit-Standards Unicode bereitgestellt. Zeichen vom Typ *WideChar* bestehen aus zwei Bytes. In diesem

Zeichensatz können also wesentlich mehr Zeichen dargestellt werden. Mit der Implementierung von *AnsiChar* und *WideChar* wurde *Char* der Standard-Zeichentyp, der die aktuell empfohlene Implementierung repräsentiert. Beachten Sie beim Einsatz von *Char*, daß die Implementierung dieses Typs voraussichtlich in zukünftigen Versionen von Delphi geändert wird.

Die folgende Tabelle stellt die Zeichentypen im Überblick dar.

Tabelle 3.2 Zeichentypen von Object Pascal

Typ	Bytes	Inhalt	Zweck
Char	1	Enthält ein einzelnes ANSI-Zeichen.	Standard-Zeichentyp.
AnsiChar	1	Enthält ein einzelnes ANSI-Zeichen.	Ansi-Zeichensatzstandard (8 Bit) unter Windows.
WideChar	2	Enthält ein einzelnes Unicode-Zeichen.	Unicode-Zeichensatzstandard (16 Bit) unter Windows.

Weitere Informationen zum Einsatz dieser Zeichentypen finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter Zeichentypen. Weitere Informationen zu Unicode-Zeichen finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Erweiterte Zeichensätze«.

String-Typen

Delphi besitzt drei Typkategorien, die Sie bei der Arbeit mit Strings nutzen können: Zeichenzeiger, String-Typen und VCL-String-Klassen. Dieser Abschnitt beschreibt die String-Typen und ihren Einsatz in Verbindung mit Zeichenzeigern. Weitere Informationen zu den VCL-String-Klassen finden Sie in der Online-Hilfe unter »TStrings«.

Derzeit sind drei String-Implementierungen in Delphi verfügbar: kurze, lange und Wide-Strings. Diese Implementierungen werden durch unterschiedliche String-Typen repräsentiert. Zusätzlich ist das reservierte Wort **string** verfügbar, das die aktuell empfohlene String-Implementierung bildet.

Kurze Strings

String war der erste in Turbo Pascal verwendete String-Typ. **String** wurde ursprünglich als kurzer String implementiert. Es handelt sich dabei um eine Zuweisung von 1 bis 256 Bytes. Das erste Byte gibt die Länge des Strings an, die weiteren Bytes enthalten die Zeichen des Strings:

```
S: string[0..n]// Ursprünglicher String-Typ
```

Mit der Einführung langer Strings wurde **string** standardmäßig als Typ für lange Strings implementiert. *ShortString* wurde eingeführt, um die Abwärtskompatibilität zu gewährleisten. *ShortString* ist ein vordefinierter Typ für Strings mit einer Maximallänge:

```
S: string[255]// Der Typ ShortString
```

Der Umfang des für einen *ShortString* reservierten Speichers ist statisch, wird also während der Compilierung festgelegt. Die Position des Speichers für den *ShortString* kann jedoch dynamisch zugewiesen werden, wenn Sie beispielsweise *PShortString* verwenden, einen Zeiger auf einen *ShortString*. Die Anzahl der von einer Variablen des Typs *ShortString* belegten Bytes ergibt sich aus der maximalen Länge dieses Strings plus Eins. Für den vordefinierten Typ *ShortString* beträgt die Anzahl 256 Bytes.

Kurze Strings - ob sie mit der Syntax **string**[0..n] oder dem vordefinierten Typ *ShortString* deklariert werden - dienen in erster Linie der Abwärtskompatibilität mit früheren Versionen von Delphi und Borland Pascal.

Mit der Compiler-Direktive **$H** legen Sie fest, ob das reservierte Wort **string** einen kurzen oder einen langen String repräsentiert. Standardmäßig ({**$H+**}) repräsentiert string einen langen String. Sie können den Typ *ShortString* aktivieren, indem Sie die Direktive {**$H-**} verwenden. {**$H-**} ist insbesondere hilfreich, wenn Code aus Object-Pascal-Versionen verwendet wird, der standardmäßig kurze Strings verwendet. Kurze Strings sind beispielsweise in Datenstrukturen hilfreich, für die eine Komponente fester Größe erforderlich ist, oder in DLLs, wenn die Unit *ShareMem* nicht verwendet werden soll (weitere Informationen finden Sie in der Online-Hilfe unter »Speicherverwaltung«). Sie können die Bedeutung von String-Typdefinitionen lokal überschreiben, damit kurze Strings generiert werden. Außerdem können Sie die Deklarationen kurzer String-Typen in string[255] oder *ShortString* ändern. Diese Deklarationen sind eindeutig und nicht von der Compiler-Direktive **$H** abhängig.

Weitere Informationen zu kurzen Strings und zum Typ *ShortString* finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Kurze Strings«.

Lange Strings

Lange Strings werden dynamisch zugewiesen. Die Maximallänge wird nur durch den verfügbaren Speicher beschränkt. Lange Strings verwenden wie kurze Strings Ansi-Zeichen (8 Bit) und enthalten ebenfalls eine Längenangabe. Im Unterschied zu kurzen Strings besitzen sie jedoch kein Element an der Position Null, das die dynamische Länge des Strings angibt. Die genaue Länge eines langen Strings können Sie mit der Standardfunktion *Length* ermitteln und mit der Standardprozedur *SetLength* einstellen. Lange Strings werden mit Hilfe eines Referenzzählers verwaltet und wie *PChars* mit einem Null-Zeichen abgeschlossen. Weitere Informationen zur Implementierung langer Strings finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Lange Strings«.

Lange Strings werden mit dem reservierten Wort **string** und dem vordefinierten Bezeichner *AnsiString* eingeleitet. Für neue Anwendungen wird der Einsatz langer Strings empfohlen. Alle Komponenten in der VCL wurden in dieser Form compiliert (normalerweise mit **string**). Wenn Sie Komponenten schreiben, sollten Sie ebenfalls lange Strings verwenden. Dies gilt auch für Quelltext, der Daten von VCL-String-Eigenschaften verarbeitet. Zur Erstellung von Quelltext, der immer lange Strings verwendet, sollten Sie *AnsiString* verwenden. Soll der Quelltext dagegen flexibel sein, damit auch neue String-Implementierungen problemlos unterstützt werden können, empfiehlt sich der Einsatz des reservierten Wortes **string**.

WideString

Der Typ *WideChar* ermöglicht die Repräsentation von Strings mit Doppelbyte-Zeichen als Arrays mit *WideChars*. Wide-Strings bestehen aus Unicode-Zeichen (16 Bit). Wide-Strings werden wie lange Strings dynamisch zugewiesen, die Länge ist also nur durch den verfügbaren Speicher beschränkt. Für Wide-Strings wird jedoch keine Referenzzählung durchgeführt. Der dynamisch zugewiesene Speicher mit dem String wird freigegeben, sobald der String nicht mehr im Gültigkeitsbereich liegt. Im übrigen besitzen Wide-Strings dieselben Attribute wie lange Strings. Der Typ *WideString* wird mit dem vordefinierten Bezeichner WideString deklariert.

Da die 32-Bit-Version von OLE für alle Strings Unicode verwendet, müssen Strings in Eigenschaften und Methoden für die OLE-Automatisierung den String-Typ *Wide* besitzen. Auch die meisten OLE-API-Funktionen verwenden nullterminierte Wide-Strings.

Weitere Informationen zu Wide-Strings finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Wide-Strings«.

PChar-Typen

Ein *PChar* ist ein Zeiger auf einen nullterminierten String mit Zeichen des Typs *Char*. Jeder der drei Zeichentypen besitzt einen integrierten Zeigertyp:

- *PChar* ist ein Zeiger auf einen nullterminierten String mit 8-Bit-Zeichen.
- *PAnsiChar* ist ein Zeiger auf einen nullterminierten String mit 8-Bit-Zeichen.
- *PWideChar* ist ein Zeiger auf einen nullterminierten String mit 16-Bit-Zeichen.

PChars bilden mit kurzen Strings den ursprünglichen String-Typ von Object Pascal. Sie wurden als Kompatibilitätstyp zur Sprache C und zur Windows-API eingeführt.

OpenString

Der Typ *OpenString* ist veraltet, kommt aber gelegentlich noch in älterem Quelltext vor. Er dient der 16-Bit-Kompatibilität und darf ausschließlich in Parametern verwendet werden. *OpenString* wurde vor der Implementierung langer Strings verwendet, um die Übergabe kurzer Strings ohne Längenangabe als Parameter zu ermöglichen. Ein Beispiel:

```
procedure a(v : openstring);
```

Diese Deklaration erlaubt die Übergabe eines Strings beliebiger Länge als Parameter, obwohl sich die Länge des formalen und des eigentlichen Parameters normalerweise exakt entsprechen müssen. Die Verwendung des Typs *OpenString* ist in neuen Anwendungen nicht erforderlich.

Beachten Sie auch die Beschreibung des Schalters {$P+/-} unter »Compiler-Direktiven für Strings« auf Seite 3-35.

Routinen der Laufzeitbibliothek zur String-Verarbeitung

Die Laufzeitbibliothek enthält verschiedene Routinen zur String-Verarbeitung, die für die unterschiedlichen String-Typen spezifisch sind. Sie stellt Routinen für Wide-Strings, lange Strings und nullterminierte Strings (also PChars) bereit. Die Routinen für PChar-Typen verwenden das abschließende Null-Zeichen, um die Länge des Strings zu ermitteln. Weitere Informationen zu nullterminierten Strings finden Sie in der Online-Hilfe »Object Pascal Sprachreferenz« unter »Nullterminierte Strings«.

Die Laufzeitbibliothek enthält außerdem eine Reihe von Routinen zur Formatierung von Strings. Für ShortString-Typen liegen keine speziellen Routinen vor. Es gibt jedoch einige integrierte Compiler-Routinen für diesen Typ. Dazu gehören beispielsweise die Standardfunktionen *Low* und *High*.

Da Wide-Strings und lange Strings die gebräuchlichsten String-Typen sind, befassen sich die folgenden Abschnitte mit den Routinen für diese Typen.

Wide-Zeichenroutinen

Bei der Arbeit mit Strings müssen Sie sicherstellen, daß der Quelltext die Strings auch in den gegebenenfalls verwendeten unterschiedlichen Sprachumgebungen verarbeiten kann. Unter Umständen müssen Sie Wide-Strings und Wide-Zeichen einsetzen. Ein Ansatz der Arbeit mit ideografischen Zeichensätzen besteht deshalb auch in der Konvertierung aller Zeichen in ein Codierungsschema für Wide-Zeichen wie Unicode. Die Laufzeitbibliothek enthält die folgenden String-Funktionen für Wide-Zeichen, mit deren Hilfe zwischen Strings mit Einzelbyte-Zeichen (oder MBCS-Strings) und Unicode-Strings konvertiert werden kann:

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

Die Verwendung eines Codierungsschemas für Wide-Zeichen hat den Vorteil, daß Sie bei der Programmierung von Voraussetzungen ausgehen können, die von MBCS-Systemen nicht unterstützt werden. Es gibt eine direkte Beziehung zwischen der Anzahl der Bytes und der Anzahl der Zeichen im String. Dadurch reduziert sich die Gefahr des Abschneidens der Hälfte aller Zeichen im String oder der Verwechslung der zweiten Hälfte eines Zeichens mit der ersten Hälfte des folgenden Zeichens.

Ein Nachteil des Einsatzes von Wide-Zeichen besteht jedoch darin, daß Windows 95 keine Aufrufe von API-Funktionen mit Wide-Zeichen unterstützt. Aus diesem Grund repräsentieren alle VCL-Komponenten String-Werte als Einzelbyte- bzw. MBCS-Strings. Die Konvertierung zwischen Wide-Zeichen und dem MBCS-System, wenn eine String-Eigenschaft gelesen oder bearbeitet werden muß, macht nicht nur viel zusätzlichen Quelltext erforderlich, sondern bremst auch die Anwendung erheblich. Sie können trotzdem die Konvertierung in Wide-Zeichen vornehmen, wenn Sie spezielle

Algorithmen zur String-Verarbeitung einsetzen, um die 1:1-Zuordnung zwischen Zeichen und WideChars zu nutzen.

Gebräuchliche Routinen für lange Strings

Die Routinen für lange Strings decken verschiedene Funktionsbereiche ab. In diesen Bereichen dienen verschiedene Routinen demselben Zweck, verwenden jedoch für die Berechnungen unterschiedliche Kriterien. Die folgenden Tabellen führen diese Routinen nach Funktionsbereichen geordnet auf:

- Vergleich
- Konvertierung der Groß-/Kleinschreibung
- Bearbeitung
- Teil-String

Gegebenenfalls enthalten die Tabellen weitere Spalten, in denen die Erfüllung der folgenden Kriterien durch die Routine beschrieben werden:

- Berücksichtigung der Groß-/Kleinschreibung – Bei Verwendung eines Windows-Sprachtreibers wird die Definition der Groß-/Kleinschreibung ermittelt. Wenn die Routine nicht auf einem Windows-Sprachtreiber basiert, erfolgt die Analyse nach Maßgabe der Ordinalwerte der Zeichen. Berücksichtigt die Routine die Groß-/Kleinschreibung nicht, werden Groß- und Kleinbuchstaben nach einem vordefinierten Muster gemischt.
- Verwendung des Windows-Sprachtreibers – Die Berücksichtigung des Windows-Sprachtreibers ermöglicht die Unterstützung sprachtreiberspezifischer Funktionen in der Anwendung. Dies gilt insbesondere für asiatische Sprachumgebungen. Die meisten Windows-Sprachtreiber unterstellen, daß Kleinbuchstaben einen geringeren Wert als die entsprechenden Großbuchstaben besitzen. Sie unterscheiden sich darin von der ASCII-Reihenfolge, in der Kleinbuchstaben einen größeren Wert als Großbuchstaben haben. Routinen, die den Windows-Sprachtreiber berücksichtigen, besitzen normalerweise einen mit *Ansi...* eingeleiteten Namen (also *AnsiXXX*).
- Unterstützung des Multibyte-Zeichensatzes (MBCS) – Multibyte-Zeichensätze werden beim Schreiben von Quelltext für fernöstliche Sprachtreiber eingesetzt. Multibyte-Zeichen werden durch eine Kombination aus Einzelbyte- und Doppelbyte-Zeichencodes repräsentiert. Die Länge des Strings in Byte entspricht also nicht unbedingt der Anzahl der Zeichen im String. Die Routinen zur Unterstützung von Multibyte-Zeichensätzen verarbeiten Einzel- und Doppelbyte-Zeichen. *ByteType* und *StrByteType* ermitteln, ob ein bestimmtes Byte das erste Byte eines Doppelbyte-Zeichens darstellt. Achten Sie beim Einsatz von Multibyte-Zeichen sorgfältig darauf, Strings nicht abzuschneiden, indem ein Doppelbyte-Zeichen halbiert wird. Übergeben Sie Zeichen nicht als Parameter an Prozeduren oder Funktionen, da die Größe der Zeichen nicht von vornherein feststeht. Sie können statt dessen einen Zeiger auf das Zeichen bzw. den String übergeben. Weitere In-

formationen zu Multibyte-Zeichensätzen finden Sie unter »Quelltext anpassen« auf Seite 10-2.

Tabelle 3.3 Routinen zum String-Vergleich

Routine	Groß-/ Kleinschreibung	Windows-Sprachtreiber	MBCS-Unterstützung
AnsiCompareStr	Ja	Ja	Ja
AnsiCompareText	Nein	Ja	Ja
AnsiCompareFileName	Nein	Ja	Ja
CompareStr	Ja	Nein	Nein
CompareText	Nein	Nein	Nein

Tabelle 3.4 Routinen zur Konvertierung zwischen Groß-/Kleinschreibung

Routine	Windows-Sprachtreiber	MBCS-Unterstützung
AnsiLowerCase	Ja	Ja
AnsiLowerCaseFileName	Ja	Ja
AnsiUpperCaseFileName	Ja	Ja
AnsiUpperCase	Ja	Ja
LowerCase	Nein	Nein
UpperCase	Nein	Nein

Tabelle 3.5 Routinen zur String-Bearbeitung

Routine	Groß-/Kleinschreibung	MBCS-Unterstützung
AdjustLineBreaks	Nicht anwendbar	Ja
AnsiQuotedStr	Nicht anwendbar	Ja
StringReplace	Optional, Flag-gesteuert	Ja
Trim	Nicht anwendbar	Ja
TrimLeft	Nicht anwendbar	Ja
TrimRight	Nicht anwendbar	Ja
WrapText	Nicht anwendbar	Ja

Tabelle 3.6 Routinen für Teil-Strings

Routine	Groß-/Kleinschreibung	MBCS-Unterstützung
AnsiExtractQuotedStr	Nicht anwendbar	Ja
AnsiPos	Ja	Ja
IsDelimiter	Ja	Ja
IsPathDelimiter	Ja	Ja

Tabelle 3.6 Routinen für Teil-Strings

Routine	Groß-/Kleinschreibung	MBCS-Unterstützung
LastDelimiter	Ja	Ja
QuotedStr	Nein	Nein

Die Routinen für String-Dateinamen (*AnsiCompareFileName*, *AnsiLowerCaseFileName* und *AnsiUpperCaseFileName*) verwenden den Windows-Sprachtreiber. Sie sollten immer Dateinamen verwenden, die problemlos portierbar sind, da der für Dateinamen verwendete Zeichensatz vom Sprachtreiber abhängig ist und deshalb auf jedem Benutzer-Desktop abweichen kann.

Strings deklarieren und initialisieren

Ein langer String braucht bei der Deklaration nicht initialisiert zu werden:

```
S: string;
```

Lange Strings werden automatisch mit einem Leerwert initialisiert. Mit der Variablen *EmptyStr* können Sie ermitteln, ob ein String einen Leerwert enthält:

```
S = EmptyStr;
```

Sie können die Prüfung auch mit Hilfe eines leeren Strings vornehmen:

```
S = '';
```

Ein leerer String enthält keine gültigen Daten. Aus diesem Grund entspricht die Indizierung eines leeren Strings einem Zugriff auf **nil**: Das Ergebnis ist eine Zugriffsverletzung.

```
var
      S: string;
begin
      S[i];  // Löst eine Zugriffsverletzung aus
      // Anweisungen
end;
```

Wenn Sie einen leeren String in einen *PChar* umwandeln, ist das Ergebnis ein **nil**-Zeiger. Einen solchen *PChar* dürfen Sie nur an Routinen übergeben, die zur Verarbeitung von **nil**-Werten eingerichtet wurden:

```
var
      S: string; // Leerer String
begin
      proc(PChar(S));// Prozedur muß nil-Werte verarbeiten können
      // Anweisungen
end;
```

Kann die Prozedur keine **nil**-Werte verarbeiten, müssen Sie den String initialisieren:

```
S := 'Nicht mehr nil';
proc(PChar(S));// Prozedur muß keine nil-Werte verarbeiten können
```

Alternativ können Sie die Länge des Strings mit der Prozedur *SetLength* einstellen:

```
SetLength(S, 100);// Weist S die dynamische Länge 100 zu
```

```
proc(PChar(S));// Prozedur muß keine nil-Werte verarbeiten können
```

Wenn Sie *SetLength* verwenden, bleiben die Zeichen im String erhalten. Der Inhalt des neu zugewiesenen Speichers ist allerdings nicht definiert. Nach einem Aufruf von *SetLength* ist sichergestellt, daß S einen eindeutigen String referenziert, dessen Referenzzähler den Wert Eins enthält. Die Länge des Strings kann mit der Funktion Length ermittelt werden.

Beachten Sie die folgende Deklaration:

```
S: string[n];
```

String deklariert implizit einen kurzen String, keinen langen String der Länge *n*. Wollen Sie einen langen String der angegebenen Länge n deklarieren, müssen Sie eine Variable des Typs **string** deklarieren und anschließend die Prozedur *SetLength* verwenden.

```
S: string;
SetLength(S, n);
```

String-Typen mischen und konvertieren

Kurze Strings, lange Strings und Wide-Strings können in Zuweisungen und Ausdrücken gemischt werden. Der Compiler generiert automatisch Code zur Durchführung der erforderlichen String-Typkonvertierungen. Wenn Sie einer kurzen String-Variablen einen String-Wert zuweisen, wird der Wert gegebenenfalls abgeschnitten, wenn seine Länge die deklarierte Maximallänge der kurzen String-Variablen übersteigt.

Lange Strings werden immer dynamisch zugewiesen. Wenn Sie einen der integrierten Zeigertypen wie *PAnsiString*, *PString* oder *PWideString* verwenden, müssen Sie berücksichtigen, daß dadurch eine neue Umleitungsstufe eingeführt wird.

Konvertierungen von String in PChar

Konvertierungen langer Strings in *PChar* werden nicht automatisch durchgeführt. Aufgrund bestimmter Unterschiede zwischen Strings und *PChars* können bei Konvertierungen Probleme auftreten:

- Lange Strings werden mit Referenzzählern verwaltet, *PChars* nicht.
- Bei der Zuweisung an einen String werden die Daten kopiert, während ein *PChar* ein Zeiger auf eine Speicheradresse ist.
- Lange Strings sind nullterminiert und enthalten zusätzlich eine Längenangabe, *PChars* sind einfach nullterminiert.

Dieser Abschnitt beschreibt Situationen, in denen diese Unterschiede zu schwer korrigierbaren Fehlern führen können.

String-Abhängigkeiten

Gelegentlich muß ein langer String in einen nullterminierten String konvertiert werden, wenn Sie beispielsweise eine Funktion aufrufen, die einen *PChar* als Parameter benötigt. Da für lange Strings eine Referenzzählung durchgeführt wird, wird der Abhängigkeitswert des Strings um Eins erhöht, obwohl der eigentliche Referenzzähler nicht erhöht wird. Sobald der Referenzzähler den Wert Null erreicht, wird der String freigegeben, trotz der noch vorhandenen Abhängigkeit. Der umgewandelte *PChar* ist ebenfalls nicht mehr verfügbar, obwohl die Routine, an die er übergeben wurde, möglicherweise noch darauf zugreift. Wenn Sie also einen String in einen *PChar* umwandeln müssen, sind Sie dafür verantwortlich, das der resultierende *PChar* verfügbar bleibt. Ein Beispiel:

```
procedure my_func(x: string);
begin
      // Operation mit x
      some_proc(PChar(x)); // Umwandlung des Strings in PChar
      // Jetzt müssen Sie sicherstellen, daß der String verfügbar bleibt,
      // solange er von some_proc benötigt wird
end;
```

Lokale PChar-Variable zurückgeben

Ein häufiger Fehler bei der Arbeit mit *PChars* ist das Speichern in einer Datenstruktur bzw. die Rückgabe als Wert in einer lokalen Variablen. Sobald die Routine beendet wird, ist der *PChar* nicht mehr verfügbar, da es sich nur um einen Zeiger auf eine Speicheradresse handelt, nicht um eine von der Referenzzählung berücksichtigte Kopie des Strings. Ein Beispiel:

```
function title(n: Integer): PChar;
var
   s: string;
begin
   s := Format('title - %d', [n]);
   Result := PChar(s); // FEHLER
end;
```

In diesem Beispiel wird ein Zeiger auf String-Daten zurückgegeben, die bei Beendigung der Funktion *title* freigegeben werden.

Lokale Variable als PChar übergeben

In bestimmten Fällen liegt eine lokale String-Variable vor, die durch den Aufruf einer Funktion initialisiert werden muß, die ihrerseits einen *PChar* als Parameter entgegennimmt. Eine mögliche Lösung ist die Erstellung eines lokalen **array of char**, das dann an die Funktion übergeben wird. Nach Ausführung der Funktion kann die Variable einem String zugewiesen werden:

```
// MAX_SIZE ist hier eine vordefinierte Konstante
var
      i: Integer;
      buf: array[0..MAX_SIZE] of char;
      S: string;
```

```
begin
      i := GetModuleFilename(0, @buf, SizeOf(buf));// Behandelt @buf als PChar
      S := buf;
      // Anweisungen
end;
```

Dieser Lösungsweg ist hilfreich, wenn der Puffer relativ klein ist, da dieser auf dem Stack zugeordnet wird. Gleichzeitig handelt es sich um eine sichere Lösung, da die Konvertierung zwischen **array of char** und **string** automatisch erfolgt. Sobald *GetModuleFilename* beendet ist, gibt die Funktion *Length* für diesen String die richtige Anzahl Bytes an, die in buf geschrieben wurden.

Um den zusätzlichen Aufwand beim Kopieren des Puffers zu vermeiden, können Sie den String in einen *PChar* umwandeln, sofern die Routine nicht voraussetzt, daß der *PChar* im Speicher erhalten bleibt. Die Synchronisation der String-Länge erfolgt jedoch nicht automatisch wie bei der Zuweisung von **array of char** an **string**. Sie sollten die Länge des Strings zurücksetzen, damit sie der tatsächlichen Länge entspricht. Wenn Sie eine Funktion verwenden, welche die Anzahl der kopierten Bytes zurückgibt, ist dies in nur einer Zeile Quelltext möglich:

```
var
      S: string;
begin
      SetLength(S, 100);// Bei der Umwandlung in einen PChar darf der String nicht leer sein
      SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
      // Anweisungen
end;
```

Compiler-Direktiven für Strings

Die folgenden Compiler-Direktiven wirken sich auf Zeichen- und String-Typen aus:

- {$H+/-}: Diese Compiler-Direktive steuert, ob das reservierte Wort **string** einen kurzen oder einen langen String repräsentiert. Beim Standardwert {$H+} wird **string** als Repräsentation langer Strings interpretiert. Mit der Direktive {$H-} wird **string** dagegen als Repräsentation kurzer Strings interpretiert.
- {$P+/-}: Die Direktive $P ist nur von Bedeutung, wenn der Quelltext mit der Direktive {$H-} kompiliert wird. $P dient der Abwärtskompatibilität zu früheren Versionen von Delphi und Borland Pascal. $P steuert die Bedeutung von **var**-Parametern, wenn die Direktive {$H-} und das Schlüsselwort **string** verwendet wird. Bei {$P-} werden mit dem Schlüsselwort **string** deklarierte Parameter als normale **var**-Parameter interpretiert, bei der Direktive {$P+} dagegen als offene String-Parameter. Unabhängig von der Einstellung der Direktive $P kann der Bezeichner *OpenString* immer zur Deklaration offener String-Parameter verwendet werden.
- {$V+/-}: Die Direktive $V steuert die Typprüfung für kurze Strings, die als **var**-Parameter übergeben werden. Mit {$V+} wird eine strenge Typprüfung durchgeführt: formale und tatsächliche Parameter müssen identische String-Typen aufweisen. Mit {$V-} kann dagegen jede Variable des Typs *ShortString* als tatsächlicher Parameter verwendet werden, auch wenn die deklarierte Maximallänge nicht

mit der des formalen Parameters identisch ist. Dieses Vorgehen kann jedoch zu Speicherfehlern führen. Ein Beispiel:

```
var S: string[3];

procedure Test(var T: string);
begin
   T := '1234';
end;

begin
   Test(S);
end.
```

- {$X+/-}: Die Compiler-Direktive {$X+} aktiviert die Delphi-Unterstützung für nullterminierte Strings durch Verwendung der speziellen Regeln für den integrierten Typ *PChar* und Zeichen-Arrays mit der Basis Null. Diese Regeln ermöglichen den Einsatz von Arrays mit der Basis Null und von Zeichenzeigern mit den Funktionen *Write*, *Writeln*, *Val*, *Assign* und *Rename* in der Unit *System*.

Strings und Zeichen: Verwandte Themen

Die folgenden Themen in der *Object Pascal Sprachreferenz* behandeln Strings und Zeichensätze. Weitere Informationen finden Sie in Kapitel 10, »Anwendungen für den internationalen Markt«.

- Erweiterte Zeichensätze. (Hier finden Sie Informationen über internationale Zeichensätze.)
- Nullterminierte Strings. (Hier finden Sie Informationen über Zeichen-Arrays.)
- Zeichenstrings.
- Zeichenzeiger.
- Stringoperatoren.

Mit Dateien arbeiten

Dieser Abschnitt beschreibt die Arbeit mit Dateien. Dabei wird unterschieden zwischen der Bearbeitung von Dateien auf Datenträgern und Eingabe-/Ausgabeoperationen (z. B. Lese- und Schreibzugriffe auf Dateien). Der erste Abschnitt stellt die Routinen der Laufzeitbibliothek und der Windows-API für allgemeine Programmieraufgaben vor. Der nächste Abschnitt enthält eine Übersicht: »Dateitypen mit Datei-E/A«. Der letzte Abschnitt erläutert Streams, die empfohlene Methode zur Durchführung von dateibezogenen Eingabe-/Ausgabeoperationen.

Hinweis In früheren Versionen von Object Pascal wurden Dateioperationen direkt durchgeführt, d. h. ohne die jetzt üblichen Dateinamenparameter. Bei diesen älteren Versionen mußten Sie die Position der Datei ermitteln und einer Dateivariablen zuweisen, bevor Sie die Datei beispielsweise umbenennen konnten.

Dateien bearbeiten

Die Laufzeitbibliothek von Object Pascal unterstützt viele gebräuchliche Dateioperationen. Diese Prozeduren und Funktionen für dateibezogene Operationen arbeiten auf hoher Ebene. Für die meisten Routinen geben Sie einfach den Dateinamen an. Die erforderlichen Betriebssystemaufrufe werden von der Routine vorgenommen. In bestimmten Fällen können Sie auch Datei-Handles verwenden. Object Pascal stellt Routinen für die meisten Dateioperationen zur Verfügung. Ist dies nicht der Fall, werden alternative Methoden vorgestellt.

Datei löschen

Beim Löschen einer Datei wird die Datei vom Datenträger und der entsprechende Eintrag aus dem Verzeichnis des Datenträgers entfernt. Es gibt keine Operation zum Wiederherstellen gelöschter Dateien. Benutzer sollten in Anwendungen deshalb immer zum Bestätigen dieser Operation aufgefordert werden. Zum Löschen einer Datei übergeben Sie deren Namen an die Funktion *DeleteFile:*

```
DeleteFile(FileName);
```

DeleteFile gibt *True* zurück, wenn die Datei gelöscht wurde. Konnte die Datei nicht gelöscht werden, weil sie beispielsweise nicht existiert oder schreibgeschützt ist, wird *False* zurückgegeben. *DeleteFile* löscht die in *FileName* angegebene Datei vom Datenträger.

Datei suchen

Es gibt drei Routinen, mit deren Hilfe Dateien gesucht werden können: *FindFirst, FindNext* und *FindClose. FindFirst* sucht die erste Instanz eines Dateinamens mit den angegebenen Attributen im angegebenen Verzeichnis. *FindNext* gibt die nächste Fundstelle zurück, die den in einem vorhergehenden Aufruf von *FindFirst* angegebenen Attributen (einschließlich Name) entspricht. *FindClose* gibt den von *FindFirst* zugeordneten Speicher frei. In 32-Bit-Versionen von Windows sollten Sie *FindClose* immer verwenden, um *FindFirst* / *FindNext*-Operationen abzuschließen. Das Vorhandensein einer Datei können Sie mit der Funktion *FileExists* ermitteln. Diese Funktion gibt *True* zurück, wenn die Datei existiert, andernfalls *False.*

Die drei Routinen zum Suchen von Dateien nehmen einen *TSearchRec*-Parameter entgegen. *TSearchRec* definiert die zu suchenden Dateiinformationen für *FindFirst* und *FindNext*. Hier die Deklaration von *TSearchRec*:

```
type
TFileName = string;
TSearchRec = record
         Time: Integer;// Zeitstempel der Datei.
         Size: Integer;// Größe der Datei in Byte.
         Attr: Integer;// Dateiattribute der Datei.
         Name: TFileName;// DOS-Dateiname und -Dateinamenserweiterung.
         ExcludeAttr: Integer;
         FindHandle: THandle;
         FindData: TWin32FindData;// Zusätzliche Suchinformationen wie Erstellungszeit
         // der Datei, letzte Zugriffszeit sowie langer und kurzer Dateiname.
```

```
end;
```

Wird eine Datei gefunden, nehmen die Felder des *TSearchRec*-Parameters die Daten zu dieser Datei auf. In *Attr* können die folgenden Attributkonstanten oder -werte für die Dateisuche angegeben werden:

Tabelle 3.7 Attributkonstanten und -werte

Konstante	Wert	Beschreibung
faReadOnly	$00000001	Schreibgeschützt
faHidden	$00000002	Verborgen
faSysFile	$00000004	System
faVolumeID	$00000008	Volume-ID-Dateien
faDirectory	$00000010	Verzeichnisdateien
faArchive	$00000020	Archiv
faAnyFile	$0000003F	Beliebig

Um den Wert eines Attributes abzufragen, können Sie den Wert im Feld *Attr* mit Hilfe des Operators **and** mit der Attributkonstante kombinieren. Wenn die Datei das betreffende Attribut besitzt, ist das Ergebnis größer als Null. Besitzt die gefundene Datei beispielsweise das Attribut *Verborgen*, gibt der folgende Ausdruck *True* zurück: `(SearchRec.Attr and faHidden > 0)`. Attribute können kombiniert werden, indem ihre Konstanten bzw. Werte addiert werden. Um nach normalen, schreibgeschützten und verborgenen Dateien zu suchen, übergeben Sie beispielsweise `(faReadOnly + faHidden)` im Parameter *Attr*.

Beispiel In diesem Beispiel werden ein Label sowie zwei Schaltflächen namens *Search* und *Again* in einem Formular eingesetzt. Wenn der Benutzer auf die Schaltfläche *Search* klickt, wird die erste Datei im angegebenen Pfad gesucht. Der Name und die Größe dieser Datei werden dann als Beschriftung angezeigt. Immer wenn der Benutzer auf die Schaltfläche *Again* klickt, werden der Dateiname und die Größe der nächsten gefundenen Datei angezeigt:

```
var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\delphi4\bin\*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
end;

procedure TForm1.AgainClick(Sender: TObject);
begin
  if (FindNext(SearchRec) = 0)
        Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in
size';
  else
    FindClose(SearchRec);
end;
```

Dateiattribute ändern

Jede Datei besitzt verschiedene Attribute, die vom Betriebssystem in Form von Flags gespeichert werden. Mit Hilfe von Dateiattributen wird beispielsweise der schreibgeschützte oder der verborgene Status einer Datei festgelegt. Das Ändern von Dateiattributen erfolgt in drei Schritten: Lesen, Ändern, Zuweisen.

Dateiattribute lesen: Betriebssysteme speichern Dateiattribute unterschiedlich. Eine häufige Methode ist die Verwendung von Flags, die durch einzelne Bits repräsentiert werden . Die Attribute einer Datei können mit der Funktion *FileGetAttr* gelesen werden. Wenn Sie einen Dateinamen an diese Funktion übergeben, liefert die Funktion die Attribute der betreffenden Datei. Der Rückgabewert (vom Typ *Word*) besteht aus einer Reihe von Dateiattributen. Die Attribute können mit Hilfe der in *TSearchRec* definierten Konstanten und dem Operator **and** untersucht werden. Ein Rückgabewert von –1 weist auf einen Fehler hin.

Dateiattribute ändern: Da Delphi Dateiattribute stets in einer Gruppe repräsentiert, können Sie die üblichen logischen Operatoren verwenden, um die einzelnen Attribute zu ändern. Jedes Attribut besitzt einen mnemonischen Namen, der in der Unit *SysUtils* definiert ist. Das Schreibschutzattribut einer Datei können Sie beispielsweise folgendermaßen einstellen:

```
Attributes := Attributes or faReadOnly;
```

Sie können auch gleichzeitig mehrere Attribute zuweisen bzw. deaktivieren. Die folgende Zeile löscht beispielsweise die Dateiattribute *System* und *Verborgen*:

```
Attributes := Attributes and not (faSysFile or faHidden);
```

Dateiattribute zuweisen: Mit Delphi können Sie jederzeit alle Dateiattribute zuweisen, indem Sie den Namen der betreffenden Datei an die Funktion *FileSetAttr* übergeben. *FileSetAttr* stellt die Dateiattribute der angegebenen Datei ein.

Sie können die Operationen zum Lesen und Einstellen der Attribute unabhängig voneinander einsetzen, wenn Sie nur die aktuellen Werte ermitteln oder neue Werte ohne Berücksichtigung der alten Werte festlegen wollen. Sollen Attribute unter Berücksichtigung der vorherigen Werte geändert werden, müssen Sie zunächst die vorhandenen Attribute lesen, sie ändern und die geänderten Attribute dann zuweisen.

Datei umbenennen

Sie können Dateinamen mit der Funktion *RenameFile* umbenennen:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

Diese Funktion ändert den in *OldFileName* angegebenen Dateinamen in den in *NewFileName* angegebenen neuen Namen. Wurde die Operation erfolgreich durchgeführt, gibt *RenameFile* den Wert *True* zurück. Kann die Datei nicht umbenannt werden, weil beispielsweise bereits eine Datei mit dem in *NewFileName* angegebenen Namen existiert, gibt die Funktion *False* zurück. Ein Beispiel:

```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
```

Sie können Dateien mit *RenameFile* nicht über Laufwerke hinweg umbenennen (verschieben). Zu diesem Zweck müssen Sie zunächst die Datei kopieren und dann die alte Dateiversion löschen.

Hinweis *RenameFile* kapselt die Windows-API-Funktion *MoveFile*. Auch *MoveFile* kann deshalb nicht zum Verschieben über Laufwerke eingesetzt werden.

Datums-/Zeit-Routinen

Die Funktionen *FileAge, FileGetDate* und *FileSetDate* verwenden die Datums-/Zeitwerte des Betriebssystems. *FileAge* gibt den Datums-/Zeitstempel zurück bzw. -1, wenn die Datei nicht existiert. *FileSetDate* stellt den Datums-/Zeitstempel einer Datei ein. Wurde die Operation erfolgreich ausgeführt, gibt die Funktion Null zurück, andernfalls einen Windows-Fehlercode. *FileGetDate* gibt den Datums-/Zeitstempel der angegebenen Datei zurück bzw. -1, wenn das Handle ungültig ist.

Wie die meisten anderen Routinen zur Dateibearbeitung verwendet auch *FileAge* einen String-Dateinamen. *FileGetDate* und *FileSetDate* nehmen jedoch ein Windows-Handle als Parameter entgegen. Um auf ein Windows-Datei-Handle zuzugreifen, können Sie folgendermaßen vorgehen:

- Rufen Sie die Windows-API-Funktion *CreateFile* auf. *CreateFile* ist eine reine 32-Bit-Funktion, die eine Datei erstellt oder öffnet und ein Handle zurückgibt, mit dem auf die Datei zugegriffen werden kann.
- Alternativ können Sie ein *TFileStream*-Objekt instantiieren, um eine Datei zu erstellen bzw. zu öffnen. Anschließend können Sie die Eigenschaft *Handle* wie ein Windows-Datei-Handle einsetzen. Im Abschnitt »Datei-Streams verwenden« auf Seite 3-41 finden Sie weitere Informationen.

Datei kopieren

Die Laufzeitbibliothek enthält keine Routinen zum Kopieren einer Datei. Sie können zu diesem Zweck jedoch die Windows-API-Funktion *CopyFile* direkt aufrufen. Wie die meisten Routinen der Delphi-Laufzeitbibliothek nimmt auch *CopyFile* einen Dateinamen als Parameter entgegen. Ein Windows-Handle wird nicht benötigt. Beim Kopieren einer Datei werden auch die Dateiattribute der ursprünglichen Datei kopiert, die Sicherheitsattribute jedoch nicht. *CopyFile* kann auch zum Verschieben von Dateien auf ein anderes Laufwerk eingesetzt werden. Diese Operation ist weder mit der Delphi-Funktion *RenameFile* noch mit der Windows-API-Funktion *MoveFile* möglich. Weitere Informationen finden Sie in der Online-Hilfe zu Microsoft Windows.

Dateitypen mit Datei-E/A

Bei der Durchführung von Datei-E/A-Operationen können drei Dateitypen verwendet werden: klassische Pascal-Dateien, Windows-Datei-Handles und Datei-Stream-Objekte. Dieser Abschnitt enthält eine zusammenfassende Darstellung dieser Typen.

Klassische Pascal-Dateien: Dieser Typ wird mit den alten Dateivariablen verwendet. Das Format lautet normalerweise `F: Text` oder `F: file`. Es gibt diesen Typ in den drei Klassen *Typisiert, Text* und *Untypisiert*, die von einer Reihe von Delphi-Routinen zur

Dateibearbeitung (beispielsweise *AssignPrn* und *Writeln*) verwendet werden. Diese Dateitypen sind veraltet und zu Windows-Datei-Handles nicht kompatibel. Wenn Sie mit diesen Dateitypen arbeiten müssen, finden Sie in der »Object Pascal Sprachreferenz« ausführliche Informationen.

Windows-Datei-Handles: Die Datei-Handles von Object Pascal kapseln den Datei-Handle-Typ von Windows. Dateibearbeitungsroutinen der Laufzeitbibliothek, die Windows-Datei-Handles verwenden, kapseln normalerweise Windows-API-Funktionen. Die Funktion *FileRead* ruft beispielsweise die Windows-Funktion *ReadFile* auf. Da die Delphi-Funktionen auf der Syntax von Object Pascal basieren und gelegentlich Standard-Parameterwerte bereitstellen, bilden sie eine komfortable Schnittstelle zur Windows-API. Der Einsatz dieser Routinen ist ohne besonderen Zusatzaufwand möglich. Wenn Sie mit den Datei-Routinen der Windows-API vertraut sind, können Sie diese für Datei-E/A-Operationen einsetzen.

Datei-Streams: Datei-Streams sind Objektinstanzen der VCL-Klasse *TFileStream*, mit deren Hilfe auf Informationen in Dateien auf Datenträgern zugegriffen werden kann. Datei-Streams bilden eine portierbare Lösung für Datei-E/A-Operationen auf hoher Ebene. *TFileStream* besitzt eine Eigenschaft namens Handle, die den Zugriff auf das Windows-Datei-Handle ermöglicht. Der folgende Abschnitt erläutert *TFileStream*.

Datei-Streams verwenden

TFileStream ist eine VCL-Klasse zur Objekt-Repräsentation von Datei-Streams auf hoher Ebene. *TFileStream* bietet folgende Funktionalität: Persistenz, Interaktion mit anderen Streams und Datei-E/A.

- *TFileStream* ist ein Nachfahre der Stream-Klassen. Ein Vorteil beim Einsatz von Datei-Streams besteht deshalb in der automatischen Vererbung der Persistenzunterstützung. Die Stream-Klassen können mit den *TFiler*-Klassen *TReader* und *TWriter* arbeiten, um Stream-Objekte auf den Datenträger zu schreiben bzw. von diesem zu lesen. Wenn also ein Datei-Stream vorliegt, können Sie denselben Quelltext für den VCL-Mechanismus für Stream-Operationen verwenden. Weitere Informationen zum Stream-System der VCL finden Sie in der Online-Hilfe in der VCL-Referenz unter
  »TStream«,
  »TFiler«,
  »TReader«,
  »TWriter« und
  »TComponent«.
- *TFileStream* kann problemlos mit anderen Stream-Klassen interagieren. Wollen Sie beispielsweise einen dynamischen Speicherblock auf einen Datenträger schreiben, können Sie diese Operation mit einem *TFileStream*- und einem *TMemoryStream*-Objekt durchführen.
- *TFileStream* stellt die grundlegenden Methoden und Eigenschaften für Datei-E/A-Operationen bereit. Die folgenden Abschnitte konzentrieren sich auf diesen Aspekt von Datei-Streams.

Dateien erstellen und öffnen

Um eine Datei zu erstellen bzw. zu öffnen und Zugriff auf das Handle für diese Datei zu erhalten, instantiieren Sie einfach ein *TFileStream*-Objekt. Dieses öffnet oder erstellt die angegebene Datei und stellt Methoden für Lese- und Schreiboperationen zur Verfügung. Kann die Datei nicht geöffnet werden, löst *TFileStream* eine Exception aus.

```
constructor Create(const filename: string; Mode: Word);
```

Der Parameter *Mode* gibt an, wie die Datei beim Erstellen des Datei-Streams geöffnet werden soll. Er enthält den Öffnungsmodus und den Freigabemodus. Beide Werte werden mit einem logischen ODER verknüpft. Für den Öffnungsmodus sind folgende Werte möglich:

Wert	Bedeutung
fmCreate	*TFileStream* öffnet eine Datei mit dem angegebenen Namen. Existiert bereits eine Datei mit diesem Namen, wird sie im Schreibmodus geöffnet.
fmOpenRead	Öffnet die Datei schreibgeschützt.
fmOpenWrite	Öffnet die Datei nur für Schreiboperationen. Beim Schreiben in die Datei wird der bisherige Inhalt vollständig ersetzt.
fmOpenReadWrite	Öffnet die Datei zur Bearbeitung. Beim Schreiben in die Datei wird der bisherige Inhalt also nicht vollständig ersetzt.

Der Freigabemodus muß einen der folgenden Werte besitzen:

Wert	Bedeutung
fmShareCompat	Der Freigabemodus ist mit dem Öffnen von FCBs kompatibel.
fmShareExclusive	Andere Anwendungen können die Datei nicht öffnen.
FmShareDenyWrite	Andere Dateien können die Datei zum Lesen, nicht aber zum Schreiben öffnen.
FmShareDenyRead	Andere Anwendungen können die Datei zum Schreiben, nicht aber zum Lesen öffnen.
FmShareDenyNone	Andere Anwendungen werden nicht an Lese- oder Schreibzugriffen auf die Datei gehindert.

Die Konstanten für den Öffnungs- bzw. Freigabemodus sind in der Unit *SysUtils* definiert.

Datei-Handle verwenden

Wenn Sie *TFileStream* instantiieren, erhalten Sie Zugriff auf das Datei-Handle. Das Datei-Handle wird in der Eigenschaft *Handle* gespeichert. Handle ist schreibgeschützt und gibt den Modus an, in dem die Datei geöffnet wurde. Wenn Sie die Attribute des Datei-Handles ändern wollen, müssen Sie ein neues Datei-Stream-Objekt erstellen.

Einige Routinen zur Dateibearbeitung benötigen als Parameter ein Windows-Datei-Handle. Sobald Sie einen Datei-Stream angelegt haben, können Sie die Eigenschaft

Handle jederzeit wie ein Windows-Datei-Handle nutzen. Beachten Sie jedoch, daß Datei-Streams das Datei-Handle schließen, wenn das Objekt freigegeben wird (im Unterschied zu Handle-Streams).

Dateien lesen und schreiben

TFileStream besitzt verschiedene Methoden zum Lesen aus und zum Schreiben in Dateien. Diese Methoden werden nach folgenden Kriterien unterschieden:

- Rückgabe der Anzahl gelesener bzw. geschriebener Bytes.
- Anzahl der Bytes muß bekannt sein.
- Bei Fehlern wird eine Exception ausgelöst.

Read ist eine Funktion, die bis zu *Count Bytes* aus der dem Datei-Stream zugeordneten Datei in Buffer liest, beginnend an der aktuellen Position. *Read* korrigiert dann den Dateizeiger nach Maßgabe der tatsächlich übertragenen Bytes. Der Prototyp für *Read* lautet:

```
function Read(var Buffer; Count: Longint): Longint; override;
```

Read ist hilfreich, wenn die Anzahl der Bytes in der Datei nicht bekannt ist. *Read* gibt die Anzahl der tatsächlich übertragenen Bytes zurück. Dieser Wert kann kleiner als *Count* sein, wenn ein Dateiendezeichen auftritt.

Write ist eine Funktion, die *Count Bytes* aus Buffer in die Datei schreibt, die dem Datei-Stream zugeordnet ist. Die Operation beginnt an der aktuellen Position. Der Prototyp für *Write* lautet:

```
function Write(const Buffer; Count: Longint): Longint; override;
```

Nach dem Schreiben in die Datei setzt *Write* den Dateizeiger um die Anzahl von Bytes vor, die geschrieben wurde. Zurückgegeben wird die Anzahl der tatsächlich geschriebenen Bytes. Dieser Wert kann kleiner als *Count* sein, wenn das Ende des Puffers bereits vorher erreicht wird.

Die Gegenstücke heißen *ReadBuffer* und *WriteBuffer*. Diese Methoden geben im Unterschied zu den Funktionen *Read* und *Write* nicht die Anzahl der gelesenen bzw. geschriebenen Bytes zurück. Sie sind hilfreich, wenn die Anzahl der Bytes erforderlich und bekannt ist, beispielsweise beim Lesen einer Struktur. *ReadBuffer* und *WriteBuffer* lösen eine Exception aus, wenn die Fehler *EReadError* und *EWriteError* auftreten. Die Funktionen *Read* und *Write* lösen in diesen Fällen keine Exception aus. Die Prototypen für *ReadBuffer* und *WriteBuffer* lauten folgendermaßen:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

Diese Methoden rufen die Methoden *Read* und *Write* auf, um die eigentlichen Lese- bzw. Schreiboperationen durchzuführen.

Strings lesen und schreiben

Wenn Sie einen String an eine Lese- oder Schreibfunktion übergeben, müssen Sie die zu verwendende Syntax kennen. Die Parameter *Buffer* der Lese- bzw. Schreibroutinen

sind **var**- bzw. **const**-Parameter. Da es sich um untypisierte Parameter handelt, nimmt die Routine also die Adresse einer Variablen entgegen.

Bei der Arbeit mit Strings werden normalerweise lange Strings verwendet. Mit langen Strings erhalten Sie hier aber nicht das richtige Ergebnis. Lange Strings enthalten die Größe, einen Referenzzähler und einen Zeiger auf die Zeichen im String. Das Dereferenzieren eines langen Strings bezieht sich deshalb nicht nur auf das Zeigerelement. Sie müssen den String in einen *Pointer* oder *PChar* umwandeln, bevor Sie ihn dereferenzieren. Ein Beispiel:

```
procedure cast-string;
var
  fs: TFileStream;
  s: string = 'Hello';
begin
  fs := TFileStream.Create('Temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// Liefert Datenmüll
  fs.Write(PChar(s)^, Length(s));// Das ist die richtige Methode
end;
```

Datei durchsuchen

Typische Mechanismen für die Datei-E/A enthalten einen Prozeß zum Durchsuchen von Dateien, damit Lese- und Schreiboperationen an einer bestimmten Position in der Datei durchgeführt werden können. Zu diesem Zweck stellt *TFileStream* die Methode *Seek* bereit. Der Prototyp für *Seek* lautet:

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

Der Parameter *Origin* gibt an, wie der Parameter *Offset* interpretiert werden soll. *Origin* muß einen der folgenden Werte enthalten:

Wert	Bedeutung
soFromBeginning	*Offset* wird ausgehend vom Beginn der Ressource ermittelt. *Seek* bewegt den Dateizeiger zur Position *Offset*. *Offset* muß größer oder gleich Null sein.
SoFromCurrent	*Offset* bezieht sich auf die aktuelle Position in der Ressource. *Seek* verschiebt den Dateizeiger nach Position + *Offset*.
SoFromEnd	*Offset* wird ausgehend vom Ende der Ressource ermittelt. *Offset* muß kleiner oder gleich Null sein, um eine Position anzugeben, die vor dem Ende der Datei liegt.

Seek setzt die aktuelle *Position* im Stream zurück, indem der Zeiger um den angegebenen Offset verschoben wird. *Seek* gibt den neuen Wert der Eigenschaft *Position* zurück, also die neue, aktuelle Position in der Ressource.

Dateiposition und -größe

TFileStream besitzt Eigenschaften, in denen die aktuelle Position des Dateizeigers und die Größe der Datei gespeichert ist. Diese Eigenschaften werden von der Methode *Seek* und von den Routinen zum Lesen und Schreiben verwendet.

Die Eigenschaft *Position* von *TFileStream* dient der Angabe des aktuellen Offset im Stream in Byte (ausgehend vom Beginn der Stream-Daten). Die Deklaration von *Position* lautet:

```
property Position: Longint;
```

Die Eigenschaft *Size* gibt die Größe des Streams in Byte an. Sie wird als Endemarke verwendet, um die Datei abzuschneiden. Die Deklaration von *Size* lautet:

```
property Size: Longint;
```

Size wird intern von Routinen verwendet, die aus dem Stream lesen und in den Stream schreiben.

Durch eine Zuweisung an die Eigenschaft *Size* wird die Größe der Datei geändert. Wenn die Eigenschaft *Size* einer Datei nicht geändert werden kann, wird eine Exception ausgelöst. Dies ist beispielsweise der Fall, wenn Sie die Größe einer Datei ändern wollen, die im Modus *fmOpenRead* geöffnet wurde.

Kopieren

CopyFrom kopiert die angegebene Anzahl Bytes aus einem Datei-Stream in einen anderen.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Durch den Einsatz von *CopyFrom* ist es nicht mehr erforderlich, einen Puffer zu erstellen, Daten in diesen einzulesen, anschließend aus dem Puffer zu schreiben und abschließend den Puffer freizugeben, um Daten zu kopieren.

CopyFrom kopiert *Count Bytes* aus *Source* in den Stream. *CopyFrom* verschiebt die aktuelle Position um *Count Bytes* und gibt die Anzahl der kopierten Bytes zurück. Wenn *Count* Null ist, setzt *CopyFrom* die Position in *Source* auf Null, bevor der Inhalt von *Source* vollständig eingelesen und dann in den Stream kopiert wird. Ist *Count* größer oder kleiner Null, liest *CopyFrom* ausgehend von der aktuellen Position in *Source*.

Neue Datentypen definieren

Object Pascal umfaßt viele vordefinierte Datentypen. Sie können diese Datentypen verwenden, um neue Datentypen zu erstellen, welche die spezifischen Anforderungen einer Anwendung erfüllen. Eine Übersicht über die verfügbaren Typen finden Sie in der *Object Pascal Sprachreferenz*.

Kapitel

4

Anwendungen, Komponenten und Bibliotheken erstellen

Dieses Kapitel gibt einen Überblick über die Erstellung von Anwendungen, Bibliotheken und Komponenten.

Anwendungen erstellen

Delphi wird hauptsächlich zum Entwerfen und Erstellen von Windows-Anwendungen eingesetzt. Die folgenden drei Anwendungsarten können erstellt werden:

- Windows-GUI-Anwendungen
- Konsolenanwendungen
- Service-Anwendungen

Windows-Anwendungen

Wenn Sie ein Projekt compilieren, wird eine ausführbare Datei (EXE) erstellt. Sie enthält normalerweise die Grundfunktionen der Anwendung. Einfache Programme bestehen oft nur aus einer ausführbaren Datei. Sie können den Funktionsumfang einer Anwendung erweitern, indem Sie aus der EXE heraus DLLs, Packages und andere Unterstützungsdateien aufrufen.

Für Windows-Anwendungen können die beiden folgenden Oberflächenmodelle verwendet werden:

- SDI (Single Document Interface)
- MDI (Multiple Document Interface)

Das Entwurfszeitverhalten des Projekts und das Laufzeitverhalten der Anwendung kann mit Hilfe verschiedener Projektoptionen in der IDE gesteuert werden.

Benutzeroberflächen

Jedes Formular in einer Anwendung kann entweder als MDI-Fenster (Multiple Document Interface) oder als SDI-Fenster (Single Document Interface) implementiert werden. In einer MDI-Anwendung können mehrere Dokumentenfenster in einem übergeordneten Fenster geöffnet werden. Dieses Modell wird für Anwendungen wie Tabellenkalkulationen oder Textverarbeitungen verwendet. Im Gegensatz dazu besteht eine SDI-Anwendung normalerweise aus einer einzigen Dokumentenansicht. Um ein Formular als SDI-Fenster zu verwenden, setzen Sie die Eigenschaft *FormStyle* des Formularobjekts auf *fsNormal*.

Detaillierte Informationen zur Erstellung von Benutzeroberflächen finden Sie in Kapitel 5, »Die Benutzeroberfläche erstellen«.

SDI-Anwendungen

So erstellen Sie eine neue SDI-Anwendung:

1. Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.
2. Aktivieren Sie die Registerkarte *Projekte*, und wählen Sie den Eintrag *SDI-Anwendung*.
3. Bestätigen Sie mit *OK*.

Da die Eigenschaft *FormStyle* des Formulars den Standardwert *fsNormal* hat, werden alle neuen Anwendungen automatisch mit dem SDI-Modell erstellt.

MDI-Anwendungen

So erstellen Sie eine neue MDI-Anwendung:

1. Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.
2. Aktivieren Sie die Registerkarte *Projekte*, und wählen Sie den Eintrag *MDI-Anwendung*.
3. Bestätigen Sie mit *OK*.

MDI-Anwendungen erfordern eine gründlichere Planung und sind schwieriger zu erstellen als SDI-Anwendungen. Eine MDI-Anwendung besteht normalerweise aus einem Client-Fenster und mehreren untergeordneten Dokumentenfenstern. Die untergeordneten Formulare sind Bestandteil des Hauptformulars. Mit Hilfe der Eigenschaft *FormStyle* des jeweiligen TForm-Objekts läßt sich festlegen, ob es als untergeordnetes Formular (*fsMDIChild*) oder als Hauptformular (*fsMDIForm*) verwendet wird. Es empfiehlt sich, für die untergeordneten Fenster eine eigene Basisklasse zu erstellen und die einzelnen Formulare von ihr abzuleiten. Sie erhalten dadurch ein einheitliches Erscheinungsbild und müssen nicht für jedes untergeordnete Formular die Eigenschaften erneut festlegen.

IDE-, Projekt- und Compiler-Optionen festlegen

Mit Hilfe des Dialogfeldes *Projekt / Projektoptionen* können Sie verschiedene Einstellungen für ein Projekt festlegen. Detaillierte Informationen zu diesen Einstellungen finden Sie in der Online-Hilfe.

Standardeinstellungen festlegen

Wenn Sie das Kontrollkästchen *Vorgabe* in der linken unteren Ecke des Dialogfeldes Projektoptionen aktivieren, werden für jedes neue Projekt automatisch die aktuellen Einstellungen aller Registerkarten verwendet.

Quelltextvorlagen

Quelltextvorlagen bestehen aus häufig verwendete Codestrukturen, die Sie in den Quelltext einfügen und dann mit den entsprechenden Werten füllen. Wenn Sie beispielsweise eine **for**-Schleife benötigen, können Sie die folgende Quelltextvorlage einfügen:

```
for  :=  to  do
begin

end;
```

Wenn Sie eine Quelltextvorlage in den Editor einfügen wollen, drücken Sie *Strg+J* und wählen in der Dropdown-Liste den gewünschten Eintrag. Sie können auch jederzeit eigene Vorlagen in die Liste aufnehmen. Gehen Sie dazu folgendermaßen vor:

1. Wählen Sie *Tools* / *Umgebungsoptionen*.
2. Klicken Sie auf das Register *Programmierhilfe*.
3. Klicken Sie im Bereich *Quelltextvorlagen* auf *Hinzufügen*.
4. Geben Sie ein Kürzel und eine kurze Beschreibung für die neue Vorlage ein.
5. Geben Sie den Quelltext in das Textfeld *Code* ein.
6. Klicken Sie auf *OK*.

Konsolenanwendungen

Konsolenanwendungen sind Windows-Anwendungen (32 Bit), die ohne grafische Benutzeroberfläche ausgeführt werden (üblicherweise in einem Konsolenfenster). Sie erfordern normalerweise nur wenige Benutzereingaben und verfügen über einen begrenzten Funktionsumfang.

Neue Konsolenanwendungen werden mit dem Konsolen-Experten erstellt. Um diesen Experten zu aktivieren, wählen Sie *Datei / Neu* und klicken im Dialogfeld *Objektgalerie* auf *Konsolen-Experte*.

Service-Anwendungen

Service-Anwendungen nehmen Anforderungen von Client-Anwendungen entgegen, verarbeiten diese und geben Informationen an den Client zurück. Sie werden normalerweise ohne Benutzereingabe im Hintergrund ausgeführt. Beispiele hierfür sind Web-, FTP- oder E-Mail-Server.

Sie können in Delphi auch Win32-Systemdienste erstellen. Wählen Sie dazu *Datei / Neu*, und klicken Sie anschließend im Dialogfeld *Objektgalerie* auf *Service-Anwendung*. Dadurch wird dem Projekt eine globale Variable mit dem Namen *Application* (Typ *TServiceApplication*) hinzugefügt.

Im Designer wird für den neuen Service (*TService*) ein Fenster angezeigt. Dieses Fenster verfügt wie ein Formular über verschiedene Eigenschaften und Ereignisse, die Sie im Objektinspektor bearbeiten können. Sie können auch weitere Systemdienste hinzufügen, indem Sie *Datei / Neu* und anschließend *Service* wählen. Fügen Sie aber auf keinen Fall Dienste zu einem Programm hinzu, das keine Service-Anwendung ist. Das *TService*-Objekt wird zwar hinzugefügt, die Anwendung generiert aber nicht die Ereignisse, die für die erforderlichen Windows-Aufrufe unbedingt notwendig sind.

Nachdem Sie die Service-Anwendung erstellt haben, können Sie die Dienste mit Hilfe des Dienstkontroll-Managers installieren. Andere Anwendungen können diese Dienste dann nutzen, indem Sie entsprechende Anforderungen an den Dienstkontroll-Manager senden.

Um die Dienste Ihrer Anwendung zu installieren, führen Sie sie unter Verwendung der Option /INSTALL aus. Die Anwendung installiert dann ihre Dienste. Wenn sie beendet wird, erscheint im Falle einer erfolgreichen Installation eine entsprechende Bestätigungsmeldung. Sie können die Anzeige dieser Meldung unterdrücken, indem Sie die Service-Anwendung mit der Option /SILENT starten.

Um die Dienste zu de-installieren, starten Sie sie von der Befehlszeile aus mit der Option /UNINTALL. (Auch in diesem Fall können Sie die Anzeige der Bestätigungsmeldung mit Hilfe der Option /SILENT unterdrücken.).

Beispiel Der Service im folgenden Beispiel enthält eine *TServerSocket*-Komponente, deren Anschluß (Eigenschaft *Port*) auf 80 gesetzt ist. Dies ist der Standardanschluß für WWW-Server und -Browser. Die Komponente erstellt im Verzeichnis C:\TEMP eine Textdatei mit dem Namen WEBLOGXXX.LOG (XXX ist der Wert von *ThreadID*). Jedem Anschluß sollte immer nur ein Server zugeordnet sein. Wenn auf Ihrem System ein Web-Server installiert ist, müssen Sie daher sicherstellen, daß er deaktiviert (der Dienst beendet) ist.

Und so können Sie sich die Ausgabe des Beispiels anzeigen lassen: öffnen Sie auf dem lokalen Computer einen Browser, und geben Sie als Adresse `localhost` ein. Der Browser zeigt zwar möglicherweise eine Fehlermeldung an, aber die Datei WEBLOGXXX.LOG befindet sich im Verzeichnis C:\TEMP.

1. Um diesen Service zu erstellen, wählen Sie *Datei / Neu* und klicken im Dialogfeld *Objektgalerie* auf Service-Anwendung. Daraufhin wird ein Fenster mit dem Namen *Service1* geöffnet. Fügen Sie eine *ServerSocket*-Komponente aus der Registerkarte *Internet* der Komponentenpalette in das Service-Fenster (Service1) ein.

2 Fügen Sie der Klasse *TService1* ein als **private** deklariertes Datenelement des Typs *TMemoryStream* hinzu. Der **interface**-Abschnitt der Unit muß nun folgendermaßen aussehen:

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;

type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Fügen Sie hier diese Zeile ein
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;

var
  Service1: TService1;
```

3 Wählen Sie nun die Komponente *ServerSocket1* aus, die Sie in Schritt 1 hinzugefügt haben. Doppelklicken Sie dann im Objektinspektor auf das Ereignis *OnClientRead*, und schreiben Sie folgende Ereignisbehandlungsroutine:

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;

begin
  Buffer := nil;

while Socket.ReceiveLength > 0 do begin
    try
      Buffer := AllocMem(Socket.ReceiveLength);
      Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
      Stream.Write(Buffer^, StrLen(Buffer));
    finally
      FreeMem(Buffer);
    end;

  Stream.Seek(0, soFromBeginning);
    Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
  end;
end;
```

4 Wählen Sie *Service1* aus, indem Sie auf den Client-Bereich des Fensters klicken (nicht auf die *ServerSocket*-Komponente). Doppelklicken Sie anschließend im Objektinspektor auf das Ereignis *OnExecute*, und fügen Sie die folgende Ereignisbehandlungsroutine hinzu:

```
procedure TService1.Service1Execute(Sender: TService);
```

```
begin
  Stream := TMemoryStream.Create;
  try
    ServerSocket1.Port := 80; // WWW port
    ServerSocket1.Active := True;

    while not Terminated do begin
      ServiceThread.ProcessRequests(False);
    end;

    ServerSocket1.Active := False;
  finally
    Stream.Free;
  end;
end;
```

Beim Schreiben von Service-Anwendungen müssen Sie folgende Punkte beachten:

- Service-Threads
- Service-Namenseigenschaften
- Services testen

Service-Threads

Da jeder Service in einem eigenen Thread (*TServiceThread*) ausgeführt wird, müssen Sie bei Anwendungen mit mehreren Services auf eine Thread-sichere Implementierung achten. Die Klasse *TServiceThread* ist so angelegt, daß der Service in der *OnExecute*-Ereignisbehandlungsroutine von *TService* implementiert werden kann. Der Service-Thread verfügt über eine eigene *Execute*-Methode. Sie enthält eine Schleife, in der die Service-Ereignisbehandlungsroutinen für *OnStart* und *OnExecute* vor dem Verarbeiten neuer Anforderungen aufgerufen werden. Die Verarbeitung von Service-Anforderungen kann sehr viel Zeit in Anspruch nehmen, und die Service-Anwendung kann gleichzeitig Anforderungen von mehreren Clients erhalten. Es ist daher am effizientesten, für jede Anforderung einen neuen Thread (abgeleitet von TThread, nicht von *TServiceThread*) abzuspalten und die Service-Implementierung in der Methode Execute des neuen Threads vorzunehmen. Dadurch können in der *Execute*-Schleife des Service-Threads ständig neue Anforderungen bearbeitet werden, ohne daß auf das Ende der *OnExecute*-Ereignisbehandlungsroutine gewartet werden muß. Beachten Sie dazu das folgende Beispiel.

Beispiel Der Service in diesem Beispiel gibt innerhalb eines *TThread*-Objekts alle 500 Millisekunden einen Ton aus. Er sorgt für das Pausieren, Stoppen und Fortsetzen des Threads, wenn dies angefordert wird.

1. Wählen Sie *Datei / Neu*, und klicken Sie im Dialogfeld *Objektgalerie* auf das Symbol *Service-Anwendung*. Daraufhin wird ein Fenster mit dem Namen *Service1* geöffnet.
2. Deklarieren Sie im **interface**-Abschnitt der Unit einen neuen Nachkommen der Klasse *TThread* mit dem Namen *TSparkyThread*. Dieser Thread führt alle Aktivitäten des Service aus. Die Deklaration von *TSparkyThread* lautet folgendermaßen:

```
TSparkyThread = class(TThread)
    public
      procedure Execute; override;
```

```
  end;
```

3 Erstellen Sie nun im Implementierungsabschnitt der Unit eine globale Variable für eine Instanz von *TSparkyThread:*

```
var
  SparkyThread: TSparkyThread;
```

4 Fügen Sie dem Implementierungsabschnitt für die *Execute*-Methode von *TSparky-Thread* (die Thread-Funktion) folgenden Quelltext hinzu:

```
procedure TSparkyThread.Execute;
begin
  while not Terminated do
  begin
    Beep;
    Sleep(500);
  end;
end;
```

5 Wählen Sie das Service-Fenster (*Service1*) aus, und doppelklicken Sie im Objektinspektor auf das Ereignis *OnStart*. Schreiben Sie dann die folgende *OnStart*-Ereignisbehandlungsroutine:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
```

6 Doppelklicken Sie im Objektinspektor auf das Ereignis *OnContinue*, und schreiben Sie folgende Behandlungsroutine für dieses Ereignis:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
  SparkyThread.Resume;
  Continued := True;
end;
```

7 Doppelklicken Sie im Objektinspektor auf das Ereignis *OnPause*, und schreiben Sie folgende Behandlungsroutine für dieses Ereignis:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
```

8 Doppelklicken Sie im Objektinspektor auf das Ereignis *OnStop*, und schreiben Sie folgende Behandlungsroutine für dieses Ereignis:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

Die Entscheidung, ob für jede Anforderung ein neuer Thread eingerichtet werden soll, ist von mehreren Faktoren abhängig, beispielsweise von der Art des zu imple-

mentierenden Systemdienstes, von der erwarteten Anzahl der Verbindungen und von der Anzahl der Prozessoren in dem Computer, der den Service ausführt.

Service-Namenseigenschaften

Die VCL-Klassen *TService* und *TDependency* werden zum Erstellen von Service-Anwendungen verwendet. Dabei können jedoch die unterschiedlichen Namenseigenschaften zu einer gewissen Verwirrung führen. Dieser Abschnitt beschreibt die Unterschiede zwischen den Namenseigenschaften.

Ein Service (*TService*) hat einen Benutzernamen (den sogenannten Service-Startnamen), der mit einem Kennwort verknüpft ist, einen Anzeigenamen für Verwaltungs- und Editorfenster und natürlich seinen eigentlichen Namen (den Service-Namen). Abhängigkeiten (*TDependency*) können Services oder Lastverteilungsgruppen sein. Auch sie verfügen über Namen und Anzeigenamen. Alle Service-Objekte sind von der Basisklasse *TComponent* abgeleitet und erben von dieser die Eigenschaft *Name*. Es folgt eine Übersicht über die verschiedenen Namenseigenschaften.

TDependency-Namenseigenschaften

Die Eigenschaft *DisplayName* gibt sowohl den Anzeigenamen als auch den eigentlichen Namen des Objekts an. Sie ist fast immer mit der Eigenschaft *Name* identisch.

TService-Namenseigenschaften

Die Eigenschaft *Name* wird von *TComponent* geerbt. Sie gibt den Namen der Komponente und den Namen des Service an. Bei Abhängigkeiten, die als Service verwendet werden, entspricht sie den Eigenschaften *Name* und *DisplayName* von *TDependency*.

DisplayName wird im Fenster des Service-Managers angezeigt. Er ist nicht immer mit dem eigentlichen Service-Namen (*TService.Name*, *TDependency.DisplayName*, *TDependency.Name*) identisch. Zu beachten ist, daß sich die *TDependency*-Eigenschaft *DisplayName* normalerweise von der *TService*-Eigenschaft *DisplayName* unterscheidet.

Service-Startnamen unterscheiden sich von den Anzeigenamen und den Service-Namen. *ServiceStartName* ist der im Dialogfeld *Starten* (Dienstkontroll-Manager) eingegebene Benutzername.

Services testen

Die einfachste Methode, eine Service-Anwendung zu testen, besteht darin, bei der Ausführung des Dienstes eine Verbindung mit dem Prozeß herzustellen. Dazu wählen Sie *Run / Mit Prozeß verbinden* und wählen dann aus der Liste der verfügbaren Prozesse die Service-Anwendung.

In manchen Fällen kann dies allerdings wegen unzureichender Zugriffsrechte zu einem Fehler führen. Rufen Sie dann den Dienstkontroll-Manager auf, um die Ausführung des Dienstes im Debugger zu ermöglichen:

1 Erstellen Sie zuerst an der nachfolgend genannten Stelle der Systemregistrierung einen Schlüssel namens **Image File Execution Options**:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`

2 Erstellen Sie einen Unterschlüssel mit dem Namen Ihrer Service-Anwendung (beispielsweise MYSERV.EXE). Fügen Sie diesem Unterschlüssel einen String-Wert des Typs REG_SZ mit dem Namen *Debugger* hinzu. Geben Sie dabei für den Debugger den vollständigen Verzeichnispfad an.

3 Markieren Sie im Dienstkontroll-Panel Ihre Service-Anwendung, dann klicken Sie auf *Startup* und markieren das Kontrollkästchen *Allow Service to Interact with Desktop*.

Packages und DLLs erstellen

DLLs sind compilierte Module, die eine Anwendung um zusätzliche Funktionen ergänzen.

Packages sind spezielle DLLs, die von Delphi-Anwendungen, der IDE oder beiden verwendet werden. Es gibt zwei Arten von Packages: Laufzeit- und Entwurfszeit-Packages. Laufzeit-Packages stellen einem Programm Funktionen zur Verfügung, während es ausgeführt wird. Entwurfszeit-Packages erweitern den Funktionsumfang der IDE.

Weitere Informationen zu Packages finden Sie in Kapitel 9, »Packages und Komponenten«.

Unterschiede zwischen Packages und DLLs

Bei den meisten Delphi-Anwendungen bieten Packages eine größere Flexibilität und sind einfacher zu erstellen als DLLs. In den folgenden Situationen verdienen jedoch DLLs den Vorzug:

- Das Codemodul wird von einer nicht mit Delphi erstellten Anwendung aufgerufen.
- Der Funktionsumfang eines Web-Servers soll erweitert werden.
- Das Codemodul soll von Fremdentwicklern verwendet werden.
- Das Projekt ist ein OLE-Container.

Datenbankanwendungen erstellen

Delphi unterstützt die Entwicklung komplexer Datenbankanwendungen. Integrierte Tools ermöglichen die Einrichtung von Verbindungen zu Oracle-, Sybase-, Informix-, dBASE-, Paradox- und anderen Servern und gewährleisten eine transparente gemeinsame Datennutzung in verschiedenen Anwendungen. Mit Hilfe der BDE (Borland Database Engine) können Desktop-Anwendungen zu Client/Server-Anwendungen erweitert werden.

Tools wie der Datenbank-Explorer helfen Ihnen bei der Entwicklung von Datenbankanwendungen. Der Datenbank-Explorer ist ein hierarchisch strukturierter Browser,

mit dem serverspezifische Objekte wie Tabellen, Felder, Stored Procedures, Trigger, Referenzen und Indexbeschreibungen überprüft und geändert werden können.

Da der Datenbank-Explorer eine dauerhafte Verbindung zu einer Datenbank einrichtet, haben Sie folgende Möglichkeiten:

- Erstellen und Verwalten von Datenbank-Aliasen.
- Anzeigen von Schemadaten einer Datenbank, z. B. von Tabellen, Stored Procedures und Trigger.
- Anzeigen von Tabellenobjekten wie Feldern und Indizes.
- Erstellen, Anzeigen und Ändern von Tabellendaten.
- Direktes Abfragen einer Datenbank mit SQL-Anweisungen.
- Erstellen und Verwalten von Daten-Dictionaries zum Speichern von Attributen.

Detaillierte Informationen zur Erstellung von Datenbank-Client-Anwendungen und Anwendungsservern finden Sie in Teil II dieses Handbuchs.

Verteilte Anwendungen erstellen

Verteilte Anwendungen können auf unterschiedlichen Computern und Plattformen ausgeführt werden. Die verschiedenen Bestandteile arbeiten (normalerweise über ein Netzwerk) zusammen, um bestimmte Funktionen durchzuführen. So benötigt beispielsweise eine Anwendung für eine landesweite Firma einzelne Client-Anwendungen für die Niederlassungen, einen Haupt-Server für die Verarbeitung der Client-Anforderungen und eine Schnittstelle zu einer Datenbank, in der die Informationen gespeichert werden. Mit Hilfe einer verteilten Client-Anwendung (z. B. als Web-basierte Anwendung) kann in dieser Situation das Pflegen und Aktualisieren der Clients spürbar vereinfacht werden.

In Delphi stehen für verteilte Anwendungen folgende Implementierungsmodelle zur Verfügung:

- TCP/IP-Anwendungen
- COM- und DCOM-Anwendungen
- CORBA-Anwendungen
- Datenbankanwendungen

Verteilte TCP/IP-Anwendungen erstellen

TCP/IP ist ein Kommunikationsprotokoll, mit dessen Hilfe Anwendungen über ein Netzwerk kommunizieren können. Das Protokoll stellt lediglich die Transportschicht zur Verfügung und erfordert keine spezielle Architektur bei den verteilten Anwendungen.

Die rasche Zunahme von Internet-Zugängen hat dazu geführt, daß die meisten Computer heutzutage über irgendeine Form von TCP/IP-Zugriff verfügen, der das Verteilen und Einrichten der Anwendung erleichtert.

TCP/IP-Anwendungen können nachrichtenbasiert (z. B. ein Web-Server, der HTTP-Anfragen bearbeitet) oder objektorientiert (z. B. verteilte Datenbankanwendungen, die über Windows-Sockets kommunizieren) erstellt werden.

Am einfachsten fügen Sie Ihren Anwendungen TCP/IP-Funktionen mit Hilfe von Client- oder Server-Sockets hinzu. Sie können auch Web-Server durch Anwendungen erweitern , indem Sie ein CGI-Skript oder eine DLL erstellen. Außerdem können TCP/IP-basierte Datenbankanwendungen erstellt werden.

Sockets in Anwendungen verwenden

Mit den beiden VCL-Klassen *TClientSocket* und *TServerSocket* können Sie TCP/IP-Socket-Verbindungen für die Kommunikation mit anderen Remote-Anwendungen erstellen. Detaillierte Informationen über Sockets finden Sie in Kapitel 30, »Arbeiten mit Sockets«.

Web-Server-Anwendungen erstellen

Um eine neue Web-Server-Anwendung zu erstellen, wählen Sie *Datei / Neu*. Klicken Sie im Dialogfeld *Objektgalerie* auf *Web-Server-Anwendung*, und wählen Sie anschließend im Dialogfeld *Neue Web-Server-Anwendung* eine der folgenden Optionen:

- ISAPI/NSAPI-DLL
- CGI, ausführbare Datei
- Win-CGI, ausführbare Datei

CGI- und Win-CGI-Anwendungen verbrauchen auf dem Server mehr Ressourcen. Erstellen Sie daher komplexe Anwendungen mit vielen Client-Zugriffen als ISAPI- oder NSAPI-DLLs.

Informationen über die Erstellung von Web-Server-Anwendungen finden Sie in Kapitel 29, »Internet-Server-Anwendungen«.

ISAPI- und NSAPI-Anwendungen

Bei dieser Anwendungsart wird das Projekt als DLL eingerichtet. ISAPI- oder NSAPI-Anwendungen sind DLLs, die vom Web-Server geladen werden. Die Informationen werden an die Routinen in der Bibliothek übergeben, dort verarbeitet und durch den Server zurück an den Client gesendet.

CGI-Anwendungen

CGI-Programme sind Konsolenanwendungen, die Client-Anforderungen von der Standardeingabe entgegennehmen, bearbeiten und das Ergebnis über die Standardausgabe zurückgeben.

Win-CGI-Anwendungen

Win-CGI-Programme sind Windows-Anwendungen, die Client-Anforderungen aus einer INI-Datei lesen, bearbeiten und das Ergebnis wieder in eine Datei schreiben.

Verteilte Anwendungen mit COM und DCOM erstellen

COM (Component Object Model) ist eine auf Windows basierende verteilte Objektarchitektur, die mittels vordefinierter Routinen (sogenannten Schnittstellen) eine Interoperabiltität von Objekten gewährleistet. Mit COM erstellte Anwendungen können auf Objekte zugreifen, die in einem anderen Prozeß oder im Falle von DCOM auf einem anderen Computer implementiert sind.

COM und DCOM

Delphi stellt Klassen und Experten bereit, die Sie bei der Erstellung der Hauptkomponenten einer COM-, OLE- oder ActiveX-Anwendung unterstützen. Wenn Sie mit Delphi eine COM-basierte Anwendung erstellen, können Sie beispielsweise durch eine interne Verwendung von Schnittstellen das Software-Design optimieren oder Objekte erstellen, die mit anderen COM-basierten API-Objekten im System interagieren (z. B. mit Win95-Shell-Erweiterungen und DirectX der Multimedia-Unterstützung).

Weitere Informationen über COM und ActiveX finden Sie in Kapitel 44, »COM-Technologien im Überblick«, in Kapitel 48, »ActiveX-Steuerelemente erstellen« sowie unter »Client-Anwendung als ActiveX-Steuerelement weitergeben« auf Seite 14-33.

Weitere Informationen zu DCOM finden Sie im Abschnitt »DCOM-Verbindungen einsetzen« auf Seite 14-9.

MTS

MTS (Microsoft Transaction Server) ist eine Erweiterung von DCOM, die Transaktionsdienste, Sicherungsfunktionen und Ressourcen-Pooling für verteilte COM-Anwendungen bereitstellt.

Informationen zu MTS finden Sie in Kapitel 51, »MTS-Objekte erstellen«, und unter »MTS verwenden« auf Seite 14-6.

Verteilte CORBA-Anwendungen erstellen

CORBA (Common Object Request Broker Architecture) ist eine Methode zur Nutzung verteilter Objekte in Anwendungen. Da dieser Standard auf vielen Plattformen unterstützt wird, können Sie mit Hilfe von CORBA-Anwendungen auch Programme verwenden, die nicht auf einem Windows-Computer ausgeführt werden.

CORBA ist wie COM eine verteilte Objektarchitektur. Dies bedeutet, daß Client-Anwendungen auch Objekte verwenden können, die auf einem Remote-Server implementiert sind.

Weitere Informationen zu CORBA finden Sie in Kapitel 28, »CORBA-Anwendungen«.

Informationen über den Vertrieb von CORBA-Anwendungen finden Sie in »CORBA-Anwendungen weitergeben« auf Seite 28-18.

Verteilte Datenbankanwendungen erstellen

Delphi unterstützt verteilte Datenbankanwendungen auf Basis von MIDAS. Zu dieser leistungsfähigen Technologie gehören mehrere spezialisierte Komponenten, mit denen Sie auf einfache Weise mehrschichtige Datenbankanwendungen erstellen können. Für die Kommunikation können verschiedene Protokolle einschließlich DCOM, CORBA, TCP/IP und OLEnterprise eingesetzt werden.

Informationen über verteilte Datenbankanwendungen finden Sie in Kapitel 14, »Mehrschichtige Anwendungen erstellen«.

Bei verteilten Datenbankanwendungen muß zusätzlich zu den Anwendungsdateien meist die BDE weitergegeben werden. Informationen zu diesem Thema finden Sie in »Datenbankanwendungen weitergeben« auf Seite 11-4.

Kapitel

5

Die Benutzeroberfläche erstellen

Mit Delphi können Sie eine Benutzeroberfläche erstellen, indem Sie Komponenten aus der Komponentenpalette in ein Formular einfügen.

Die Klassen TApplication, TScreen und TForm

Die VCL-Klassen *TApplication*, *TScreen* und *TForm* bilden das Grundgerüst aller Delphi-Anwendungen, denn sie steuern das Verhalten des gesamten Projekts. Das Anwendungsobjekt (*TApplication*) ist das Fundament einer Windows-Anwendung. In seinen Eigenschaften und Methoden sind die Grundfunktionen eines jeden Windows-Standardprogramms gekapselt. Das Bildschirmobjekt (*TScreen*) wird zur Laufzeit verwendet. Es enthält neben Informationen zu den aktuell geladenen Formularen und Datenmodulen auch systemspezifische Informationen wie Bildschirmauflösung und verfügbare Schriften. Formulare (*TForm*) sind die Grundbausteine für die Benutzeroberfläche einer Anwendung. Alle Fenster und Dialogfelder werden von dieser Klasse abgeleitet.

Das Hauptformular

TForm ist die wichtigste Klasse beim Erstellen von Windows-GUI-Anwendungen.

Das erste in einem Projekt erstellte und gespeicherte Formular wird automatisch als Hauptformular der Anwendung verwendet. Es wird beim Start der Anwendung als erstes Formular erstellt und angezeigt. Sobald einem Projekt weitere Formulare hinzugefügt werden, kann auch ein anderes Formular als Hauptformular festgelegt werden. Sie können das Laufzeitverhalten eines beliebigen Formulars am einfachsten testen, indem Sie es zum Hauptformular machen. Dadurch wird es nach dem Starten der Anwendung automatisch als erstes Fenster angezeigt (sofern Sie nicht die Erstellungsreihenfolge ändern) und braucht nicht in einer Ereignisbehandlungsroutine geöffnet zu werden.

So wechseln Sie das Hauptformular eines Projekts:

1 Wählen Sie *Projekt / Optionen*, und klicken Sie auf das Register *Formulare*.

2 Wählen Sie im Kombinationsfeld *Hauptformular* das gewünschte Formular, und bestätigen Sie mit *OK*.

Wenn Sie nun die Anwendung ausführen, wird das neue Hauptformular angezeigt.

Weitere Formulare hinzufügen

Mit *Datei / Neues Formular* können Sie einem Projekt weitere Formulare hinzufügen. Eine Liste aller Formulare und der zugehörigen Units wird im Fenster *Projektverwaltung* (*Ansicht* / *Projektverwaltung*) angezeigt.

Formulare verknüpfen

Wenn Sie einer Anwendung ein Formular hinzufügen, wird eine Referenz auf dieses Formular in die Projektdatei, nicht aber in die anderen Units des Projekts eingefügt. Sie können das Formular also in anderen Formularen erst verwenden, nachdem Sie eine Referenz darauf in die betreffenden Units aufgenommen haben. Diese Referenz bezeichnet man als Formularverknüpfung.

Ein Formular wird häufig mit anderen Formularen verknüpft, um den Zugriff auf seine Komponenten zu ermöglichen. Dies ist vor allem bei Datenbankanwendungen der Fall, in denen die datensensitiven Steuerelemente in den Formularen mit den Datenzugriffskomponenten des Datenmoduls verbunden werden müssen.

So verknüpfen Sie ein Formular mit einem anderen Formular:

1 Aktivieren Sie das Formular, in dem Sie auf das andere Formular zugreifen wollen.

2 Wählen Sie *Datei / Unit verwenden*.

3 Wählen Sie in der Liste die Unit-Datei des Formulars, auf das zugegriffen werden soll.

4 Bestätigen Sie mit *OK*.

Die **uses**-Klausel (im **implementation**-Abschnitt) des aktiven Formulars enthält nun einen Verweis auf das verknüpfte Formular, das dadurch zusammen mit seinen Komponenten für das verknüpfende Formular sichtbar ist.

Zirkuläre Unit-Referenzen vermeiden

Wenn sich zwei Formulare gegenseitig referenzieren, kann beim Compilieren der Anwendung ein Fehler aufgrund einer zirkulären Referenz auftreten. Sie können diese Situation auf eine der folgenden Arten verhindern:

- Plazieren Sie beide uses-Klauseln mit den Unit-Bezeichnern im implementation-Abschnitt der betreffenden Units (der Befehl *Datei / Unit verwenden* führt dies automatisch durch).

- Plazieren Sie die eine **uses**-Klausel im **interface**- und die andere im **implementation**-Abschnitt (Unit-Bezeichner eines anderen Formulars müssen nur äußerst selten in den interface-Abschnitt aufgenommen werden).

Nehmen Sie auf keinen Fall beide **uses**-Klauseln in den **interface**-Abschnitt der betreffenden Unit-Dateien auf. Dies führt beim Compilieren immer zu einem Fehler.

Auf Anwendungsebene arbeiten

In jeder mit Delphi erstellten Windows-Anwendung steht Ihnen die globale Variable *Application* (vom Typ *TApplication*) zur Verfügung. Sie enthält Informationen über die Anwendung und stellt eine Reihe von Funktionen bereit, die im Hintergrund arbeiten. Wenn Sie beispielsweise ein Menü zum Abrufen von Hilfeinformationen erstellen, kümmert sich Application um das interne Aufrufen der Hilfedatei. Umfassende Kenntnisse über die Arbeitsweise der Klasse *TApplication* sind in erster Linie für Komponentenentwickler von Bedeutung. Dennoch sollten Sie bei jedem neuen Projekt die Anwendungseinstellungen in der Registerkarte Anwendung des Dialogfeldes Projektoptionen (*Projekt / Optionen*) festlegen.

Zusätzlich empfängt *Application* viele Ereignisse, die sich auf die Anwendung insgesamt beziehen. Mit dem Ereignis *OnActivate* können Sie beispielsweise Aktionen durchführen, wenn die Anwendung gestartet wird, mit *OnIdle*, wenn die Anwendung nicht ausgelastet ist, mit *OnMessage* Windows-Botschaften abfangen usw. Sie können zwar mit der IDE die Eigenschaften und Methoden der globalen Variable *Application* nicht untersuchen, die Komponente *TApplicationEvents* fängt jedoch die Ereignisse ab und ermöglicht das Bereitstellen von Ereignisbehandlungsroutinen mit Hilfe der IDE.

Mit dem Bildschirm arbeiten

In jedem neuen Projekt wird automatisch eine globale Variable namens *Screen* (vom Typ *TScreen*) erstellt. Sie enthält Informationen über den Status des Bildschirms, auf dem die Anwendung angezeigt wird, und führt verschiedene Aufgaben durch. Sie bestimmt beispielsweise das Aussehen des Mauszeigers, die Größe des Fensters, in dem die Anwendung ausgeführt wird, die für den Bildschirm verfügbaren Schriften und das Verhalten der Anwendung im Mehrschirmbetrieb. Bei Mehrschirmanwendungen verwaltet Screen eine Liste der Bildschirme und ihrer Abmessungen, so daß Sie das Layout der Benutzeroberfläche möglichst effektiv gestalten können.

Das Layout festlegen

Im einfachsten Fall ergibt sich das Layout einer Benutzeroberfläche aus der Plazierung der Steuerelemente in den Formularen. Dabei wird den Eigenschaften. Dabei wird den Eigenschaften *Top, Left, Width* und *Heigh*t des eingefügten Objekts automatisch die Position und die Größe der Komponente zugewiesen. Diese Werte können zur Laufzeit geändert werden.

Steuerelemente verfügen jedoch auch über eine Reihe von Eigenschaften, mit denen sie automatisch an ihrem Inhalt oder ihrem Container ausgerichtet werden. Dadurch können Sie Ihre Formulare so gestalten, daß sie bei jeder Größe einheitlich wirken.

Zwei spezielle Eigenschaften steuern, wie Position und Größe eines Steuerelements relativ zu seinem Container geändert werden. Mit *Align* können Sie festlegen, daß ein Steuerelement entlang einer bestimmten Seite exakt in sein übergeordnetes Objekt eingepaßt wird oder zusammen mit anderen Komponenten den gesamten Client-Bereich des Containers einnimmt. Bei jeder Größenänderung des Containers wird die Größe der betreffenden Steuerelemente automatisch angepaßt, und sie behalten ihre Position am angegebenen Rand bei.

Wenn ein Steuerelement lediglich relativ zu einer bestimmten Seite seines Containers fixiert, aber nicht dessen Seite berühren oder mit diesem vergrößert bzw. verkleinert werden soll, verwenden Sie die Eigenschaft *Anchors*.

Sie können auch sicherstellen, daß ein Steuerelement bei Größenänderungen einen bestimmten Mindest- oder Höchstwert nicht überschreitet. Legen Sie dazu mit der Eigenschaft *Constraints* die maximale und minimale Höhe bzw. die maximale und minimale Breite des Objekts fest. Auf diese Weise können Sie die Höhe und Breite des Steuerelements (in Pixel) einschränken. So kann etwa mit Hilfe der *Constraints*-Untereigenschaften *MinWidth* und *MinHeight* eines Container-Objekts gewährleistet werden, daß die in ihm enthaltenen Objekte bei (fast) jeder Größe zu sehen sind.

Der Wert von *Constraints* wird in der Objekthierarchie weitergegeben. So kann die Größe eines Objekts dadurch eingeschränkt werden, daß es es untergeordnete Objekte mit Größenbeschränkungen enthält. Mit *Constraints* können auch Skalierungsoperationen (*ChangeScale*) begrenzt werden.

In der Klasse *TControl* ist das als **protected** deklarierte Ereignis *OnConstrainedResize* vom Typ *TConstrainedResizeEvent* definiert:

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
MaxHeight: Integer) of object;
```

Mit diesem Ereignis können die Beschränkungen bei einer Änderung der Größe neu zugewiesen werden. Die Grenzwerte werden als **var**-Parameter übergeben und können daher in der Ereignisbehandlungsroutine geändert werden. *OnConstrainedResize* ist bei Container-Objekten (*TForm*, *TScrollBox*, *TControlBar* und *TPanel*) als **published** deklariert. Das Ereignis kann außerdem beim Entwickeln eigener Komponenten in allen von *TControl* abgeleiteten Klassen verwendet oder als **published** deklariert werden.

Wenn sich die Größe ihres Inhalts ändern kann, verfügen Steuerelemente über die Eigenschaft *AutoSize*, mit der die Größe des Steuerelements an seinen Text oder seine untergeordneten Objekte angepaßt werden kann.

Mit Botschaften arbeiten

Eine Botschaft ist eine Benachrichtigung über ein bestimmtes Ereignis, die von Windows an eine Anwendung gesendet wird. Für die Übergabe der Informationen wird ein Record verwendet. Wenn Sie beispielsweise in einem Dialogfeld mit der Maus klicken, sendet Windows die entsprechende Botschaft an das aktive Steuerelement, und die Anwendung reagiert auf das Ereignis. Klicken Sie auf eine Schaltfläche, löst die Botschaft deren Ereignis *OnClick* aus. Beim Klicken in einem Formular kann die Botschaft von der Anwendung ignoriert werden.

Der als Botschaft übergebene Typ heißt *TMsg*. In Windows ist für jede Botschaft eine bestimmte Konstante definiert. Sie wird im Feld *Message* von *TMsg* übergeben. Alle Botschaftskonstanten beginnen mit den Buchstaben wm.

Botschaften werden in der VCL automatisch behandelt, es sei denn, Sie implementieren eine eigene Botschaftsbehandlung. Weitere Informationen über Botschaften und Botschaftsbehandlung finden Sie unter »Das Botschaftsbehandlungssystem« auf Seite 37-1, »Die Behandlung von Botschaften ändern« auf Seite 37-3 und »Neue Routinen zur Botschaftsbehandlung erstellen« auf Seite 37-5.

Weitere Informationen zu Formularen

Jedes in der IDE erstellte Formular wird beim Programmstart automatisch im Speicher erzeugt, indem der entsprechende Code in die Funktion *WinMain()* aufgenommen wird. Dieses Vorgehen eignet sich normalerweise für alle Formulare, und Sie brauchen daran nichts zu ändern. Das gilt vor allem für das Hauptformular, da es während der gesamten Laufzeit der Anwendung im Speicher bleibt.

Nicht alle Formulare müssen sich jedoch während der Ausführungszeit der Anwendung im Speicher befinden. Sie können beispielsweise Dialogfelder auch dynamisch erstellen, wenn sie benötigt werden.

Formulare können modal und nichtmodal sein. Modale Formulare müssen geschlossen werden, bevor die Arbeit in anderen Fenstern fortgesetzt werden kann. Ein Beispiel für ein modales Formular wäre ein Dialogfeld, in dem bestimmte Eingaben vorgenommen werden müssen. Nichtmodale Formulare bleiben auf dem Bildschirm, bis sie geschlossen, minimiert oder von anderen Fenstern verdeckt werden. Der Benutzer kann jederzeit in ein anderes Fenster wechseln.

Die Formularerstellung im Speicher steuern

Standardmäßig wird das Hauptformular einer Anwendung im Speicher erzeugt, indem der folgende Quelltext in die Projektdatei eingefügt wird:

```
Application.CreateForm(TForm1, Form1);
```

Diese Anweisung erstellt eine globale Variable mit demselben Namen wie das Formular. Für jedes Formular einer Anwendung ist also eine entsprechende globale Variable vorhanden. Diese Variable, die ein Zeiger auf eine Instanz der Formularklasse

ist, wird zur Laufzeit für den Zugriff auf das Formular verwendet. In jeder Quelltextdatei, in deren uses-Klausel die Unit des Formulars referenziert wird, kann über diese Variable auf das Formular zugegriffen werden.

Alle auf diese Weise erstellten Formulare werden nach dem Start der Anwendung angezeigt und bleiben während der gesamten Laufzeit im Speicher.

Automatisch erstellte Formulare anzeigen

Wenn ein Formular beim Programmstart automatisch erstellt, in der Anwendung aber erst später geöffnet werden soll, kann es mit Hilfe der Methode *ShowModal* in einer Ereignisbehandlungsroutine des Hauptformulars angezeigt werden:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
   ResultsForm.ShowModal;
end;
```

Da sich das Formular bereits im Speicher befindet, muß keine weitere Instanz erstellt und später wieder freigegeben werden.

Formulare dynamisch erstellen

Es ist nicht immer sinnvoll, alle Formulare einer Anwendung ständig im Speicher zu halten. Sie können den Speicherbedarf einer Anwendung reduzieren, indem Sie bestimmte Formulare nur bei Bedarf erstellen. Ein Dialogfeld wird beispielsweise nur benötigt, während der Benutzer Eingaben vornimmt.

So erstellen Sie in der IDE ein Formular, das erst zur Laufzeit erzeugt wird:

1. Wählen Sie *Datei / Neues Formular*, um ein neues Formular anzuzeigen.
2. Löschen Sie das Formular aus der Liste *Automatische Formularerstellung* in der Registerkarte *Formulare* des Dialogfeldes *Projektoptionen*.

   Dadurch wird das Formular beim Programmstart nicht mehr aufgerufen. Sie können auch manuell die folgende Zeile aus der Projektdatei entfernen:

   ```
   Application.CreateForm(TResultsForm, ResultsForm);
   ```

3. Wenn Sie das Formular nun in der Anwendung benötigen, rufen Sie es mit Hilfe seiner Anzeigemethode auf. Nichtmodale Formulare werden mit *Show*, modale Formulare mit *ShowModal* angezeigt.

In einer Ereignisbehandlungsroutine des Hauptformulars müssen Sie eine Instanz des Formulars erstellen und wieder freigeben. Dazu können Sie wie im folgenden Beispiel die globale Variable verwenden. *ResultsForm* ist modal und wird daher mit der Methode *ShowModal* angezeigt.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm:=TResultForm.Create(self)
  ResultsForm.ShowModal;
  ResultsForm.Free;
end;
```

Das Formular wird nach dem Schließen sofort freigegeben. Es muß daher neu erstellt werden, wenn es an anderer Stelle wieder benötigt wird. Ein mit *Show* angezeigtes, nichtmodales Formular könnte so nicht freigegeben werden, da die Methode beendet wird, während das Formular noch geöffnet ist.

Hinweis Wenn Sie ein Formular mit seinem Konstruktor erstellen, müssen Sie darauf achten, daß es nicht bereits automatisch erstellt wurde. Wenn Sie das Formular nicht aus der Liste *Automatische Formularerstellung* (*Projektoptionen / Formulare*) entfernen, geschieht folgendes: Das Formular wird zuerst beim Programmstart von Delphi erstellt. Danach wird in einer Ereignisbehandlungsroutine eine neue Instanz des Formulars erzeugt, die die erste Instanz überschreibt. Obwohl die automatisch erstellte Instanz immer noch vorhanden ist, kann in der Anwendung nicht mehr darauf zugegriffen werden. Die globale Variable zeigt nach dem Beenden der Ereignisbehandlungsroutine nicht mehr auf ein gültiges Formular, und jeder Zugriffsversuch endet in einem Programmabsturz.

Nichtmodale Formulare erstellen

Sie müssen sicherstellen, daß die Referenzvariablen für nichtmodale Formulare so lange vorhanden sind, wie die Formulare benötigt werden. Die Variablen müssen also global sein. Meistens können Sie die beim Erstellen des Formulars deklarierten globalen Variablen verwenden (die Variablen mit dem Namen des Formulars). Wenn Sie in Ihrer Anwendung weitere Formularinstanzen benötigen, deklarieren Sie für jede Instanz eine eigene globale Variable.

Formularinstanzen mit lokalen Variablen erstellen

Mit Hilfe einer lokalen Variablen kann in einer Ereignisbehandlungsroutine eine eindeutige Instanz eines modalen Formulars erstellt werden. In diesem Fall spielt es keine Rolle, ob das Formular automatisch erstellt wird oder nicht. Sehen Sie sich dazu das folgende Beispiel an:

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

Beachten Sie, daß die globale Instanz des Formulars in dieser Version nicht verwendet wird.

Normalerweise werden in einer Anwendung die globalen Formularinstanzen verwendet. Wenn Sie jedoch eine neue Instanz eines modalen Formulars lediglich in einem eingegrenzten Bereich (z. B. in einer Funktion) benötigen, ist eine lokale Instanz die sicherste und effizienteste Methode.

In Ereignisbehandlungsroutinen für nichtmodale Formulare können natürlich keine lokalen Variablen verwendet werden. Dieser Formulartyp erfordert einen globalen Gültigkeitsbereich. Die Variable muß so lange sichtbar sein, wie das Formular benö-

tigt wird. Da die Methode *Show* sofort nach dem Öffnen des Formulars beendet wird, ist die lokale Variable danach nicht mehr sichtbar.

Zusätzliche Argumente an Formulare übergeben

Normalerweise erstellen Sie die Formulare Ihrer Anwendungen in der IDE. Bei diesem Vorgehen wird dem Konstruktor im einzigen Parameter (*Owner*) der Eigentümer des Formulars (also das aufrufende Anwendungs- oder Formularobjekt) übergeben. *Owner* kann auch den Wert nil haben.

Sie können einem Formular auch weitere Argumente übergeben. Erstellen Sie dazu einen neuen Konstruktor, und führen Sie in ihm die gewünschte Instantiierung des Formulars durch. Im folgenden Beispiel wird dem Konstruktor das zusätzliche Argument *whichButton* übergeben. Der Konstruktor muß manuell in die Klassendeklaration aufgenommen werden.

```
TResultsForm = class(TForm)
   ResultsLabel: TLabel;
   OKButton: TButton;
   procedure OKButtonClick(Sender: TObject);
private
public
   constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

Hier der Quelltext des Konstruktors, dem das zusätzliche Argument *whichButton* übergeben wird. Anhand dieses Wertes wird der Eigenschaft *Caption* einer *Label*-Komponente ein bestimmter String zugewiesen.

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
   case whichButton of
    1: ResultsLabel.Caption := 'Sie haben die erste Schaltfläche gewählt.';
    2: ResultsLabel.Caption := 'Sie haben die zweite Schaltfläche gewählt.';
    3: ResultsLabel.Caption := 'Sie haben die dritte Schaltfläche gewählt.';
   end;
end;
```

Wenn Sie eine Instanz eines Formulars erstellen, das mehrere Konstruktoren hat, verwenden Sie den Konstruktor, der sich für die jeweilige Situation am besten eignet. In der folgenden *OnClick*-Ereignisbehandlungsroutine wird beispielsweise eine Instanz von *TResultsForm* erstellt, die den zweiten Parameter verwendet:

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
   rf: TResultsForm;
begin
   rf := TResultsForm.CreateWithButton(2, self);
   rf.ShowModal;
   rf.Free;
end;
```

Daten aus Formularen abrufen

Die meisten Anwendungen bestehen aus mehreren Formularen, die sich oft gegenseitig Informationen übergeben müssen. Die Übergabe kann durch einen Parameter des Formularkonstruktors oder durch Wertzuweisungen an die Eigenschaften eines Formulars erreicht werden. Welche dieser beiden Möglichkeiten verwendet wird, hängt davon ab, ob es sich um ein modales oder um ein nichtmodales Formular handelt.

Daten aus nichtmodalen Formularen abrufen

Aus nichtmodalen Formularen können Informationen sehr einfach durch Aufrufen der public-Elementfunktionen oder durch Lesen der Formulareigenschaften abgerufen werden. Angenommen, eine Anwendung enthält das nichtmodale Formular *ColorForm* mit dem Listenfeld *ColorListBox*, das eine Liste verschiedener Farben anzeigt (Rot, Grün, Blau usw.). Sobald der Benutzer in der Liste einen neuen Eintrag auswählt, wird der betreffende Farbname automatisch in der Eigenschaft *CurrentColor* gespeichert. Die entsprechende Klassendeklaration lautet folgendermaßen:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

In der *OnClick*-Ereignisbehandlungsroutine der Liste (*ColorListBoxClick*) wird der Eigenschaft *CurrentColor* die aktuelle Farbe zugewiesen, wenn der Benutzer einen neuen Listeneintrag wählt. *CurrentColor* verwendet intern die Schreibfunktion *SetColor*, um den Wert im privaten Datenelement *FColor* zu speichern:

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

Nun sei angenommen, daß in einem anderen Formular der Anwendung namens *ResultsForm* die aktuelle Farbe in *ColorForm* ermittelt werden muß, wenn auf eine Schaltfläche (*UpdateButton*) geklickt wird. Die *OnClick*-Ereignisbehandlungsroutine dieser Schaltfläche sieht folgendermaßen aus:

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
```

```
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {Beliebige Aktion mit dem String MainColor ausführen}
  end;
end;
```

Zuerst wird mit der Funktion *Assigned* geprüft, ob das Formular *ColorForm* vorhanden ist. Danach wird der Wert der Eigenschaft *CurrentColor* von *ColorForm* gelesen.

Alternativ dazu kann die aktuelle Farbe auch ohne die Eigenschaft *CurrentColor* ermittelt werden, wenn in *ColorForm* die public-Funktion *GetColor* definiert ist (z. B. `MainColor := ColorForm.GetColor;`). Die ausgewählte Farbe kann in einem anderen Formular natürlich auch durch direktes Abfragen des Listenfeldes abgerufen werden:

```
with ColorForm.ColorListBox do
   MainColor := Items[ItemIndex];
```

Die Verwendung einer definierten Schnittstelle in Form einer Eigenschaft ermöglicht jedoch einen einfacheren Zugriff auf *ColorForm*. Andere Formulare brauchen dann nur den Wert von *CurrentColor* abzurufen.

Daten aus modalen Formularen abrufen

Modale Formulare enthalten wie nichtmodale Formulare oft Informationen, die in anderen Fenstern benötigt werden. In der einfachsten Form ruft Formular A das modale Formular B auf. Nach dem Schließen von Formular B muß Formular A wissen, welche Aktionen der Benutzer in Formular B durchgeführt hat, da das weitere Vorgehen davon abhängt. Befindet sich Formular B noch im Speicher, können wie beim nichtmodalen Formular im letzten Beispiel die Eigenschaften und Elementfunktionen verwendet werden. Wie können aber Werte abgerufen werden, wenn Formular B beim Schließen aus dem Speicher entfernt wird? Da ein Formular keinen expliziten Rückgabewert hat, müssen die benötigten Informationen vor dem Freigeben irgendwo gespeichert werden.

Sehen Sie sich dazu die folgende Version von *ColorForm* an. Das Formular ist diesmal modal und wird folgendermaßen deklariert:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

Das Formular enthält ein Listenfeld (*ColorListBox*) mit einer Auswahl verschiedener Farben. Beim Klicken auf die Schaltfläche *SelectButton* wird der aktuelle Farbwert gespeichert und das Formular geschlossen. Mit *CancelButton* wird das Formular geschlossen, ohne den Wert zu speichern.

Beachten Sie, daß ein zweiter Konstruktor vorhanden ist, dem ein Pointer-Argument übergeben wird. Dieser Zeiger verweist auf einen String, der dem aufrufenden Formular bekannt ist. Der Konstruktor ist folgendermaßen implementiert:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
   FColor := Value;
   String(FColor^) := '';
end;
```

Der Konstruktor speichert den Zeiger im **private**-Datenelement *FColor* und initialisiert den String mit einer leeren Zeichenfolge.

Hinweis Dieser Konstruktor kann nur verwendet werden, wenn das Formular explizit erstellt und nicht beim Programmstart automatisch instantiiert wird. Informationen hierzu finden Sie im Abschnitt »Die Formularerstellung im Speicher steuern« auf Seite 5-5.

In der Anwendung wählt der Benutzer eine Farbe aus der Liste und klickt auf *SelectButton*, um die Auswahl zu speichern und das Formular zu schließen. Der Quelltext der *OnClick*-Ereignisbehandlungsroutine sieht folgendermaßen aus:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

Beachten Sie, daß die Ereignisbehandlungsroutine den Namen der gewählten Farbe in dem String speichert, auf den der an den Konstruktor übergebene Zeiger verweist.

Damit *ColorForm* richtig verwendet werden kann, muß das aufrufende Formular dem Konstruktor einen Zeiger auf einen vorhandenen String übergeben. Nehmen wir an, *ColorForm* wird vom Formular *ResultsForm* als Reaktion auf einen Klick auf die Schaltfläche *UpdateButton* instantiiert. Die Ereignisbehandlungsroutine könnte dann folgendermaßen aussehen:

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
   MainColor: String;
begin
   GetColor(Addr(MainColor));
   if MainColor <> '' then
    {Operationen mit dem String MainColor durchführen.}
   else
    {Andere Aktionen durchführen, da keine Farbe gewählt wurde.}
end;

procedure GetColor(PColor: Pointer);
begin
   ColorForm := TColorForm.CreateWithColor(PColor, Self);
   ColorForm.ShowModal;
   ColorForm.Free;
end;
```

UpdateButtonClick erstellt zuerst den String *MainColor*. Die Adresse dieses Strings wird an die Funktion *GetColor* übergeben, in der das Formular *ColorForm* erstellt und der Zeiger als Argument an den Konstruktor übergeben wird. Obwohl *ColorForm* nach dem Schließen sofort wieder freigegeben wird, ist der gewählte Farbname noch in *MainColor* gespeichert, wenn im Formular eine Farbe ausgewählt wurde. Andernfalls enthält *MainColor* einen leeren String, woran eindeutig zu erkennen ist, daß der Benutzer *ColorForm* geschlossen hat, ohne eine Farbe auszuwählen.

In diesem Beispiel werden die Informationen aus dem modalen Formular in einer String-Variablen abgelegt. Natürlich können bei Bedarf auch komplexere Datenstrukturen oder Objekte verwendet werden. Sie sollten aber immer eine Möglichkeit vorsehen, dem aufrufenden Formular mitzuteilen, daß das modale Formular ohne eine Änderung oder Auswahl durch den Benutzer geschlossen wurde (indem Sie beispielsweise *MainColor* einen leeren String als Standardwert zuweisen).

Komponenten und Komponentengruppen wiederverwenden

Delphi bietet verschiedene Möglichkeiten zum Speichern und Wiederverwenden der bereits mit VCL-Komponenten durchgeführten Arbeiten:

- Komponentenvorlagen bieten eine einfache und schnelle Möglichkeit zum Konfigurieren und Speichern von Komponentengruppen. Weitere Informationen finden Sie unter »Komponentenvorlagen« auf Seite 5-12.
- Sie können Formulare, Datenmodule und Projekte in der Objektablage speichern. Die Objektablage stellt eine zentrale Datenbank mit wiederverwendbaren Elementen dar und ermöglicht die Nutzung der Formularvererbung, um Änderungen weiterzugeben. Weitere Informationen finden Sie unter »Die Objektablage verwenden« auf Seite 2-40.
- Sie können in der Komponentenpalette oder in der Objektablage Frames speichern. Frames verwenden die Formularvererbung und können in Formulare und andere Frames eingebettet werden. Informationen hierzu finden Sie unter »Frames« auf Seite 5-13.
- Das Erstellen einer benutzerdefinierten Komponente stellt die komlizierteste Möglichkeit zur Wiederverwendung von Quelltext dar, bietet aber die größtmögliche Flexibilität. Weitere Informationen finden Sie in Kapitel 31, »Die Komponentenentwicklung im Überblick«.

Komponentenvorlagen

Sie können Vorlagen erstellen, die aus einer oder mehreren Komponenten bestehen. Nachdem Sie die Komponenten auf einem Formular angeordnet, deren Eigenschaften eingestellt und den erforderlichen Quelltext geschrieben haben, speichern Sie das Ergebnis als Komponentenvorlage. Später können Sie diese Vorlage in der Komponentenpalette auswählen und die vorkonfigurierten Komponenten in ein Formular einfügen. Alle zugehörigen Eigenschaften und der Quelltext der Ereignisbehandlungsroutinen werden gleichzeitig in das aktuelle Projekt eingefügt.

Nachdem Sie eine Vorlage in ein Formular eingefügt haben, können Sie die Komponenten neu anordnen, die Eigenschaften zurücksetzen sowie Ereignisbehandlungsroutinen erstellen bzw. bearbeiten, wie dies auch beim Einfügen einzelner Komponenten möglich ist.

So erstellen Sie eine Komponentenvorlage:

1. Plazieren Sie die Komponenten in einem Formular, und ordnen Sie diese wie gewünscht an. Stellen Sie die Eigenschaften und Ereignisse im Objektinspektor nach Bedarf ein.
2. Wählen Sie alle Komponenten aus. Die einfachste Möglichkeit besteht im Ziehen mit der Maus über alle Komponenten. Graue Griffe werden an den Ecken jedes ausgewählten Objekts angezeigt.
3. Wählen Sie *Komponente / Komponentenvorlage* erzeugen.
4. Geben Sie einen Namen für die Komponentenvorlage in das Eingabefeld Komponentenname ein. Der vorgeschlagene Name besteht aus dem Typ der ersten in Schritt 2 ausgewählten Komponente, gefolgt von dem Wort »Vorlage«. Wenn Sie beispielsweise eine Beschriftung und dann ein Eingabefeld auswählen, lautet der vorgeschlagene Name »TLabelTemplate«. Sie können diesen Namen ändern, müssen aber darauf achten, nicht den Namen einer existierenden Komponente zu verwenden.
5. Geben Sie in das Eingabefeld *Palettenseite* die Registerkarte der Komponentenpalette ein, auf der die Vorlage verfügbar sein soll. Wenn Sie eine nicht vorhandene Registerkarte angeben, wird beim Speichern der Vorlage eine neue Registerkarte erstellt.
6. Wählen Sie unter *Palettensymbol* ein Bitmap zur Repräsentation der Vorlage in der Palette. Standardmäßig wird das Bitmap des Komponententyps der in Schritt 2 zuerst ausgewählten Komponente vorgeschlagen. Klicken Sie auf *Ändern*, um ein anderes Bitmap auszuwählen. Das ausgewählte Bitmap darf maximal eine Größe von 24 x 24 Pixel aufweisen.
7. Klicken Sie auf *OK*.

Mit *Komponente / Palette konfigurieren* können Sie Vorlagen aus der Komponentenpalette entfernen.

Frames

Ein Frame (*TFrame*) ist wie ein Formular ein Container für andere Komponenten. Für die automatische Instantiierung und Freigabe der in ihm enthaltenen Komponenten wird derselbe Eigentümermechanismus wie bei Formularen verwendet. Außerdem gelten dieselben Beziehungen zwischen unter- und übergeordneten Objekten bei der Synchronisierung von Komponenteneigenschaften. In anderer Hinsicht ähnelt ein Frame mehr einer benutzerdefinierten Komponente als einem Formular. Frames können in der Komponentenpalette gespeichert sowie in Formularen, anderen Frames oder anderen Containerobjekten verschachtelt werden. Nachdem ein Frame erstellt und gespeichert wurde, kann er als Einheit bearbeitet werden. Er kann Änderungen

von den in ihm enthaltenen Komponenten (einschließlich anderer Frames) erben. Wenn Sie einen Frame in einen anderen Frame oder ein Formular einbetten, erbt er in der Folgezeit Änderungen, die an dem Frame vorgenommen werden, aus dem er abgeleitet wurde.

Frames erstellen

Sie können einen neuen Frame erstellen, indem Sie *Datei / Neuer Frame* wählen. Alternativ können Sie *Datei / Neu* wählen und doppelt auf *Frame* klicken. Anschließend können Sie Komponenten (einschließlich anderer Frames) in den Frame einfügen.

Normalerweise bietet es sich an, Frames als Bestandteil eines Projekts zu speichern. Dies ist jedoch nicht notwendig. Wenn Sie ein Projekt erstellen wollen, das nur Frames und keine Formulare enthält, wählen Sie *Datei / Neue Anwendung*, schließen das neue Formular und die Unit, ohne zu speichern, und wählen anschließend *Datei / Neuer Frame*. Speichern Sie dann das Projekt.

Hinweis Beim Speichern von Frames sollten Sie nicht die Standardnamen Unit1, Project1 usw. verwenden, da dies bei späterer Verwendung der Frames wahrscheinlich zu Konflikten führt.

Zur Entwurfszeit können Sie einen Frame im aktuellen Projekt anzeigen, indem Sie *Ansicht* / *Formulare* und dann einen Frame wählen. Wie bei Formularen und Datenmodulen können Sie zwischen dem Formular-Designer und der DFM-Datei des Frames umschalten, indem Sie mit der rechten Maustaste auf den Frame klicken und dann *Ansicht als Formular* oder *Ansicht als Text* wählen.

Frames in die Komponentenpalette einfügen

Frames werden als Komponentenvorlage in die Komponentenpalette eingefügt. Sie können einen Frame in die Komponentenpalette einfügen, indem Sie ihn im Formular-Designer öffnen (zu diesem Zweck können Sie keinen in eine andere Komponente eingebetteten Frame verwenden), mit der rechten Maustaste auf ihn klicken und dann *Zu Palette hinzufügen* wählen. Geben Sie im Dialogfeld *Informationen* über die Vorlage einen Namen, eine Registerkarte in der Palette und ein Symbol für die neue Vorlage an.

Frames verwenden und ändern

Sie müssen einen Frame direkt oder indirekt in ein Formular einfügen, um es in einer Anwendung benutzen zu können. Sie können Frames direkt in Formulare, andere Frames oder andere Containerobjekte einfügen (beispielsweise Tafeln oder Bildlauffelder).

Der Formular-Designer bietet zwei Möglichkeiten, einen Frame in eine Anwendung einzufügen:

- Wählen Sie in der Komponentenpalette einen Frame aus, und fügen Sie ihn in ein Formular, einen anderen Frame oder ein anderes Containerobjekt ein. Sofern dies

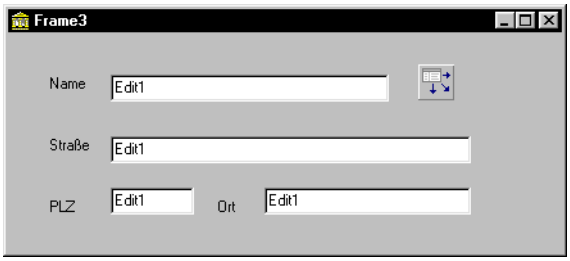
erforderlich ist, fordert der Formular-Designer eine Bestätigung zum Einfügen der Unit-Datei in das aktuelle Projekt an.

- Wählen in der Registerkarte *Standard* der Komponentenpalette *Frame*, und klicken Sie auf ein Formular oder einen anderen Frame. Ein Dialogfeld mit einer Liste der bereits im Projekt enthaltenen Frames wird geöffnet. Wählen Sie einen Frame aus, und klicken Sie auf *OK*.

Wenn Sie einen Frame oder ein Formular in einen anderen Container einfügen, deklariert Delphi eine neue Klasse, die von dem ausgewählten Frame abgeleitet ist. (Auf ähnliche Weise deklariert Delphi beim Einfügen eines Formulars in ein Projekt eine neue Klasse, die aus *TForm* abgeleitet ist.) Alle später am Original (Vorfahr) vorgenommen Änderungen werden also an den eingebetteten Frame weitergegeben, während sich Änderungen am eingebetteten Frame nicht auf die Vorfahrklasse auswirken.

Ein Beispiel: Sie wollen eine Gruppe von Datenzugriffskomponenten und datensensitiven Steuerelementen für den wiederholten Einsatz - gegebenenfalls auch in unterschiedlichen Anwendungen - zusammenstellen. Eine Möglichkeit besteht im Zusammenfassen der Komponenten in einer Komponentenvorlage. Wenn Sie jedoch die Vorlage bereits in mehreren Projekten eingesetzt haben und später die Anordnung der Steuerelemente ändern wollen, müssen Sie die Änderungen in jedem dieser Projekte vornehmen. Fügen Sie die Datenbankkomponenten dagegen in einen Frame ein, müssen die Änderungen nur an einer Stelle vorgenommen werden. Änderungen des ursprünglichen Frames werden automatisch an die eingebetteten Nachfahren weitergegeben, wenn Sie die Projekte neu compilieren. Gleichzeitig können Sie jeden eingebetteten Frame ändern, ohne daß such diese Änderungen auf den ursprünglichen Frame oder die anderen eingebetteten Frames auswirkt.

Abbildung 5.1 Ein Frame mit datensensitiven Steuerelementen und einer Datenquellenkomponente



Außer der Vereinfachung der Wartung ermöglichen Frames auch eine effizientere Nutzung der Ressourcen. Sie können beispielsweise ein Bitmap oder eine andere Grafik in einer Anwendung einsetzen, indem Sie die Grafik in die Eigenschaft Picture eines *TImage*-Steuerelements laden. Wenn Sie dieselbe Grafik mehrfach in einer Anwendung einsetzen, wird mit jedem in ein Formular eingefügten Bildobjekt eine Kopie der Grafik in die Ressourcendatei des Formulars eingefügt. (Dies gilt auch, wenn Sie T*Image.Picture* einmal zuweisen und das Image-Steuerelement als Komponentenvorlage einfügen.) Eine bessere Lösung besteht darin, das Image-Objekt in einen Frame einzufügen und diesen dann an jeder Stelle zu verwenden, an dem die Grafik angezeigt werden soll. Die Formulardateien sind bei diesem Vorgehen kleiner. Außer-

dem können Sie die Grafik an jeder Stelle einfach ändern, indem Sie das *Image*-Objekt im ursprünglichen Frame bearbeiten.

Frames freigeben

Sie können einen Frame auf zwei Arten für andere Entwickler freigeben:

- Fügen Sie den Frame in die Objektablage ein.
- Geben Sie die Unit- (PAS) und die Formulardatei (DFM) des Frames weiter.

Sie können einen Frame in die Objektablage einfügen, indem Sie ein Projekt mit diesem Frame öffnen, im Formular-Designer mit der rechten Maustaste klicken und *Der Objektablage hinzufügen* wählen. Weitere Informationen finden Sie unter »Die Objektablage verwenden« auf Seite 2-40.

Wenn Sie anderen Entwicklern die Unit- und die Formulardatei eines Frames senden, können diese die Dateien öffnen und den Frame in die Komponentenpalette einfügen. Sind in den Frame andere Frames eingebettet, müssen diese als Teil des Projekts geöffnet werden.

Menüs erstellen und verwalten

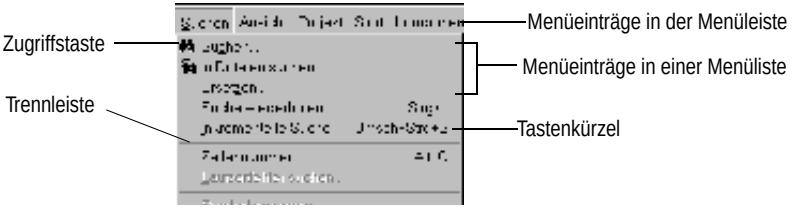
Mit Hilfe von Menüs können Befehle, die logisch zusammengehören, gruppiert und auf einfache Weise ausgeführt werden. Menüs (vordefinierte oder eigene) werden mit dem Menü-Designer erstellt. Sie plazieren einfach eine Menükomponente in einem Formular, öffnen den Menü-Designer und geben die gewünschten Optionen direkt in das Entwurfsfenster ein. Menüeinträge können in beliebiger Weise hinzugefügt, entfernt oder neu angeordnet werden.

Sie brauchen die Anwendung nicht einmal zu starten, um das Ergebnis zu sehen. Das Menü wird im Formular so dargestellt, wie es während der Ausführung angezeigt wird. Menüs können auch zur Laufzeit geändert werden, um dem Benutzer zusätzliche Informationen oder Optionen bereitzustellen.

In diesem Kapitel erfahren Sie, wie mit Hilfe des Menü-Designers Menüleisten (Hauptmenüs) und Popup-Menüs (lokale Menüs) erstellt werden. Sie finden hier ausführliche Informationen zu folgenden Themen:

- Den Menü-Designer öffnen
- Menüs entwerfen
- Menüeinträge im Objektinspektor bearbeiten
- Das lokale Menü des Menü-Designers verwenden
- Menüvorlagen verwenden
- Ein Menü als Vorlage speichern
- Grafiken zu Menüeinträgen hinzufügen

Abbildung 5.2 Menüterminonologie

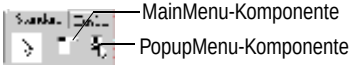


Informationen zum Verknüpfen von Menüeinträgen mit dem bei ihrer Auswahl auszuführenden Quelltext finden Sie unter »Menüereignissen eine Behandlungsroutine zuordnen« auf Seite 2-30.

Den Menü-Designer öffnen

Bevor Sie den Menü-Designer verwenden können, müssen Sie ein *MainMenu*- oder *PopupMenu*-Objekt in das Formular einfügen. Sie finden diese Menükomponenten in der Registerkarte *Standard* der Komponentenpalette.

Abbildung 5.3 MainMenu- und PopupMenu-Komponente



Mit einer *MainMenu*-Komponente erstellen Sie ein Hauptmenü in der Titelleiste des Formulars. Mit *PopupMenu*-Komponenten erstellen Sie lokale Menüs, die beim Klikken mit der rechten Maustaste angezeigt werden. Dieser Menütyp verfügt über keine Menüleiste.

Um den Menü-Designer zu öffnen, wählen Sie eine Menükomponente im Formular aus und führen anschließend eine der folgenden Aktionen durch:

- Doppelklicken Sie auf die Menükomponente.
- Wählen Sie in der Registerkarte *Eigenschaften* des Objektinspektors die Eigenschaft *Items* aus. Doppelklicken Sie anschließend auf den Eintrag in der Wertespalte (*Menu*), oder klicken Sie auf die Ellipsen-Schaltfläche (...).

  Dadurch wird der Menü-Designer geöffnet. In ihm ist der erste (leere) Menüeintrag und im Objektinspektor die Eigenschaft *Caption* ausgewählt.

Abbildung 5.4 Der Menü-Designer bei einem Hauptmenü

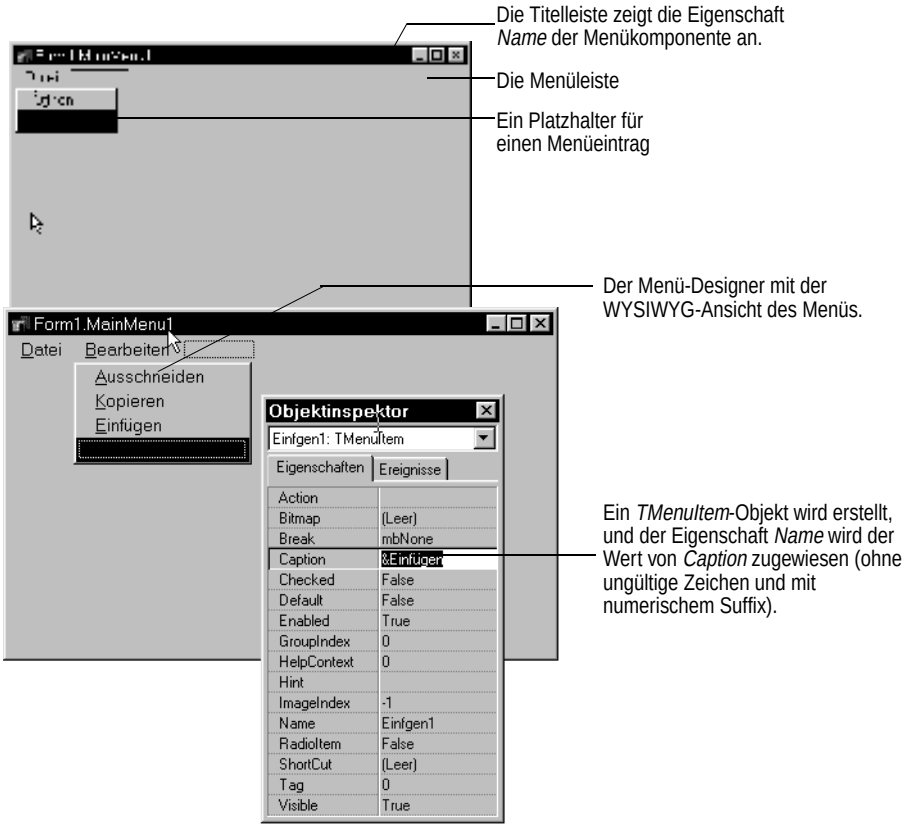


Abbildung 5.5 Der Menü-Designer bei einem Popup-Menü

Menüs entwerfen

Für jedes Menü Ihrer Anwendung müssen Sie dem Formular eine Menükomponente hinzufügen. Anschließend können Sie die Struktur des Menüs von Grund auf neu erstellen oder eine der vordefinierten Menüvorlagen verwenden.

In diesem Abschnitt erfahren Sie, wie mit dem Menü-Designer Menüs entworfen werden. Informationen zu den vordefinierten Menüvorlagen finden Sie im Abschnitt »Menüvorlagen verwenden« auf Seite 5-26.

Menüs benennen

Eine Menükomponente erhält wie alle anderen Komponenten automatisch einen Standardnamen (z. B. *MainMenu1*), wenn sie einem Formular hinzugefügt wird. Es empfiehlt sich jedoch, ihr einen aussagekräftigeren Namen zu geben (wobei Sie natürlich die Namenskonventionen für gültige Object-Pascal-Bezeichner beachten müssen).

Der Menüname wird automatisch in die Typdeklaration des Formulars und in die Komponentenliste aufgenommen.

Menüeinträge benennen

Im Gegensatz zu Menükomponenten müssen Sie neue Menüeinträge explizit benennen. Dazu haben Sie folgende Möglichkeiten:

- Weisen Sie den Wert der Eigenschaft *Name des Eintrags* zu.
- Weisen Sie den Wert zuerst der Eigenschaft *Caption* zu, und lassen Sie Delphi den entsprechenden Namen eintragen.

  Wenn Sie Caption beispielsweise den Wert Datei zuweisen, erhält die Eigenschaft *Name* automatisch den Wert *Datei1*. Wenn Sie *Name* vor *Caption* einen Wert zuweisen, bleibt die Eigenschaft *Caption* leer, und Sie müssen selbst einen Wert eingeben.

Hinweis Wenn Sie der Eigenschaft *Caption* Zeichen zuweisen, die für Object-Pascal-Bezeichner nicht zulässig sind, ändert sich der Wert von *Name* entsprechend. Geben Sie beispielsweise als Titel öffnen ein, erhält der Eintrag den Namen öffnen1.

Die folgende Tabelle zeigt einige Beispiele. Alle Einträge befinden sich in derselben Menüleiste.

Tabelle 5.1 Beispieltitel und die daraus abgeleiteten Namen

Titel	Name	Beschreibung
&Datei	Datei1	Das Zeichen & wird entfernt.
&Datei (zweites Vorkommen)	Datei2	Doppelte Einträge werden durchnumeriert.
1234	N12341	Am Anfang wird ein Buchstabe, am Ende eine Nummer hinzugefügt.
1234 (zweites Vorkommen)	N12342	Doppelte Einträge werden durchnumeriert.
$@@@#	N1	Alle Sonderzeichen werden entfernt. Am Anfang wird ein Buchstabe, am Ende eine Nummer hinzugefügt.
- (Gedankenstrich)	N2	Trennleisten erhalten einen Buchstaben und werden durchnumeriert.

Die Namen der Menüeinträge werden anschließend automatisch in die Typdeklaration des Formulars und in die Komponentenliste aufgenommen.

Menüeinträge hinzufügen, einfügen und entfernen

Die folgenden Arbeitsschritte zeigen Ihnen die wichtigsten Aufgaben beim Erstellen einer Menüstruktur. Dabei wird vorausgesetzt, daß der Menü-Designer geöffnet ist.

So fügen Sie einen Menüeintrag hinzu:

1. Wählen Sie die Position im Menü aus, an der Sie den Eintrag erstellen wollen.

   Wenn Sie den Menü-Designer eben erst geöffnet haben, ist bereits die erste Position in der Menüleiste ausgewählt.

2. Weisen Sie der Eigenschaft *Caption* einen Titel zu. Wenn Sie zuerst einen Wert für Name eingeben, müssen Sie anschließend *Caption* auswählen und einen Wert eingeben.

3. Drücken Sie *Return*.

   Dadurch wird automatisch der nächste Platzhalter für einen neuen Menüeintrag ausgewählt.

   Wenn Sie zuerst einen Wert für *Caption* eingegeben haben, wechseln Sie mit der Taste Auf noch einmal zu dem soeben erstellten Menüeintrag. Wie Sie sehen, enthält die Eigenschaft Name einen von *Caption* abgeleiteten Wert (siehe »Menüeinträge benennen« auf Seite 5-19).

4. Jetzt können Sie noch weitere Menüeinträge erstellen, indem Sie den Eigenschaften *Name* und *Caption* Werte zuweisen oder mit *Esc* zur Menüleiste zurückkehren.

   Mit Hilfe der Pfeiltasten gelangen Sie von der Menüleiste in das Menü und von dort zu den verschiedenen Einträgen. Geben Sie einen Wert ein, und schließen Sie die Aktion mit *Return* ab. Um in die Menüleiste zurückzukehren, drücken Sie *Esc*.

So fügen Sie einen neuen, leeren Menüeintrag ein:

1. So fügen Sie einen neuen, leeren Menüeintrag ein:

2. Drücken Sie die Taste *Einfg*.

   In der Menüleiste werden neue Einträge links neben dem ausgewählten Eintrag, in der Menüliste direkt darüber eingefügt.

So entfernen Sie einen Menüeintrag:

1. Wählen Sie im Menü den Eintrag aus, den Sie entfernen wollen.

2. Drücken Sie *Entf*.

Hinweis Der Standardplatzhalter links neben (Menüliste) oder unter (Menüleiste) dem zuletzt erstellten Eintrag kann nicht entfernt werden. Er wird aber zur Laufzeit nicht angezeigt.

Trennleisten hinzufügen

Mit Hilfe einer Trennleiste fügen Sie zwischen zwei Menüeinträgen eine Linie ein. Auf diese Weise können Sie zusammengehörige Optionen gruppieren und eine Menüliste übersichtlicher gestalten.

Wenn Sie einen Menüeintrag als Trennleiste verwenden wollen, geben Sie für die Eigenschaft *Caption* einen Gedankenstrich (-) ein.

Zugriffstasten und Tastenkürzel festlegen

Zugriffstasten ermöglichen die Ausführung eines Menübefehls mit Hilfe der Tastatur. Der Benutzer braucht nur die Taste *Alt* und den mit dem Zeichen & kombinierten Buchstaben zu drücken. Im Menü wird der betreffende Buchstabe unterstrichen angezeigt.

Delphi sucht automatisch nach doppelt belegten Zugriffstasten und korrigiert diese zur Laufzeit . Dies stellt sicher, daß dynamisch erzeugte Menüs zur Laufzeit keine doppelt belegten Zugriffstasten enthalten und außerdem für jeden Menübefehl eine Zugriffstaste verfügbar ist. Sie können diese automatische Prüfung deaktivieren, indem Sie der Eigenschaft *AutoHotkeys* eines Menübefehls den Wert *maManual* zuweisen.

So legen Sie eine Zugriffstaste fest:

- Fügen Sie vor dem gewünschten Buchstaben das Zeichen & (ein kaufmännisches Und) ein.

Um beispielsweise mit dem Buchstaben *S* auf den Menübefehl *Speichern* zuzugreifen, geben Sie `&Speichern` ein.

Mit Hilfe von Tastenkürzeln können Aktionen ohne Zugriff auf das Menü ausgeführt werden. Der Benutzer braucht nur die jeweilige Tastenkombination zu drücken.

So legen Sie ein Tastenkürzel fest:

- Weisen Sie der Eigenschaft *ShortCut* eine Tastenkombination zu, oder wählen Sie das gewünschte Kürzel aus der Dropdown-Liste.

  In der Liste sind nicht alle möglichen Kombinationen aufgeführt.

Das Tastenkürzel wird neben dem Menüeintrag angezeigt.

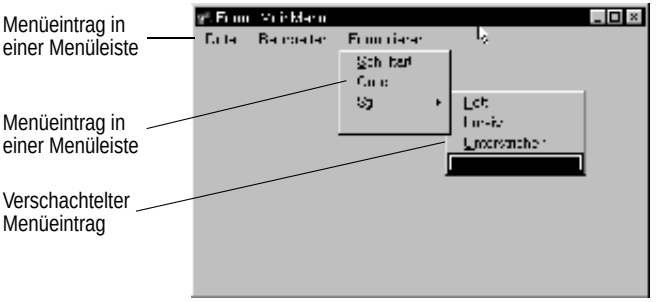
Achtung Tastenkürzel werden im Unterschied zu den Zugriffstasten nicht automatisch auf doppelte Belegung überprüft. Sie müssen die Eindeutigkeit selbst gewährleisten.

Untermenüs erstellen

Viele Menüeinträge bieten Dropdown-Listen an, in denen weitere Optionen zur Verfügung stehen. Diese Listen sind an einem Pfeil rechts neben dem Eintrag zu erkennen. Sie können in Ihren Anwendungen beliebig viele Menüebenen verwenden.

Durch Untermenüs kann der vertikale Platzbedarf eines Menüs verringert werden. Mehr als zwei oder drei Menüebenen sind jedoch aus Gründen der Ergonomie nicht zu empfehlen (bei Popup-Menüs sollten Sie maximal ein Untermenü verwenden).

Abbildung 5.6 Verschachtelte Menüstrukturen



So erstellen Sie ein Untermenü:

1 Wählen Sie den Menüeintrag aus, für den Sie ein Untermenü erstellen wollen.

2 Drücken Sie *Strg*, um den ersten Platzhalter zu erstellen, oder klicken Sie mit der rechten Maustaste und wählen Sie Untermenü erstellen.

3 Geben Sie einen Namen für den neuen Untermenüeintrag ein, oder ziehen Sie eine vorhandene Menüoption auf den Platzhalter.

4 Drücken Sie *Return*, oder ↓, um den nächsten Platzhalter zu erstellen.

5 Wiederholen Sie die Schritte 3 und 4 für alle weiteren Einträge des Untermenüs.

6 Drücken Sie *Esc*, um zur übergeordneten Menüebene zurückzukehren.

Untermenüs aus vorhandenen Menüs erstellen

Sie können ein Untermenü auch dadurch erstellen, daß Sie einen vorhandenen Menüleisteneintrag (oder eine Menüvorlage) zwischen die Einträge der Menüliste einfügen. Beim Verschieben eines Menüs in eine vorhandene Menüstruktur werden sämtliche Einträge verlagert und bilden ein vollständiges Untermenü. Auch Untermenüs können das Ziel einer solchen Einfügung sein. In diesem Fall wird einfach eine weitere Menüebene erzeugt.

Menüeinträge verschieben

Im Menü-Designer können die einzelnen Menüeinträge mit der Maus beliebig verschoben werden. Sie können einen Eintrag innerhalb der Menüleiste, innerhalb der Menüliste oder in ein völlig anderes Menü verschieben.

Bei diesen Operationen gelten natürlich hierachische Einschränkungen. Ein Menüleisteneintrag kann nicht in seine eigene Liste und ein Menüeintrag nicht in sein eigenes Untermenü verschoben werden. Sie können aber jeden Eintrag unabhängig von seiner ursprünglichen Position in ein anderes Menü verschieben.

Beim Ziehen eines Menüeintrags ist an der Form des Mauszeigers zu erkennen, ob der Eintrag an der neuen Position abgelegt werden kann. Ein Menüleisteneintrag wird immer zusammen mit seiner Eintragsliste verschoben.

So verschieben Sie einen Menüeintrag in der Menüleiste:

1. Ziehen Sie den Menüeintrag entlang der Menüleiste, bis der Mauszeiger auf die gewünschte Position zeigt.
2. Lassen Sie die Maustaste los, um den Eintrag abzulegen.

So verschieben Sie einen Menüleisteneintrag in eine Menüliste:

1. Ziehen Sie den Menüeintrag entlang der Menüleiste, bis der Mauszeiger auf das gewünschte Menü zeigt.

   Das betreffende Menü wird automatisch geöffnet, und Sie können den Eintrag an seine neue Position ziehen.
2. Lassen Sie die Maustaste los, um den Eintrag an seiner neuen Position abzulegen.

Grafiken zu Menüeinträgen hinzufügen

Mit Hilfe von Grafiken können Sie den Benutzern das Auswählen der Befehle erleichtern. Weisen Sie den Menüeinträgen einfach Bitmaps zu, die der jeweiligen Aktion entsprechen (wie bei einer Symbolleiste). Folgendermaßen fügen Sie einem Menüeintrag eine Grafik hinzu:

1. Plazieren Sie eine *TMainMenu*- oder *TPopupMenu*-Komponente im Formular.
2. Fügen Sie eine *TImageList*-Komponente hinzu.
3. Doppelklicken Sie auf das *TImageLis*t-Objekt, um den Bilderlisten-Editor zu öffnen.
4. Klicken Sie auf *Hinzufügen*, und laden Sie das oder die gewünschten Bitmaps in den Editor. Bestätigen Sie anschließend mit *OK*.
5. Weisen Sie der Eigenschaft *Images* des Menüobjekts die soeben erstellte *ImageList*-Komponente zu.
6. Erstellen Sie die Menü- und Untermenüeinträge, wie es in den letzten Abschnitten beschrieben wurde.
7. Wählen Sie den Eintrag aus, dem eine Grafik zugeordnet werden soll, und weisen Sie seiner Eigenschaft *ImageIndex* den Index der entsprechenden Grafik in der Bilderliste zu (beim Standardwert von *ImageIndex* (-1) wird keine Grafik angezeigt).

Hinweis Grafiken werden im Menü nur richtig angezeigt, wenn Sie 16 x 16 Pixel groß sind. Andere Größen führen zu Ausrichtungs- und Konsistenzproblemen.

Das Menü anzeigen

Sie können Ihr Menü während des Entwurfs jederzeit im Formular anzeigen, ohne die Anwendung auszuführen. Bei Popup-Menüs sind zur Entwurfszeit nur die Komponenten selbst zu sehen, nicht die erstellten Menüs. Verwenden Sie daher zur Anzeige dieses Menütyps den Menü-Designer.

So zeigen Sie ein Menü an:

1. Klicken Sie in dem Formular, dessen Menü angezeigt werden soll (ist das Formular nicht zu sehen, öffnen Sie es mit Hilfe des Menüs *Ansicht*).

2 Wenn das Formular mehrere Menüs enthält, wählen Sie im Objektinspektor das gewünschte Menü aus der Dropdown-Liste der Eigenschaft *Menu des Formulars*.

Das Menü wird im Formular so angezeigt wie zur Laufzeit der Anwendung.

Menüeinträge im Objektinspektor bearbeiten

In diesem Abschnitt haben Sie bisher erfahren, wie Sie verschiedene Eigenschaften von Menüeinträgen (z. B. *Name* und *Caption*) mit Hilfe des Menü-Designers festlegen können.

Außerdem wurde beschrieben, wie bestimmte Eigenschaften (z. B. *ShortCut*) direkt im Objektinspektor zugewiesen werden können.

Wenn Sie einen Menüeintrag mit dem Menü-Designer bearbeiten, werden seine Eigenschaften weiterhin im Objektinspektor angezeigt. Sie können jederzeit in den Objektinspektor wechseln und das Bearbeiten der Eigenschaften dort fortsetzen. Sie können einen Menüeintrag aber auch in der Komponentenliste des Objektinspektors auswählen und seine Eigenschaften bearbeiten, ohne den Menü-Designer zu öffnen.

So schließen Sie den Menü-Designer und setzen das Bearbeiten der Menüeinträge fort:

1 Wechseln Sie vom Menü-Designer in den Objektinspektor, indem Sie auf das Register *Eigenschaften* klicken.

2 Schließen Sie den Menü-Designer.

Der Objektinspektor ist weiterhin aktiv, und Sie können das Bearbeiten der Eigenschaften des augewählten Eintrags fortsetzen. Um einen anderen Eintrag zu bearbeiten, brauchen Sie ihn nur in der Komponentenliste auszuwählen.

Die Zuordnung einer Ereignisbehandlungsroutine zu einem Menü wird unter »Menüereignissen eine Behandlungsroutine zuordnen« auf Seite 2-30 beschrieben.

Das lokale Menü des Menü-Designers verwenden

Mit Hilfe des lokalen Menüs können Sie bequem auf die wichtigsten Befehle und Vorlagen des Menü-Designers zugreifen (weitere Informationen zu Menüvorlagen finden Sie im Abschnitt »Menüvorlagen verwenden« auf Seite 5-26).

Um das lokale Menü anzuzeigen, klicken Sie im Menü-Designer mit der rechten Maustaste oder drücken *Alt+F10*.

Die Befehle des lokalen Menüs

Die folgende Tabelle enthält die Befehle des lokalen Menüs des Menü-Designers.

Tabelle 5.2 Befehle des lokalen Menüs des Menü-Designers

Menübefehl	Aktion
Einfügen	Ein Platzhalter für einen neuen Eintrag wird über oder links von der aktuellen Position eingefügt.
Löschen	Der ausgewählte Menüeintrag wird gelöscht (und seine Untereinträge, falls vorhanden).
Untermenü erstellen	Ein Platzhalter für eine neue Untermenüebene wird erstellt und ein Pfeil rechts neben dem ausgewählten Eintrag angezeigt.
Menü auswählen	Eine Liste mit den Menüs des aktuellen Formulars wird angezeigt. Durch Doppelklicken auf ein Listenelement wird das Designer-Fenster des betreffenden Menüs geöffnet.
Als Vorlage speichern	Das Dialogfeld *Vorlage speichern* wird geöffnet, in dem Sie ein Menü für die spätere Wiederverwendung speichern können.
Aus Vorlage einfügen	Das Dialogfeld *Vorlagen einfügen* wird geöffnet, in dem Sie die gewünschte Menüvorlage auswählen können.
Vorlage löschen	Das Dialogfeld *Vorlagen löschen* wird geöffnet, in dem Sie die zu löschende Vorlage auswählen können.
Aus Ressource einfügen	Das Dialogfeld *Menü aus Ressource einfügen* wird geöffnet, in dem Sie die gewünschte Menüdatei (MNU) auswählen können.

Zur Entwurfszeit zwischen Menüs wechseln

Wenn Sie für ein Formular mehrere Menüs erstellen, können Sie mit dem lokalen Menü des Menü-Designers oder mit dem Objektinspektor problemlos zwischen ihnen wechseln.

So wechseln Sie mit Hilfe des lokalen Menüs zwischen den Menüs in einem Formular:

1 Klicken Sie im Menü-Designer mit der rechten Maustaste, und wählen Sie *Menü auswählen.*

Das Dialogfeld *Menü auswählen* wird geöffnet.

Abbildung 5.7 Dialogfeld *Menü auswählen*

Hier sind alle Menüs des aktuellen Formulars aufgeführt.

2 Wählen Sie das Menü aus, das Sie anzeigen oder bearbeiten wollen.

So wechseln Sie mit Hilfe des Objektinspektors zwischen den Menüs eines Formulars:

1 Aktivieren Sie das Formular, dessen Menüs Sie bearbeiten wollen.

2 Wählen Sie das gewünschte Menü aus der Komponentenliste am oberen Rand des Objektinspektors.

3 Wählen Sie in der Registerkarte *Eigenschaften* die Eigenschaft *Items* des Menüs aus, und doppelklicken Sie auf den Wert (*Menu*), oder klicken Sie auf die Ellipsen-Schaltfläche.

Menüvorlagen verwenden

Delphi stellt Ihnen mehrere vordefinierte Menüs (Menüvorlagen) mit häufig verwendeten Befehlen zur Verfügung, die Sie unverändert in Ihre Anwendungen übernehmen oder als Ausgangspunkt für eigene Menüs verwenden können. In Menüvorlagen sind jedoch keine Ereignisbehandlungsroutinen implementiert.

Die mitgelieferten Menüvorlagen werden bei der Standardinstallation in das Unterverzeichnis BIN kopiert und sind an der Namenserweiterung DMT (Delphi Menu Template) zu erkennen.

Sie können jedes im Menü-Designer erstellte Menü als Vorlage speichern und wie die vordefinierten Vorlagen beliebig wiederverwenden. Wenn Sie eine bestimmte Menüvorlage nicht mehr benötigen, brauchen Sie sie nur aus der Liste zu entfernen.

So fügen Sie einer Anwendung eine Menüvorlage hinzu:

1 Klicken Sie im Menü-Designer mit der rechten Maustaste, und wählen Sie *Aus Vorlage einfügen*.

(Wenn keine Vorlagen definiert sind, ist die Option *Aus Vorlage einfügen* deaktiviert.)

Das Dialogfeld *Vorlagen einfügen* wird geöffnet. Es enthält eine Liste der verfügbaren Menüvorlagen.

Abbildung 5.8 Das Dialogfeld *Schablone einfügen*

2 Wählen Sie die gewünschte Menüvorlage aus, und drücken Sie *Return* (oder klikken Sie auf *OK*).

Die vordefinierte Vorlage wird an der aktuellen Position in das Menü eingefügt. Ist beispielsweise ein Eintrag in einer Menüliste ausgewählt, wird die Vorlage unmittelbar darüber plaziert. Wenn ein Eintrag in einer Menüleiste ausgewählt ist, wird die Vorlage links davon eingefügt.

So löschen Sie eine Menüvorlage:

1 Klicken Sie im Menü-Designer mit der rechten Maustaste, und wählen Sie *Vorlagen löschen*.

(Wenn keine Vorlagen definiert sind, ist die Option *Vorlagen löschen* deaktiviert.)

Das Dialogfeld *Vorlagen löschen* wird geöffnet. Es enthält eine Liste der verfügbaren Menüvorlagen.

2 Wählen Sie die Vorlage aus, die Sie löschen wollen, und drücken Sie *Entf*.

Die Vorlage wird aus der Liste entfernt und von der Festplatte gelöscht.

Ein Menü als Vorlage speichern

Sie können jedes von Ihnen erstellte Menü als Vorlage speichern und in anderen Formularen wiederverwenden. Menüvorlagen tragen zu einem konsistenten Erscheinungsbild Ihrer Anwendungen bei.

Alle Menüvorlagen werden im Verzeichnis BIN in Dateien mit der Namenserweiterung DMT gespeichert.

So speichern Sie ein Menü als Vorlage:

1 Erstellen Sie das Menü, das Sie später wiederverwenden wollen.

Sie können beliebig viele Einträge, Befehle und Untermenüs verwenden. Der gesamte Inhalt des aktiven Fensters des Menü-Designers wird in einer einzigen Vorlage gespeichert.

2 Klicken Sie im Menü-Designer mit der rechten Maustaste, und wählen Sie *Als Vorlage speichern*.

Das Dialogfeld *Vorlage speichern* wird geöffnet.

Abbildung 5.9 Das Dialogfeld *Schablone speichern*

3 Geben Sie eine kurze Beschreibung in das Eingabefeld *Beschreibung* der Vorlage ein, und bestätigen Sie mit *OK*.

Das Dialogfeld *Vorlage speichern* wird geschlossen, und Sie befinden sich wieder im Menü-Designer.

Hinweis Die hier eingegebene Beschreibung wird nur in den Dialogfeldern *Vorlage speichern*, *Vorlagen einfügen* und *Vorlagen löschen* angezeigt. Sie steht in keinerlei Beziehung zu den Eigenschaften *Name* und *Caption* des Menüs.

Namenskonventionen für Menüeinträge und Ereignisbehandlungsroutinen in Vorlagen

Wenn Sie ein Menü als Vorlage speichern, wird seine Eigenschaft *Name* nicht berücksichtigt, da jedes Menü im Gültigkeitsbereich seines Eigentümers (des Formulars) einen eindeutigen Namen haben muß. Sobald Sie die Vorlage jedoch einfügen, werden für das Menü und seine Einträge automatisch Namen vergeben.

Angenommen, Sie speichern das Menü *Datei* als Vorlage, und seine Eigenschaft *Name* hat den Wert *MeinDateiMenue*. Wenn Sie es nun in ein neues Formular einfügen, erhält es automatisch den Namen *Datei1*. Ist bereits ein Menüeintrag mit diesem Namen vorhanden, heißt das neue Menü *Datei2*.

In einer Vorlage werden natürlich auch keine *OnClick*-Ereignisbehandlungsroutinen gespeichert, da kein Test möglich ist, ob der Quelltext im neuen Formular verwendet werden kann.

Sie können aber den Einträgen der Menüvorlage vorhandene Behandlungsroutinen zuweisen. Weitere Informationen hierzu finden Sie im Abschnitt »Ereignissen eine vorhandene Behandlungsroutine zuordnen« auf Seite 2-29.

Menüeinträge zur Laufzeit bearbeiten

In manchen Situationen ist es sinnvoll, dem Benutzer zur Laufzeit zusätzliche Informationen oder Befehle anzubieten. Zu diesem Zweck können Sie mit der Methode *Add* oder *Insert* einen neuen Eintrag in die bestehende Menüstruktur einfügen oder die vorhandenen Menüeinträge mit Hilfe der Eigenschaft *Visible* ein- oder ausblenden. Mit der Eigenschaft *Enabled* können Einträge deaktiviert (grau angezeigt) werden.

Beispiele zu den Eigenschaften *Visible* und *Enabled* finden Sie im Abschnitt »Menüeinträge deaktivieren« auf Seite 6-11.

In MDI- und OLE-Anwendungen können Menüeinträge auch mit vorhandenen Menüleisten kombiniert werden. Im nächsten Abschnitt erfahren Sie mehr über dieses Thema.

Menüs kombinieren

Bei MDI-Anwendungen (wie der Texteditor-Beispielanwendung) und OLE-Client-Anwendungen muß das Hauptmenü der Anwendung in der Lage sein, Menüeinträge anderer Formulare oder des OLE-Servers aufzunehmen. Man bezeichnet dies als *kombiniertes Menü*.

Um Ihre Menüs auf diesen Vorgang vorzubereiten, brauchen Sie nur Werte für die folgenden beiden Eigenschaften anzugeben:

- *Menu* (Eigenschaft des Formulars)
- *GroupIndex* (Eigenschaft der Menüeinträge)

Das aktive Menü festlegen: die Eigenschaft Menu

Die Eigenschaft *Menu* gibt das aktive Menü des Formulars an. Nur das aktive Menü kann mit einem anderen Menü kombiniert werden. Bei Formularen mit mehreren Menüs können Sie das aktive Menü zur Laufzeit wie im folgenden Beispiel ändern:

```
Form1.Menu := SecondMenu;
```

Die Reihenfolge der kombinierten Menüeinträge festlegen: die Eigenschaft GroupIndex

Die Eigenschaft *GroupIndex* bestimmt, an welcher Position die kombinierten Menüeinträge in die gemeinsame Menüleiste aufgenommen werden. Sie können anstelle der vorhandenen Menüleisteneinträge oder zusätzlich zu diesen angezeigt werden.

Der Standardwert von *GroupIndex* ist 0. Beachten Sie bei dieser Eigenschaft folgende Regeln:

- Kleinere Zahlen erscheinen im Menü zuerst (also weiter links).

  Setzen Sie *GroupIndex* bei einem Menü, das immer ganz links angezeigt werden soll (z. B. Datei), auf 0. Geben Sie einen hohen Wert für ein Menü an, das rechts au-

ßen erscheinen soll (z. B. Hilfe). Die Indexe müssen nicht unmittelbar aufeinanderfolgen.

- Um Einträge im Hauptmenü zu ersetzen, brauchen Sie nur Einträgen im untergeordneten Menü denselben *GroupIndex*-Wert zuzuweisen.

  Dieses Vorgehen ist bei Gruppen und einzelnen Einträgen möglich. Gibt es beispielsweise im Hauptformular ein Menü *Bearbeiten* mit dem *GroupIndex*-Wert 1, können Sie es durch einen oder mehrere Einträge aus dem Menü des untergeordneten Formulars ersetzen, indem Sie deren Eigenschaft *GroupIndex* ebenfalls auf 1 setzen.

  Wenn Sie mehreren Einträgen im untergeordneten Menü denselben *GroupIndex*-Wert zuweisen, bleibt ihre Reihenfolge beim Einfügen in das Hauptmenü erhalten.

- Um Einträge in das Hauptmenü einzufügen, ohne vorhandene Einträge zu ersetzen, vergeben Sie den Index nicht fortlaufend. Die Lücken im Index bieten Platz für die Menüs des untergeordneten Formulars.

  Wenn Sie den Einträgen des Hauptmenüs beispielsweise 0 und 5 als Index zuweisen, können Sie die Einträge mit den Nummern 1, 2, 3 und 4 des untergeordneten Menüs einfügen.

Ressourcen-Dateien importieren

Sie können in Delphi auch Menüs verwenden, die mit anderen Anwendungen erstellt wurden. Voraussetzung dafür ist, daß diese Menüs im Standarddateiformat für Windows-Ressourcen (RC) gespeichert wurden. Die Menüs können direkt in ein Delphi-Projekt importiert werden.

So laden Sie eine Ressourcen-Datei mit einem Menü:

1. Wählen Sie im Menü-Designer die Position aus, an der das Menü eingefügt werden soll.

   Das importierte Menü kann einem vorhandenen Menü hinzugefügt werden oder selbst ein eigenständiges Menü bilden.

2. Klicken Sie im Menü-Designer mit der rechten Maustaste, und wählen Sie *Aus Ressource einfügen*.

   Das Dialogfeld *Menü aus Ressource einfügen* wird geöffnet.

3. Wählen Sie im Dialogfeld die gewünschte Ressourcen-Datei aus, und bestätigen Sie mit *OK*.

   Das Menü wird aus der Datei in das Designer-Fenster importiert und angezeigt.

Hinweis Wenn die Ressourcen-Datei mehrere Menüs enthält, müssen Sie vor dem Importieren jedes Menü in einer eigenen Datei speichern.

ToolBar- und CoolBar-Komponenten erstellen

Symbolleisten befinden sich normalerweise am oberen Rand eines Formulars (direkt unterhalb des Hauptmenüs) und enthalten Schaltflächen oder andere Steuerelemente. Eine CoolBar-Komponente (im Englischen auch »Rebar« genannt) ist eine Art Symbolleiste, in der die Steuerelemente auf mehrere Bereiche verteilt sind, deren Position und Größe zur Laufzeit geändert werden kann. Sind mehrere Symbolleisten am oberen Formularrand ausgerichtet, werden sie in der Reihenfolge ihrer Plazierung vertikal übereinander angeordnet.

In eine Symbolleiste können nicht nur Schaltflächen, sondern beliebige Arten von Komponenten eingefügt werden, also auch Bildlaufleisten, Beschriftungen usw.

Es gibt verschiedene Möglichkeiten, eine Symbolleiste hinzuzufügen:

- Plazieren Sie eine Panel-Komponente (*TPanel*) im Formular, und fügen Sie ihr Steuerelemente (normalerweise *TSpeedButton*-Objekte) hinzu.
- Verwenden Sie anstelle von *TPanel* ein ToolBar-Objekt (*TToolBar*), und fügen Sie ihm die gewünschten Steuerelemente hinzu. Diese Komponente verwaltet Schaltflächen und andere Steuerelemente, ordnet sie in Reihen an und korrigiert automatisch ihre Größe und Position. Wenn Sie als Schaltflächen *TToolButton*-Objekte verwenden, können Sie diese nach Funktionen gruppieren und verschiedene Anzeigeoptionen festlegen.
- Verwenden Sie eine CoolBar-Komponente, (*TCoolBar*), und fügen Sie ihr die gewünschten Steuerelemente hinzu. Bei CoolBar-Komponenten sind die Steuerelemente auf Bereiche verteilt, deren Größe und Position unabhängig voneinander geändert werden kann.

Auf welche Weise Sie Symbolleisten implementieren, hängt von Ihrer Anwendung ab. Der Vorteil einer Panel-Komponente besteht darin, daß Sie hier die vollständige Kontrolle über das Aussehen und die Funktionsweise der Symbolleiste haben.

ToolBar- und CoolBar-Komponenten sind native Windows-Steuerelemente und bieten als solche den Vorzug, daß Ihre Anwendungen immer wie ein professionelles Windows-Standardprogramm aussehen. Wenn diese Steuerelemente in künftigen Versionen geändert werden, spiegelt sich das auch in Ihren Programmen wider. Sie können jedoch nur verwendet werden, wenn die Bibliothek COMCTL32.DLL im System installiert ist. *TToolBar* und *TCoolBar* werden in WinNT 3.51 nicht unterstützt.

In den nächsten Abschnitten finden Sie Informationen zu folgenden Themen:

- Eine Panel-Komponente als Symbolleiste hinzufügen
- Eine ToolBar-Komponente als Symbolleiste hinzufügen
- Eine CoolBar-Komponente hinzufügen
- Auf Mausklicks reagieren
- Verborgene Symbolleisten hinzufügen
- Symbolleisten ein- und ausblenden

Eine Panel-Komponente als Symbolleiste hinzufügen

So fügen Sie einem Formular eine Panel-Komponente als Symbolleiste hinzu:

1. Plazieren Sie eine Panel-Komponente im Formular (aus der Registerkarte *Standard* der Komponentenpalette).
2. Setzen Sie die Eigenschaft *Align* der Komponente auf *alTop*. Bei dieser Ausrichtung behält die Tafel ihre Höhe bei, wird jedoch automatisch an die Breite des Client-Bereichs des Formulars angepaßt (auch bei Größenänderungen).
3. Fügen Sie SpeedButton-Komponenten (Registerkarte *Zusätzlich*) oder andere Steuerelemente hinzu.

SpeedButton-Objekte sind spezielle Schaltflächen für Symbolleisten, die mit Panel-Komponenten erstellt werden. Sie enthalten normalerweise keine Beschriftung, sondern nur eine kleine Grafik (*Glyph*), die ihre Funktion andeutet.

SpeedButton-Objekte können folgendermaßen verwendet werden:

- Als normale Schaltflächen
- Als Ein-/Ausschalter
- Als Optionsfeldgruppe

In den folgenden Abschnitten finden Sie Informationen darüber, wie Sie in einer Symbolleiste SpeedButton-Objekte implementieren können:

- Einer Symbolleiste (*TPanel*) eine SpeedButton-Komponente hinzufügen
- Einer SpeedButton-Komponente eine Grafik zuweisen
- Die Anfangseinstellungen einer SpeedButton-Komponente festlegen
- Eine Gruppe von SpeedButton-Komponenten erstellen
- Eine SpeedButton-Komponente als Ein-/Ausschalter verwenden

Einer Symbolleiste (TPanel) eine SpeedButton-Komponente hinzufügen

Um einer Symbolleiste ein SpeedButton-Objekt hinzuzufügen, brauchen Sie es nur in der Registerkarte *Zusätzlich* der Komponentenpalette auszuwählen und in der Panel-Komponente zu plazieren.

Das übergeordnete Objekt der Schaltfläche ist die Tafel (nicht das Formular). Die Schaltfläche wird daher immer zusammen mit der Tafel verschoben oder ausgeblendet.

Panel-Komponenten sind standardmäßig 41, SpeedButton-Objekte 25 Pixel hoch. Sie können die Schaltfläche in der Symbolleiste vertikal zentrieren, indem Sie ihrer Eigenschaft Top den Wert 8 zuweisen. Wenn die Standardeinstellung des Rasters nicht geändert wurde, rastet die Schaltfläche beim Verschieben mit der Maus automatisch an dieser Position ein.

Einer SpeedButton-Komponente eine Grafik zuweisen

Jede SpeedButton-Komponente enthält eine kleine Grafik (Eigenschaft *Glyph*), an der ihre Funktion zu erkennen ist. Wenn Sie einer Schaltfläche nur eine Grafik zuweisen, ändert sich diese automatisch entsprechend dem Schalterzustand (gedrückt, nicht gedrückt, ausgewählt oder deaktiviert). Sie können aber jedem Status auch eine eigene Grafik zuordnen.

Die Grafiken können während des Entwurfs im Objektinspektor oder zur Laufzeit im Programm zugewiesen werden.

So weisen Sie im Objektinspektor einem SpeedButton-Objekt eine Grafik zu:

1. Wählen Sie die gewünschte Schaltfläche aus.
2. Wählen Sie im Objektinspektor die Eigenschaft *Glyph* aus.
3. Doppelklicken Sie in der Wertespalte der Eigenschaft, um den Grafikeditor zu öffnen, und laden Sie die gewünschte Grafikdatei.

Die Anfangseinstellungen einer SpeedButton-Komponente festlegen

Der aktuelle Status und die Funktion einer SpeedButton-Komponente sind an ihrem Aussehen zu erkennen. Da die Komponente keine Beschriftung enthält, müssen Sie dem Benutzer einen aussagekräftigen visuellen Hinweis geben.

Die folgende Tabelle enthält Informationen zu den Eigenschaften, mit denen Sie das Erscheinungsbild von SpeedButton-Komponenten beeinflussen können:

Tabelle 5.3 Anzeigeeigenschaften für SpeedButton-Objekte

Anzeige	Aktion
Gedrückt	Setzen Sie *GroupIndex* auf einen Wert ungleich Null und *Down* auf *True*.
Deaktiviert	Setzen Sie *Enabled* auf *False*.
Linker Rand	Setzen Sie *Indent* auf einen Wert größer Null.

Wenn es in Ihrer Anwendung ein bestimmtes Standardwerkzeug gibt (z. B. das Auswahlwerkzeug in einer Grafikanwendung), müssen Sie sicherstellen, daß die entsprechende Schaltfläche beim Programmstart bereits gedrückt ist. Weisen Sie dazu der Eigenschaft *GroupIndex* der betreffenden Schaltfläche einen Wert ungleich Null und der Eigenschaft *Down* den Wert *True* zu.

Eine Gruppe von SpeedButton-Komponenten erstellen

SpeedButton-Objekte werden häufig für Auswahlmöglichkeiten verwendet, die sich gegenseitig ausschließen. Dazu müssen die betreffenden Komponenten derselben Gruppe zugeordnet werden. Wenn der Benutzer auf eine Schaltfläche dieser Gruppe klickt, werden die anderen Schaltflächen automatisch in den nicht gedrückten Zustand versetzt.

Sie können eine beliebige Anzahl von SpeedButton-Komponenten zu einer Gruppe zusammenfasssen, indem Sie der Eigenschaft *GroupIndex* dieser Schaltflächen denselben Wert zuweisen.

Sie brauchen diese Aktion nicht mit jedem einzelnen Objekt durchzuführen. Markieren Sie einfach alle Schaltflächen, und weisen Sie *GroupIndex* einen eindeutigen Wert zu.

Eine SpeedButton-Komponente als Ein-/Ausschalter verwenden

In manchen Situationen sollen gruppierte Schaltflächen durch Anklicken im gedrückten Zustand wieder in den nicht gedrückten Zustand zurückkehren. Daraus können sich Gruppen ergeben, in denen keine einzige Schaltfläche gedrückt ist. Die entsprechenden Steuerelemente heißen *Ein-/Ausschalter* und werden mit Hilfe ihrer Eigenschaft *AllowAllUp* definiert. Beim ersten Klicken wird die Schaltfläche in den gedrückten, beim zweiten Klicken wieder in den nicht gedrückten Zustand versetzt.

Um einen Ein-/Ausschalter zu erstellen, setzen Sie die Eigenschaft *AllowAllUp* einer gruppierten Schaltfläche auf *True*.

Den anderen Schaltflächen der Gruppe wird dadurch automatisch derselbe Wert zugewiesen. Die Gruppe kann als normale Schaltergruppe verwendet werden, bei der jeweils nur eine Schaltfläche gedrückt ist, sie erlaubt aber auch, daß sich alle Schaltflächen im nicht gedrückten Zustand befinden.

Eine ToolBar-Komponente als Symbolleiste hinzufügen

Eine ToolBar-Komponente (*TToolBar*) verfügtim Gegensatz zu einer *Panel*-Komponente über eine interne Schalterverwaltung und spezielle Anzeigeoptionen. So können Sie einem Formular eine ToolBar-Komponente als Symbolleiste hinzufügen:

1. Plazieren Sie eine ToolBar-Komponente (Registerkarte *Win32*) im Formular. Sie wird nach dem Einfügen automatisch am oberen Rand ausgerichtet.
2. Fügen Sie ToolButton-Objekte oder andere Steuerelemente in die Symbolleiste ein.

ToolButton-Objekte sind speziell für die Verwendung in Symbolleisten vorgesehen. Sie können wie SpeedButton-Objekte folgendermaßen verwendet werden:

- Als normale Schaltflächen
- Als Ein-/Ausschalter
- Als Optionsfeldgruppe

In den folgenden Abschnitten finden Sie Informationen, wie Sie ToolButton-Objekte in einer Symbolleiste implementieren können:

- Einer Symbolleiste (*TToolBar*) eine ToolButton-Komponente hinzufügen
- Einer ToolButton-Komponente eine Grafik zuweisen
- Die Anfangseinstellungen einer ToolButton-Komponente festlegen
- Eine Gruppe von ToolButton-Komponenten erstellen
- Eine ToolButton-Komponente als Ein-/Ausschalter verwenden

Einer Symbolleiste (TToolBar) eine ToolButton-Komponente hinzufügen

Um einer ToolBar-Komponente ein ToolButton-Objekt hinzuzufügen, klicken Sie in der ToolBar-Komponente mit der rechten Maustaste und wählen Neuer Schalter.

Da die Symbolleiste den Schaltflächen übergeordnet ist, werden diese zusammen mit ihr ausgeblendet und verschoben. Außerdem haben alle Schaltflächen dieselbe Höhe und Breite. Wenn Sie andere Steuerelemente in die Symbolleiste einfügen, erhalten auch diese eine einheitliche Höhe. Steuerelemente, die horizontal nicht in die Symbolleiste passen, werden automatisch in einer neuen Reihe plaziert.

Einer ToolButton-Komponente eine Grafik zuweisen

Bei einem ToolButton-Objekt legt die Eigenschaft *ImageIndex* fest, welche Grafik zur Laufzeit angezeigt wird. Wenn Sie nur eine Grafik angeben, ändert sich diese automatisch, sobald die Schaltfläche deaktiviert wird. Gehen Sie folgendermaßen vor, um einem ToolButton-Objekt eine Grafik zuzuweisen:

1. Wählen Sie die Symbolleiste aus, in der sich die ToolButton-Objekte befinden.
2. Weisen Sie der Eigenschaft *Images* der Symbolleiste eine Bilderliste (*TImageList*) zu. Bilderlisten enthalten Symbole oder Bitmaps mit identischer Größe.
3. Wählen Sie eine ToolButton-Komponente aus.
4. Weisen Sie der Eigenschaft *ImageIndex* der Schaltfläche den Index der gewünschten Grafik in der Bilderliste zu.

Eine Schaltfläche kann eine andere Grafik anzeigen, wenn sie deaktiviert ist oder sich unter dem Mauszeiger befindet. Weisen Sie dazu den Eigenschaften *DisabledImages* und *HotImages* der Symbolleiste jeweils eine Bilderliste zu.

Die Anfangseinstellungen einer ToolButton-Komponente festlegen

Die folgende Tabelle enthält Informationen zu den Eigenschaften, die das Erscheinungsbild von ToolButton-Komponenten beeinflussen.

Tabelle 5.4 Anzeigeeigenschaften für ToolButton-Objekte

Anzeige	Aktion
Gedrückt	Setzen Sie *GroupIndex* auf einen Wert ungleich Null und *Down* auf *True*.
Deaktiviert	Setzen Sie *Enabled* auf *False*.
Linker Rand	Setzen Sie *Indent* auf einen Wert größer Null.
Popup-Rahmen (die Symbolleiste wirkt dadurch transparent)	Setzen Sie *Flat* auf *True*.

Hinweis Die Eigenschaft *Flat* wirkt sich nur dann auf *TToolBar* aus, wenn COMCTL32.DLL in der Version 4.70 oder später im System installiert ist.

Um nach einem bestimmten ToolBar-Objekt eine neue Schalterreihe zu beginnen, wählen Sie die betreffende Komponente aus und setzen ihre Eigenschaft *Wrap* auf *True*.

Sie können das automatische Umbrechen der Schaltflächen deaktivieren, indem Sie die Eigenschaft *Wrapable* der Symbolleiste auf *False* setzen.

Eine Gruppe von ToolButton-Komponenten erstellen

Sie können mehrere ToolButton-Komponenten zu einer Gruppe zusammenfassen. Wählen Sie dazu die betreffenden Objekt aus, und setzen Sie ihre Eigenschaft *Style* auf *tbsCheck* und ihre Eigenschaft *Grouped* auf *True*. Wählt der Benutzer nun eine Schaltfläche der Gruppe aus, werden die anderen automatisch in den nicht gedrückten Zustand versetzt. Diese Funktionsweise eignet sich für Auswahlmöglichkeiten, die sich gegenseitig ausschließen.

Alle benachbarten ToolButton-Objekte, bei denen Style auf *tbsCheck* und *Grouped* auf *True* gesetzt ist, bilden eine Gruppe. Sie können die Gruppierung aufheben, indem Sie an der gewünschten Position zwischen den Schaltflächen eines der folgenden Elemente einfügen:

- Eine ToolButton-Komponente, bei der *Grouped* den Wert *False* hat.
- Eine ToolButton-Komponente, bei der *Style* nicht auf *tbsCheck* gesetzt ist. Um Zwischenräume oder Trennlinien zu erstellen, weisen Sie *Style* den Wert *tbsSeparator* bzw. *tbsDivider* zu.
- Ein anderes Steuerelement als ein ToolButton-Objekt.

Eine ToolButton-Komponente als Ein-/Ausschalter verwenden

Mit Hilfe der Eigenschaft *AllowAllUp* können Sie ToolButton-Objekte erstellen, die als Ein-/Ausschalter dienen. Beim ersten Klicken wird die Schaltfläche in den gedrückten, beim zweiten Klicken wieder in den nicht gedrückten Zustand versetzt. Setzen Sie einfach die Eigenschaft *AllowAllUp* der betreffenden Komponente auf *True*.

Wie bei SpeedButton-Objekten wird die Eigenschaft automatisch auch bei allen anderen Schaltflächen der Gruppe auf *True* gesetzt.

Eine CoolBar-Komponente hinzufügen

Die Komponente *TCoolBar* (auch *ReBar* genannt) zeigt fensterorientierte Steuerelemente in getrennten Bereichen an, deren Größe und Position unabhängig voneinander geändert werden kann. Die Bereiche können zur Laufzeit mit Hilfe der Griffmarkierungen am linken Rand vergrößert und verkleinert werden.

So fügen Sie einem Formular eine CoolBar-Komponente hinzu:

1. Plazieren Sie eine CoolBar-Komponente (Registerkarte *Win32*) im Formular. Die Komponente wird nach dem Einfügen automatisch am oberen Rand ausgerichtet.
2. Fügen Sie beliebige fensterorientierte Steuerelement in die Komponente ein.

Fensterorientierte Steuerelemente sind Komponenten, die von *TWinControl* abgeleitet sind. Sie können auch grafische Steuerelemente (z. B. *TLabel* oder *TSpeedButton*) einfügen, jedoch werden diese nicht in eigenen Bereichen angezeigt.

Hinweis CoolBar-Komponenten können nur verwendet werden, wenn die Bibliothek COMCTL.DLL in der Version 4.70 oder später im System installiert ist.

Die Anfangseinstellungen einer CoolBar-Komponente festlegen

CoolBar-Komponenten bieten verschiedene nützliche Konfigurationsmöglichkeiten. Die folgende Tabelle enthält Informationen zu den Eigenschaften, die das Erscheinungsbild von CoolBar-Komponenten beeinflussen.

Tabelle 5.5 Anzeigeeigenschaften für CoolBar-Komponenten

Verhalten	Aktion
Die Größe der Komponente wird automatisch an ihre Bereiche angepaßt.	Setzen Sie *AutoSize* auf *True*
Alle Bereiche erhalten dieselbe Höhe.	Setzen Sie *FixedSize* auf *True*.
Die Bereiche werden nicht horizontal, sondern vertikal angeordnet.	Setzen Sie *Vertical* auf *True*. *FixedSize* wirkt sich nun auf die Breite aus.
Die Beschriftungen (Eigenschaft *Text*) der Bereiche werden zur Laufzeit nicht angezeigt.	Setzen Sie *ShowText* auf *False*. Jeder Bereich hat seine eigene Eigenschaft Text.
Der Rahmen um die Bereiche wird nicht angezeigt.	Setzen Sie *BandBorderStyle* auf *bsNone*
Die Reihenfolge der Bereiche kann zur Laufzeit nicht geändert werden (Größe und Position können jedoch geändert werden).	Setzen Sie *FixedOrder* auf *True*
In der Komponente wird eine Hintergrundgrafik angezeigt.	Weisen Sie *Bitmap* ein *TBitmap*-Objekt zu.
Auf der linken Seite jedes Bereichs wird eine Grafik angezeigt.	Weisen Sie *Images* ein *TImageList*-Objekt zu.

Wenn Sie einzelnen Bereichen eine Grafik zuweisen wollen, wählen Sie die *CoolBar*-Komponente aus und doppelklicken im Objektinspektor auf die Eigenschaft *Bands*. Klicken Sie anschließend auf einen Bereich in der Liste, und weisen Sie seiner Eigenschaft *ImageIndex* den Index der gewünschten Grafik zu.

Auf Mausklicks reagieren

Wenn der Benutzer zur Laufzeit auf ein Steuerelement (z. B. eine Schaltfläche) klickt, generiert die Anwendung ein *OnClick*-Ereignis, das Sie in einer Ereignisbehandlungsroutine verarbeiten können. Da *OnClick* das Standardereignis für Schaltflächen ist, können Sie ein Gerüst der Behandlungsroutine für das Ereignis generieren, indem Sie zur Entwurfszeit doppelt auf die Schaltfläche klicken. Weitere Informationen finden Sie unter »Mit Ereignissen und Ereignisbehandlungsroutinen arbeiten« auf Seite 2-28 und »Eine Behandlungsroutine für das Standardereignis einer Komponente erstellen« auf Seite 2-29.

Einer ToolButton-Komponente ein Menü zuweisen

Sie können den ToolButton-Komponenten einer Symbolleiste (*TToolBar*) auch Popup-Menüs zuweisen. Gehen Sie dazu folgendermaßen vor:

1 Wählen Sie eine Schaltfläche aus.

2 Weisen Sie der Eigenschaft *DropDownMenu* im Objektinspektor ein Popup-Menü (*TPopupMenu*) zu.

Wenn die Eigenschaft *AutoPopup* des Menüs *True* ist, wird es automatisch geöffnet, sobald der Benutzer auf die Schaltfläche klickt.

Verborgene Symbolleisten hinzufügen

Symbolleisten müssen zur Laufzeit nicht dauernd sichtbar sein. Es ist sogar in vielen Situationen sinnvoller, mehrere Symbolleisten in Bereitschaft zu halten und nur diejenigen anzuzeigen, mit denen der Benutzer arbeiten will. Dazu werden mehrere Symbolleisten benötigt, die jedoch nicht alle angezeigt werden.

So erstellen Sie eine verborgene Symbolleiste:

1 Plazieren Sie eine ToolBar-, CoolBar- oder Panel-Komponente im Formular.

2 Setzen Sie die Eigenschaft *Visible* der Komponente auf *False*.

Die Symbolleiste bleibt während der gesamten Entwurfsphase sichtbar, damit sie bearbeitet werden kann. Zur Laufzeit wird sie jedoch erst angezeigt, wenn ihre Eigenschaft *Visible* auf *True* gesetzt wird.

Symbolleisten ein- und ausblenden

In vielen Anwendungen werden zwar mehrere Symbolleisten verwendet, aber aus Gründen der Ergonomie nicht zur selben Zeit angezeigt. Der Benutzer soll selbst entscheiden, welche Symbolleisten jeweils zu sehen sind. Symbolleisten können wie alle anderen Steuerelemente zur Laufzeit angezeigt oder ausgeblendet werden.

Um eine Symbolleiste ein- oder auszublenden, setzen Sie einfach ihre Eigenschaft *Visible* auf *True* oder *False*. Normalerweise geschieht dies als Reaktion auf ein bestimmtes Benutzerereignis oder eine Änderung im Modus der Anwendung. Die meisten Symbolleisten enthalten eine Schaltfläche, mit der sie geschlossen (ausgeblendet) werden können.

Sie können auch eine Möglichkeit anbieten, die Symbolleiste zur Laufzeit mit Hilfe einer anderen Komponente ein- oder auszublenden. Im folgenden Beispiel ist dies ein Ein-/Ausschalter in der Hauptsymbolleiste. In seiner *OnClick*-Ereignisbehandlungsroutine wird die Anzeige der Werkzeugleiste mit dem Status des Ein-/Ausschalters synchronisiert.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
      PenBar.Visible := PenButton.Down;
end;
```

Aktionslisten verwenden

Mit Hilfe von Aktionslisten können Reaktionen auf Benutzerbefehle (Aktionen) für Komponenten wie Menüs und Schaltflächen zentral verwaltet werden. In diesem Abschnitt erfahren Sie, wie diese Objekte verwendet werden und wie sie mit ihren Clients und Zielkomponenten interagieren.

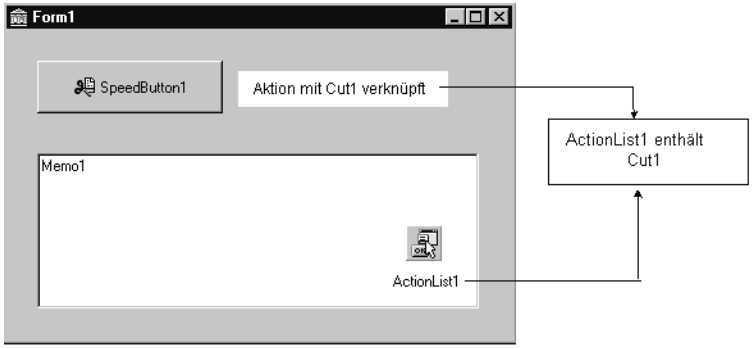
Aktionsobjekte

Aktionen können mit Hilfe des Aktionslisten-Editors erstellt werden. Sie definieren Benutzerbefehle, die mit einem bestimmten Zielobjekt ausgeführt werden. Die Verbindung mit dem Client-Steuerelement wird durch die zugehörige Aktionsverknüpfung hergestellt. Die folgende Übersicht zeigt die verschiedenen Komponenten des Aktions-/Aktionslistenmechanismus:

- Eine Aktion (*TAction*) implementiert eine Operation (z. B. Kopieren von markiertem Text), die mit einem bestimmten Zielobjekt (z. B. einem Eingabefeld) durchgeführt wird. Sie wird vom Client als Reaktion auf einen Benutzerbefehl (wie ein Mausklick) ausgelöst. Die beteiligten Clients sind normalerweise Menüeinträge oder Schaltflächen. Die Unit *Stdactns* enthält spezielle, von *TAction* abgeleitete Klassen, in denen die Grundbefehle der Windows-Standardmenüs Bearbeiten und Fenster implementiert sind.
- Eine Aktionsliste (*TActionList*) ist eine Komponente, die eine Liste von Aktionen (*TAction*) verwaltet. Sie stellt während des Entwurfs eine Benutzeroberfläche für die Arbeit mit Aktionen zur Verfügung.
- Eine Aktionsverknüpfung (*TActionLink*) ist ein Objekt, das die Verbindung zwischen Aktionen und Clients verwaltet. Sie legt fest, welche Aktion momentan für einen bestimmten Client verwendet wird.
- Ein Client einer Aktion ist normalerweise eine Menüoption oder eine Schaltfläche (*TToolButton, TSpeedButton, TMenuItem, TButton, TCheckBox, TRadioButton* usw.). Aktionen werden durch einen entsprechenden Befehl in der Client-Komponente gestartet. Normalerweise wird zu diesem Zweck die Methode *Click* des Clients der Methode *Execute* einer Aktion zugeordnet.
- Ein Aktionsziel ist normalerweise ein Steuerelement (z. B. ein Eingabefeld, ein Memofeld oder ein datensensitives Steuerelement). Die Unit *Dbactns* enthält beispielsweise verschiedene Klassen, in denen spezielle Aktionen für die Arbeit mit Datenmengen implementiert sind. Komponentenentwickler können spezielle Aktionen für ihre Steuerelemente definieren und aus diesen Units Packages erstellen. Auf diese Weise entstehen modulare Anwendungen.

Die folgende Abbildung zeigt die Beziehungen zwischen diesen Objekten. *Cut1* ist die Aktion. *ActionList1* ist die Aktionsliste, die *Cut1* enthält. *SpeedButton1* ist der Client von *Cut1*. *Memo1* ist die Zielkomponente. Da Aktionsverknüpfungen nichtvisuelle Objekte sind, wird die Verknüpfung in der Abbildung durch ein weißes Rechteck angezeigt. Sie verknüpft den Client *SpeedButton1* mit der Aktion *Cut1* in *ActionList1*.

Abbildung 5.10 Aktionslistenmechanismus



In der VCL stehen für die Arbeit mit Aktionslisten die Klassen *TAction*, *TActionList* und *TActionLink* zur Verfügung. Sie befinden sich in folgenden Units:

- ACTNLIST.PAS: *TAction, TActionLink, TActionList, TContainedAction, TCustomAction* und *TCustomActionList*.
- CLASSES.PAS: *TBasicAction* und *TBasicActionLink*.
- CONTROLS.PAS: *TControlActionLink* und *TWinControlActionLink*.
- COMCTRLS.PAS: *TToolButtonActionLink*.
- MENUS.PAS: *TMenuActionLink*.
- STDCTRLS.PAS: *TButtonActionLink*.

Die beiden Units *Stdactns* und *Dbactns* enthalten verschiedene Hilfsklassen, die häufig benötigte Windows-Standardaktionen und Datenmengenaktionen implementieren. Informationen hierzu finden Sie im Abschnitt »Vordefinierte Aktionsklassen« auf Seite 5-43. Viele VCL-Steuerelemente verfügen über Eigenschaften (wie *Action*) und Methoden (wie *ExecuteAction*), durch die sie als Clients und Zielobjekte für Aktionen verwendet werden können.

Aktionen verwenden

Mit Hilfe der entsprechenden Komponente in der Registerkarte *Standard* der Komponentenpalette fügen Sie einem Formular oder Datenmodul eine Aktionsliste hinzu. Doppelklicken Sie anschließend auf das Objekt, um den Aktionslisten-Editor zu öffnen. Dort können Sie Aktionen hinzufügen, entfernen oder neu anordnen.

Im Objektinspektor können Sie anschließend die verschiedenen Eigenschaften der einzelnen Aktionen festlegen. *Name* gibt den Namen der Aktion an. Die anderen Eigenschaften (*Caption, Checked, Enabled, HelpContext, Hint, ImageIndex, ShortCut* und *Visible*) entsprechen den Eigenschaften der Client-Steuerelemente. Sie haben normalerweise (aber nicht unbedingt) denselben Namen wie die jeweilige Client-Eigenschaft. So entspricht beispielsweise die Eigenschaft *Checked* einer Aktion der Eigenschaft *Down* einer *TToolButton*-Komponente.

Quelltext zentral verwalten

Einige Steuerelemente wie *TToolButton*, *TSpeedButton*, *TMenuItem* und *TButton* verfügen über die **published**-Eigenschaft *Action*. Wenn Sie dieser Eigenschaft eine Aktion zuweisen, werden die Werte der entsprechenden Aktionseigenschaften in die Eigenschaften des Steuerelements kopiert. Dabei findet eine dynamische Verknüpfung zwischen den gemeinsamen Eigenschaften und Ereignissen statt (mit Ausnahme von *Name* und *Tag*). Sie können daher mit Hilfe eines Aktionsobjekts Quelltext zentral verwalten, der für mehrere Steuerelemente verwendet werden soll. Um beispielsweise verschiedene Schaltflächen und Menüoptionen zu aktivieren oder zu deaktivieren, brauchen Sie den Quelltext nur einmal für das Aktionsobjekt zu erstellen. Wenn die Aktion in der Anwendung deaktiviert wird, betrifft dies automatisch alle mit ihr verknüpften Steuerelemente.

Eigenschaften verknüpfen

Den Mechanismus, durch den die Eigenschaften eines Clients mit den Eigenschaften einer Aktion verbunden werden, bezeichnet man als *Aktionsverknüpfung*. Sobald sich die Werte einer Aktion ändern, aktualisiert die Aktionsverknüpfung automatisch die entsprechenden Client-Eigenschaften. Informationen dazu, welche Eigenschaften von einer bestimmten Verknüpfungsklasse behandelt werden, finden Sie bei den einzelnen Klassen in der VCL-Referenz.

Sie können die Werte der Eigenschaften, die von der zugeordneten Aktion gesteuert werden, auch selektiv überschreiben. Weisen Sie einfach der Eigenschaft der Client-Komponente den gewünschten Wert zu. Die Änderung betrifft nur den Client und wirkt sich nicht auf den Wert der Aktionseigenschaft aus.

Aktionen ausführen

Wenn der Benutzer auf eine Client-Komponente klickt, wird das Ereignis *OnExecute* der verknüpften Aktion ausgelöst. Auf diese Weise wird in der folgenden Ereignisbehandlungsroutine der Sichtbarkeitsstatus einer Symbolleiste geändert:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
      { Toolbar1 ein- und ausblenden }
      ToolBar1.Visible := not ToolBar1.Visible;
end;
```

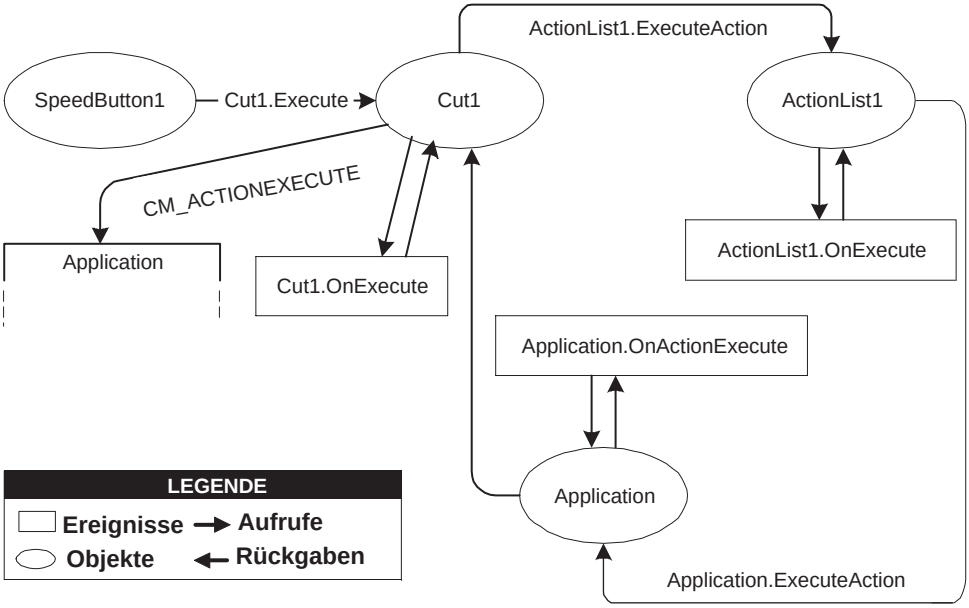
Hinweis Der Wert der Eigenschaft *Images* einer Aktionsliste muß der Eigenschaft *Images* eines Menüeintrags oder ToolButton-Objekts manuell zugewiesen werden. Dies gilt auch dann, wenn die Eigenschaft *ImageIndex* dynamisch mit dem Client verknüpft ist.

Allgemeine Informationen zu Ereignissen und Ereignisbehandlungsroutinen finden Sie unter »Mit Ereignissen und Ereignisbehandlungsroutinen arbeiten« auf Seite 2-28.

Die folgende Abbildung zeigt den Ausführungszylus der Aktion *Cut1*. Dabei gelten die Komponentenbeziehungen aus Abbildung 5.10, in der die Client-Komponente *Speedbutton1* durch eine Aktionsverknüpfung mit *Cut1* verbunden ist. Da die *Action*-

Eigenschaft von *Speedbutton1* den Wert *Cut1* hat, ruft die Methode *Click* von *Speedbutton1* die Methode *Execute* von *Cut1* auf.

Abbildung 5.11 Ausführungszyklus einer Aktion



Hinweis In den folgenden Beschreibungen bedeutet das Aufrufen einer Methode durch eine andere nicht unbedingt, daß der Aufruf explizit im Quelltext durchgeführt wird.

Wenn der Benutzer auf *Speedbutton1* klickt, wird folgender Ausführungszyklus in Gang gesetzt:

- In der *Click*-Methode von *Speedbutton1* wird *Cut1.Execute* aufgerufen.
- Die Aktion *Cut1* überläßt die Verarbeitung ihrer *Execute*-Methode der Aktionsliste (*ActionList1*). Sie ruft dazu die Methode *ExecuteAction* der Aktionsliste auf und übergibt sich selbst als Parameter.
- *ActionList1* ruft ihre Ereignisbehandlungsroutine (*OnExecute*) für *ExecuteAction* auf (die Methode *ExecuteAction* einer Aktionsliste wird für alle ihre Aktionen verwendet). Diese Routine hat einen Parameter (*Handled*), der als Standardwert *False* zurückgibt. Wenn die Routine zugewiesen ist und das Ereignis behandeln kann, gibt sie wie im folgenden Beispiel *True* zurück, und die Verarbeitung wird beendet:

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
  { Ausführung aller Aktionen in ActionList1 verhindern. }
  Handled := True;
end;
```

Findet in der Aktionsliste keine Behandlung statt, wird die Verarbeitung fortgesetzt:

Die Aktion *Cut1* wird an die Methode *ExecuteAction* des *Application*-Objekts übergeben. Diese ruft die Ereignisbehandlungsroutine für *OnExecuteAction* auf (die Metho-

de *ExecuteAction* des Anwendungsobjekts wird für alle Aktionen in der Anwendung verwendet). Die weitere Ausführung entspricht der Methode *ExecuteAction* der Aktionsliste: die Routine hat einen Parameter (*Handled*), der als Standardwert *False* zurückgibt. Wenn die Routine zugewiesen ist und das Ereignis behandeln kann, gibt sie wie im folgenden Beispiel *True* zurück, und die Verarbeitung wird beendet:

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
  { Ausführung aller Aktionen in der Anwendung verhindern. }
  Handled := True;
end;
```

Findet auch in der Ereignisbehandlungsroutine des Anwendungsobjekts keine Behandlung statt, sendet *Cut1* die Botschaft CM_ACTIONEXECUTE an die Methode *WndProc* des Anwendungsobjekts und übergibt sich selbst als Parameter. Die Anwendung versucht dann, eine Zielkomponente für die Aktion zu finden (siehe Abbildung 5.12, »Aktionsziele,« auf Seite 5-46).

Aktionen aktualisieren

In inaktiven Phasen der Anwendung wird für jede Aktion, die mit einem sichtbaren Steuerelement oder Menüeintrag verknüpft ist, das Ereignis *OnUpdate* ausgelöst. Auf diese Weise können Sie zentralen Programmcode ausführen, in dem Komponenten aktiviert, deaktiviert oder ähnliche Operationen durchgeführt werden. In der folgenden *OnUpdate*-Routine wird beispielsweise eine Aktion aktiviert, wenn die Symbolleiste sichtbar ist:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
      { Feststellen, ob ToolBar1 sichtbar ist }
      (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

Sehen Sie sich zu diesem Thema auch die RichEdit-Beispielanwendung (DEMO\RICHEDIT) an.

Die Ausführungsfolge bei Aktualisierungsaktionen entspricht dem Ausführungszyklus in Abbildung 5.11.

Hinweis Führen Sie in einer Ereignisbehandlungsroutine für *OnUpdate* keine zeitintensiven Operationen durch. Da die Routine immer ausgeführt wird, wenn die Anwendung inaktiv ist, wirken sich die Operationen auf das Zeitverhalten der gesamten Anwendung aus.

Vordefinierte Aktionsklassen

Die Klassen in den Units *Stdactns* und *Dbactns* sind gute Beispiele dafür, wie Sie spezielle Aktionsklassen für eigene Komponenten ableiten können. In den Basisklassen dieser spezialisierten Aktionen (*TEditAction, TWindowAction*) werden *HandlesTarget*, *UpdateTarget* und andere Methoden überschrieben, um das Ziel der betreffenden Aktion auf eine bestimmte Objektklasse einzuschränken. In den abgeleiteten Klassen

wird normalerweise *ExecuteTarget* überschrieben, um klassenspezifische Operationen durchzuführen.

Standard-Bearbeitungsaktionen

Die Standard-Bearbeitungsaktionen (Basisklasse *TEditAction*) werden für Eingabefelder verwendet. In allen abgeleiteten Klassen wird die Methode *ExecuteTarget* überschrieben, um das Ausschneiden, Kopieren und Einfügen mit Hilfe der Windows-Zwischenablage zu implementieren.

- *TEditAction* stellt sicher, daß als Zielkomponente ein Objekt der Klasse *TCustomEdit* (oder einer abgeleiteten Klasse) verwendet wird.
- *TEditCopy* kopiert markierten Text in die Zwischenablage.
- *TEditCut* schneidet markierten Text aus und kopiert ihn in die Zwischenablage.
- *TEditPaste* fügt Text aus der Zwischenablage in die Zielkomponente ein und stellt sicher, daß die Zwischenablage das Textformat verwenden kann.

Windows-Standardaktionen

Die Windows-Standardaktionen (Basisklasse *TWindowAction*) werden für die Formulare einer MDI-Anwendung verwendet. In allen abgeleiteten Klassen wird die Methode *ExecuteTarget* überschrieben, um das Anordnen, Schließen und Minimieren der untergeordneten MDI-Formulare durchzuführen.

- *TWindowAction* stellt sicher, daß als Zielkomponente ein Objekt der Klasse *TForm* verwendet wird, und prüft, ob untergeordnete MDI-Formulare vorhanden sind.
- *TWindowArrange* ordnet die Symbole der minimierten MDI-Formulare an.
- *TWindowCascade* ordnet die MDI-Formulare übereinander an.
- *TWindowClose* schließt das aktive MDI-Formular.
- *TWindowMinimizeAll* minimiert alle MDI-Formulare.
- *TWindowTileHorizontal* ordnet die MDI-Formulare nebeneinander an, wobei alle dieselbe Größe haben.
- *TWindowTileVertical* ordnet die MDI-Formulare untereinander an, wobei alle dieselbe Größe haben.

Datenmengenaktionen

Die Datenmengenaktionen (Basisklasse *TDataSetAction*) können für eine Datenmengenkomponente verwendet werden. In allen abgeleiteten Klassen werden die Methoden *ExecuteTarget* und *UpdateTarget* überschrieben, um das Navigieren und Bearbeiten in der Zielkomponente zu implementieren.

In *TDataSetAction* ist die Eigenschaft *DataSource* definiert. Sie stellt sicher, daß die Aktionen mit der gewünschten Datenmenge durchgeführt werden. Hat *DataSource* den Wert **nil**, wird das aktive datensensitive Steuerelement verwendet. Einzelheiten dazu können Sie der Abbildung 5.12, »Aktionsziele,« auf Seite 5-46 entnehmen.

- *TDataSetAction* stellt sicher, daß als Ziel ein Objekt der Klasse *TDataSource* verwendet wird und diesem eine Datenmenge zugeordnet ist.
- *TDataSetCancel* bricht die Bearbeitung des aktuellen Datensatzes ab, stellt die vorherigen Feldwerte wieder her und deaktiviert den Einfügen- und Bearbeiten-Modus.
- *TDataSetDelete* löscht den aktuellen und positioniert auf den nächsten Datensatz.
- *TDataSetEdit* aktiviert den Bearbeiten-Modus, damit der aktuelle Datensatz geändert werden kann.
- *TDataSetFirst* positioniert auf den ersten Datensatz.
- *TDataSetInsert* fügt vor dem aktuellen einen neuen Datensatz ein und aktiviert den Einfügen- und Bearbeiten-Modus.
- *TDataSetLast* positioniert auf den letzten Datensatz.
- *TDataSetNext* positioniert auf den nächsten Datensatz.
- *TDataSetPost* schreibt die Änderungen des aktuellen Datensatzes in die Datenmenge.
- *TDataSetPrior* positioniert auf den vorherigen Datensatz.
- *TDataSetRefresh* aktualisiert den Datensatzpuffer.

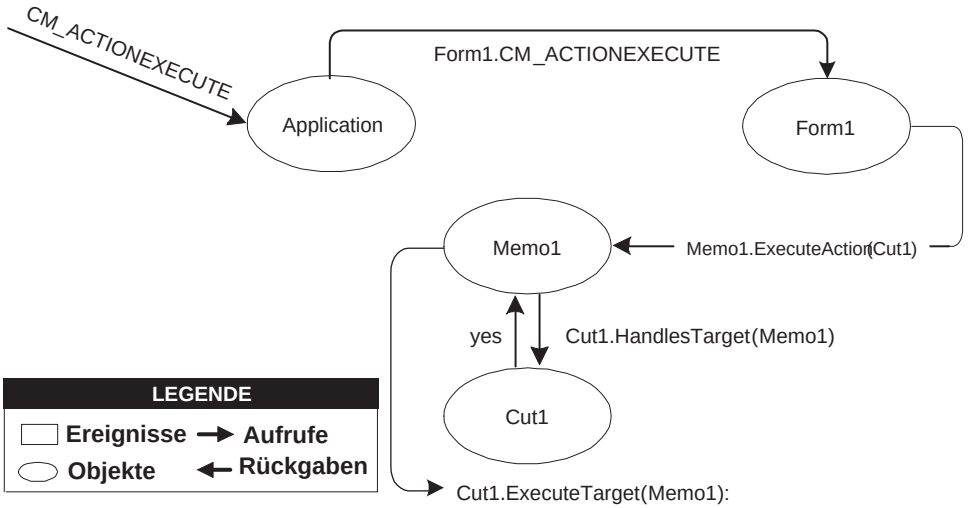
Aktionskomponenten erstellen

Die vordefinierten Aktionen sind Beispiele dafür, wie die VCL-Aktionsklassen erweitert werden können. Wenn Sie eigene Aktionsklassen erstellen wollen, finden Sie hier weitere Informationen:

Wie Aktionen ihre Zielobjekte lokalisieren

Abbildung 5.11 zeigt den Ausführungszyklus der VCL-Standardaktionsklassen. Wenn eine Behandlung durch die Aktionsliste, durch die Anwendung oder durch Standard-Ereignisbehandlungsroutinen nicht möglich ist, wird die Botschaft CM_ACTIONEXECUTE an die Methode *WndProc* des Anwendungsobjekts gesendet. Die folgende Abbildung zeigt, wie die Ausführung an diesem Punkt fortgesetzt wird. Diese Vorgehensweise gilt für die zuvor beschriebenen vordefinierte Aktionsklassen und für alle selbsterstellten Aktionsklassen.

Abbildung 5.12 Aktionsziele



- Das Anwendungsobjekt leitet die Botschaft CM_ACTIONEXECUTE zuerst an das *ActiveForm* des Bildschirmobjekts weiter. Ist kein aktives Formular vorhanden, wird die Botschaft an das *ActiveForm* gesendet.
- *Form1* (in diesem Beispiel das aktive Formular) sucht zuerst nach dem aktiven Steuerelement (*Memo1*) und ruft dessen Methode *ExecuteAction* mit *Cut1* als Parameter auf.
- *Memo1* ruft die Methode *HandlesTarget* von *Cut1* auf und übergibt sich selbst als Parameter, damit bestimmt werden kann, ob es als Ziel geeignet ist. Ist *Memo1* kein zulässiges Ziel, geben *HandlesTarget* und die *ExecuteAction*-Ereignisbehandlungsroutine von *Memo1* den Wert *False* zurück.
- Da *Memo1* hier ein gültiges Ziel für *Cut1* ist, gibt *HandlesTarget* den Wert *True* zurück. *Memo1* ruft dann *Cut1.ExecuteTarget* mit sich selbst als Parameter auf. Zum Schluß wird die Methode *Memo1.CutToClipoard* aufgerufen, da *Cut1* eine Instanz einer *TEditCut*-Aktion ist:

```
procedure TEditCut.ExecuteTarget(Target: TObject);
begin
  GetControl(Target).CutToClipboard;
end;
```

Wenn das Steuerelement kein geeignetes Ziel ist, wird die Verarbeitung folgendermaßen fortgesetzt:

- *Form1* ruft seine Methode *ExecuteAction* auf. Ist *Form1* ein zulässiges Ziel (ein Formular ist beispielsweise ein Ziel für die Aktion *TWindowCascade*), ruft es die Methode *ExecuteTarget* von *Cut1* auf und übergibt sich selbst als Parameter.
- Wenn *Form1* kein geeignetes Ziel ist, ruft es die Methode *ExecuteAction* aller in ihm enthaltenen sichtbaren Steuerelemente auf, bis das Zielobjekt gefunden wird.

Hinweis Eine Aktion vom Typ *TCustom*Action wird automatisch deaktiviert, wenn sie nicht behandelt wird und ihre Eigenschaft *DisableIfNoHandler* den Wert *True* hat.

Aktionen registrieren

Mit Hilfe der globalen Routinen in der Unit *ActnList* können Sie auch eigene Aktionen in der IDE registrieren bzw. ihre Registrierung aufheben:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);

procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

Gehen Sie dabei genauso vor wie beim Registrieren von Komponenten mit *RegisterComponents*. Die Standardaktionen in der Unit *StdReg* werden beispielsweise folgendermaßen registriert:

```
{ Registrierung der Standardaktionen }

RegisterActions('', [TAction], nil);

RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);

RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

Einen Aktionslisten-Editor erstellen

Sie können einen eigenen Komponenteneditor für Aktionslisten erstellen. Weisen Sie dabei den vier globalen Variablen in der Unit *ActnList* eigene Prozeduren zu:

```
CreateActionProc: function (AOwner: TComponent; ActionClass: TBasicActionClass):
TBasicAction = nil;

EnumRegisteredActionsProc: procedure (Proc: TEnumActionProc; Info: Pointer) = nil;

RegisterActionsProc: procedure (const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass) = nil;

UnRegisterActionsProc: procedure (const AClasses: array of TBasicActionClass) = nil;
```

Sie brauchen diese Variablen nur dann neu zuzuweisen, wenn das Registrieren, Aufheben der Registrierung, Erstellen und Aufzählen der Aktionen anders als in der Standardimplementierung durchgeführt werden soll. Schreiben Sie dazu eigene Routinen, und weisen Sie diese im initialization-Abschnitt Ihrer Entwurfszeit-Unit den Variablen zu.

Beispielprogramme

Ein Beispielprogramm, in dem Aktionen und Aktionslisten verwendet werden, finden Sie im Verzeichnis DELPHI\DEMO\RICHEDIT. Die Objekte können auch in den Anwendungs-Experten (Registerkarte *Datei / Neu / Projekte)* für MDI-, SDI- und Win95-Logo-Anwendungen verwendet werden.

Kapitel

6

Mit Steuerelementen arbeiten

Steuerelemente sind visuelle Komponenten, mit denen der Benutzer zur Laufzeit interagieren kann. Dieses Kapitel beschreibt eine Reihe von Merkmalen, über die viele Steuerelemente verfügen.

Drag&Drop-Operationen in Steuerelementen implementieren

Drag&Drop bietet dem Benutzer häufig eine komfortable Möglichkeit zum Bearbeiten von Objekten. Sie können Benutzern das Ziehen eines Steuerelements oder das Ziehen von Einträgen aus einem Steuerelement (beispielsweise einem Listenfeld oder Baumdiagramm) in ein anderes ermöglichen.

- Eine Drag-Operation beginnen
- Gezogene Elemente akzeptieren
- Elemente ablegen
- Eine Drag-Operation beenden
- Drag&Drop-Operationen durch ein Drag-Objekt anpassen
- Den Drag-Mauszeiger ändern

Eine Drag-Operation beginnen

Jedes Steuerelement verfügt über die Eigenschaft *DragMode*. Diese bestimmt, wie Drag-Operationen eingeleitet werden. Hat *DragMode* den Wert *dmAutomatic*, beginnt die Drag-Operation automatisch, sobald der Benutzer innerhalb der Komponente eine Maustaste drückt. Da diese Einstellung zu Konflikten mit normalen Mausaktivitäten führen kann, sollten Sie *DragMode* auf den Standardwert *dmManual* setzen und die Drag-Operation in einer Ereignisbehandlungsroutine für *OnMouseDown* starten.

Sie können die Drag-Operation manuell starten, indem Sie die Methode *BeginDrag* der Komponente aufrufen. An *BeginDrag* wird der Boolesche Parameter *Immediate* übergeben. Hat er den Wert *True*, beginnt die Operation sofort. Wenn Sie den Wert *False* übergeben, wird die Drag-Operation erst gestartet, wenn der Benutzer die Maus bewegt. Der Aufruf `BeginDrag(False)` ermöglicht Mausklicks, ohne eine Drag-Operation zu starten.

Eine Drag-Operation kann von weiteren Bedingungen abhängig gemacht werden. Sie können beispielsweise prüfen, welche Maustaste gedrückt wurde, indem Sie vor dem Aufruf von *BeginDrag* die Parameter des *OnMouseDown*-Ereignisses auswerten. Im folgenden Beispiel wird in der Ereignisbehandlungsroutine *OnMouseDown* einer Dateiliste eine Drag-Operation eingeleitet, wenn die linke Maustaste gedrückt ist.

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
      if Button = mbLeft then  { Nur ziehen, wenn linke Maustaste gedrückt ist }
         with Sender as TFileListBox do  { Sender als TFileListBox behandeln }
         begin
            if ItemAtPos(Point(X, Y), True) >= 0 then  { Element vorhanden? }
            BeginDrag(False);  { Wenn ja, dann ziehen }
         end;
end;
```

Gezogene Elemente akzeptieren

Wenn der Benutzer ein Objekt über ein Steuerelement zieht, wird für dieses Steuerelement ein *OnDragOver*-Ereignis ausgelöst. Das Steuerelement muß nun entscheiden, ob es das Objekt akzeptiert. Ist dies der Fall, ändert sich die Form des Mauszeigers. Um Objekte zu akzeptieren, die über ein Steuerelement gezogen werden, erstellen Sie für dessen Ereignis *OnDragOver* eine Behandlungsroutine.

Die Behandlungsroutine verfügt über den Parameter *Accept*, der auf *True* gesetzt wird, wenn das Element abgelegt werden kann. Wenn *Accept True* ist, kann die Anwendung ein *OnDragDrop*-Ereignis an das Steuerelement senden.

Die weiteren Parameter der Behandlungsroutine bezeichnen beispielsweise die Quelle der Drag-Operation und die aktuelle Mausposition. Mit Hilfe dieser Informationen kann die Ereignisbehandlungsroutine bestimmen, ob das Ablegen zulässig ist.

Im folgenden Beispiel können in einem Verzeichnisdiagramm nur Objekte abgelegt werden, die aus einer Dateiliste stammen:

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True;
  else
    Accept := False;
end;
```

Elemente ablegen

Akzeptiert ein Steuerelement gezogene Objekte, muß auch die Behandlung dieser Objekte definiert werden. Um mit den abgelegten Objekten Aktionen durchzuführen, ordnen Sie dem Ereignis *OnDragDrop* des Steuerelements eine Behandlungsroutine zu.

Dieses Ereignis gibt wie *OnDragOver* die Quelle des gezogenen Objekts und die Koordinaten des Mauszeigers an. Der letzte Parameter ermöglicht die Überwachung des Pfads eines Elements beim Ziehen. Sie können also beispielsweise die Farbe von Komponenten ändern, während ein Element über die betreffenden Komponenten gezogen wird.

Im folgenden Beispiel werden Dateien durch Ablegen in einem Verzeichnisdiagramm in das betreffende Verzeichnis verschoben:

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileList.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

Eine Drag-Operation beenden

Eine Drag-Operation endet, wenn das Element erfolgreich abgelegt (Drop) oder über einem Steuerelement freigegeben wurde, welches das Objekt entgegennehmen kann. Zu diesem Zeitpunkt wird ein Ereignis zum Abschluß der Drag-Operation an das Steuerelement gesendet, aus dem das Element gezogen wurde. Damit ein Steuerelement auf das Ziehen eines seiner Objekte reagieren kann, weisen Sie seinem Ereignis *OnEndDrag* eine Behandlungsroutine zu.

Der wichtigste Parameter des Ereignisses *OnEndDrag* ist *Target*. Er gibt an, in welchem Steuerelement das Objekt abgelegt werden kann. Hat *Target* den Wert **nil**, wird das gezogene Objekt von keinem Steuerelement akzeptiert. Die weiteren Parameter geben die Koordinaten des Steuerelements an.

Im folgenden Quelltext wird eine Dateiliste durch ein *OnEndDrag*-Ereignis aktualisiert:

```
procedure TFMForm.FileList1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileList1.Update;
end;
```

Drag&Drop-Operationen durch ein Drag-Objekt anpassen

Mit Hilfe eines Nachkommens von *TDragObject* können Sie das Drag&Drop-Verhalten eines Objekts anpassen. Standardmäßig geben die Ereignisse *OnDragOver* und *OnDragDrop* die Quellkomponente des gezogenen Objekts und die Koordinaten des

Mauszeigers im Ziel der Operation an. Sie können jedoch weitere Statusinformationen abrufen, indem Sie von *TDragObject* ein benutzerdefiniertes Drag-Objekt ableiten und die virtuellen Methoden nach Bedarf überschreiben.

Erstellen Sie das benutzerdefinierte Drag-Objekt in der Ereignisbehandlungsroutine für *OnStartDrag*. Normalerweise enthält der Parameter *Source* von *Drag-Over*- und *Drag-Drop*-Ereignissen das Steuerelement, das die Drag-Operation eingeleitet hat. Wenn unterschiedliche Arten von Steuerelementen eine Operation mit derselben Art von Daten starten können, muß die Quelle jeden dieser Steuerelementtypen unterstützen. Verwenden Sie einen Nachfahren von *TDragObject*, bildet das Drag-Objekt selbst die Quelle. Erstellt jedes Steuerelement dieselbe Art von Drag-Objekt in seinem *OnStartDrag*-Ereignis, muß das Ziel nur eine Art von Objekt verarbeiten. Die Drag-Over- und Drag-Drop-Ereignisse können also im Unterschied zum Steuerelement angeben, ob es sich bei der Quelle um ein Drag-Objekt handelt, indem die Funktion *IsDragObject* aufgerufen wird.

Drag-Objekte ermöglichen das Ziehen von Objekten zwischen einem in der ausführbaren Hauptdatei (EXE) der Anwendung implementierten Formular und einem Formular, das in einer DLL implementiert ist (oder zwischen Formularen, die in verschiedenen DLLs implementiert sind).

Den Drag-Mauszeiger ändern

Sie können die Form des Mauszeigers während einer Drag-Operation ändern, indem Sie zur Entwurfszeit der Eigenschaft *DragCursor* der Komponente den gewünschten Wert zuweisen.

Drag&Dock in Steuerelementen implementieren

Jeder Nachkomme von *TWinControl* kann Ziel einer Andock-Operation sein. Die Nachkommen von *TControl* können als untergeordnete Fenster definiert werden, die an mögliche Ziele einer Andock-Operation angedockt werden. Um beispielsweise den linken Rand eines Formularfensters zum Ziel einer Andock-Operation zu machen, richten Sie dort eine Tafel aus und definieren sie als Ziel für Andock-Operationen. Wenn der Benutzer andockbare Steuerelemente auf diese Tafel zieht und sie dort ablegt, werden sie zu untergeordneten Steuerelementen der Tafel.

- Ein fensterorientiertes Steuerelement als Ziel einer Andock
- Ein Steuerelement als andockbares Steuerelement definieren
- Andock-Operationen von Steuerelementen steuern
- Steuerelemente vom Ziel einer Andock-Operation trennen
- Die Behandlung von Drag&Dock-Operationen durch Steuerelemente festlegen

Ein fensterorientiertes Steuerelement als Ziel einer Andock-Operation definieren

So definieren Sie ein fensterorientiertes Steuerelement als Ziel einer Andock-Operation:

1. Setzen Sie die Eigenschaft *DockSite* auf *True*.
2. Soll das Zielobjekt nur angezeigt werden, wenn es einen angedockten Client enthält, setzen Sie seine Eigenschaft *AutoSize* auf *True*. Dies bewirkt, daß die Größe des Ziels 0 ist, bis ein untergeordnetes Steuerelement angedockt wird. Erst dann wird die Größe an das untergeordnete Steuerelement angepaßt.

Ein Steuerelement als andockbares Steuerelement definieren

So definieren Sie ein Steuerelement als andockbares Steuerelement:

1. Weisen Sie der Eigenschaft *DragKind* des Steuerelements den Wert *dkDock* zu. Bei diesem Wert bewirkt das Ziehen des Steuerelements, daß es an ein anderes Ziel angedockt oder vom aktuellen Ziel getrennt und dadurch zu einem schwebenden Fenster wird. Hat *DragKind* den Wert *dkDrag* (Standardeinstellung), wird durch das Ziehen des Steuerelements eine Drag&Drop-Operation ausgelöst, die mit den Ereignissen *OnDragOver*, *OnEndDrag* und *OnDragDrop* implementiert werden muß.
2. Weisen Sie der Eigenschaft *DragMode* des Steuerelements den Wert *dmAutomatic* zu. Bei diesem Wert wird die Drag-Operation (Drag&Drop oder Drag&Dock entsprechend dem unter *DragKind* eingestellten Wert) automatisch eingeleitet, wenn der Benutzer beginnt, das Steuerelement mit der Maus zu ziehen. Hat *DragMode* den Wert *dmManual*, können Sie eine Drag&Dock- bzw. eine Drag&Drop-Operation starten, indem Sie die Methode *BeginDrag* aufrufen.
3. Weisen Sie der Eigenschaft *FloatingDockSiteClass* des Steuerelements den *TWinControl*-Nachkommen zu, dem das Steuerelement zugeordnet werden soll, wenn es vom Ziel einer Andock-Operation getrennt und dadurch zu einem schwebenden Fenster wird. Wenn der Benutzer das Steuerelement nicht auf dem Ziel einer Andock-Operation ablegt, wird dynamisch ein fensterorientiertes Steuerelement der angegebenen Klasse erzeugt und dem andockbaren Steuerelement als übergeordnetes Objekt zugewiesen. Wenn es sich bei dem andockbaren Steuerelement um einen Nachkommen von *TWinControl* handelt, ist die Erstellung eines separaten Ziels nicht erforderlich. Sie können jedoch ein Formular angeben, wenn Sie einen Rahmen und eine Titelleiste benötigen. Um die dynamische Erstellung eines übergeordneten Fensters zu umgehen, weisen Sie *FloatingDockSiteClass* die Klasse des Steuerelements zu. Dadurch wird dieses zu einem schwebenden Fenster, das keinem übergeordneten Objekt zugewiesen ist.

Andock-Operationen von Steuerelementen steuern

Das Ziel einer Andock-Operation akzeptiert andere Steuerelemente automatisch, wenn diese auf ihr abgelegt werden. Bei den meisten Steuerelementen wird das erste Objekt so angedockt, daß es den Client-Bereich ausfüllt, das zweite teilt diesen Bereich in Abschnitte auf usw.

Ziele von Andock-Operationen können mit Hilfe von drei Ereignissen festlegen, wie untergeordnete Steuerelemente angedockt werden.

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var   InfluenceRect:
TRect; var CanDock: Boolean) of object;
```

Zieht der Benutzer ein andockbares Objekt über ein Steuerelement, wird für das Ziel der Andock-Operation ein *OnGetSiteInfo*-Ereignis ausgelöst. Durch dieses Ereignis kann das Ziel angeben, ob es das im Parameter *DockClient* übergebene Steuerelement akzeptiert und wo sich dieses Steuerelement befinden muß, damit es für eine Andock-Operation in Frage kommt. Wenn das Ereignis *OnGetSiteInfo* eintritt, wird *InfluenceRect* mit den Bildschirmkoordinaten des Ziels und *CanDock* mit *True* initialisiert. Um den Andockbereich einzuschränken, weisen Sie *InfluenceRect* neue Werte zu. Um das Steuerelement abzulehnen, setzen Sie *CanDock* auf *False*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer; State:
TDragState; var Accept: Boolean) of object;
```

Zieht der Benutzer ein andockbares Objekt über ein Steuerelement, wird für das Ziel der Andock-Operation ein *OnDockOver*-Ereignis ausgelöst. Dies ist analog dem Ereignis *OnDragOver* bei Drag&Drop-Operationen. Über den Parameter *Accept* geben Sie an, ob das untergeordnete Objekt angedockt werden kann. Wenn das andockbare Steuerelement von der Ereignisbehandlungsroutine für *OnGetSiteInfo* nicht akzeptiert wird (weil es beispielsweise nicht der richtige Typ von Steuerelement ist), wird *OnDockOver* nicht ausgelöst.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer) of
object;
```

Wenn der Benutzer ein andockbares Objekt auf einem Ziel ablegt, wird für das Ziel ein *OnDockDrop*-Ereignis ausgelöst. Dieses Ereignis entspricht dem Ereignis *OnDragDrop* einer normalen Drag&Drop-Operation. Sie können hier Aktionen durchführen, die nach dem Ablegen des Steuerelements erforderlich sind. Der Zugriff auf das Steuerelement erfolgt über die Eigenschaft *Control* des *TDockObject*-Objekts, das durch den Parameter *Source* angegeben wird.

Steuerelemente vom Ziel einer Andock-Operation trennen

Jedes Ziel einer Andock-Operation erlaubt automatisch, daß Steuerelemente von ihm getrennt werden, wenn der Benutzer sie mit der Maus zieht (vorausgesetzt, die Eigenschaft *DragMode* dieser Steuerelemente hat den Wert *dmAutomatic*). Ein Ziel kann

aber in einer Ereignisbehandlungsroutine für *OnUnDock* auf die Trennung reagieren oder sie sogar verhindern.

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean)   of
object;
```

Der Parameter *Client* zeigt auf das Steuerelement, das versucht, den angedockten Zustand zu beenden. Der Parameter *Allow* ermöglicht dem Ziel (Parameter *Sender*), diesen Versuch zurückzuweisen. Beim Implementieren einer *OnUnDock*-Ereignisbehandlungsroutine ist vielleicht von Interesse, welche weiteren Objekte zur Zeit angedockt sind. Diese Informationen stellt die Nur-Lesen-Eigenschaft *DockClients* bereit (ein indiziertes *TControl*-Array). Die Anzahl der angedockten Objekte können Sie der Nur-Lesen-Eigenschaft *DockClientCount* entnehmen.

Die Behandlung von Drag&Dock-Operationen durch Steuerelemente festlegen

Andockbare Steuerelemente können während einer Drag&Dock-Operation zwei Ereignisse empfangen: *OnStartDock* entspricht dem Ereignis *OnStartDrag* einer Drag&Drop-Operation, indem es dem andockbaren Steuerelement ermöglicht, ein benutzerdefiniertes Drag-Objekt zu erstellen. *OnEndDock* entspricht dem Ereignis *OnEndDrag* einer Drag&Drop-Operation und tritt beim Beenden der Drag-Operation auf.

Text in Steuerelementen bearbeiten

Die folgenden Abschnitte erläutern, wie die verschiedenen Merkmale von RTF- und Memo-Steuerelementen genutzt werden können. Einige dieser Merkmale können auch mit Textkomponenten verwendet werden.

- Textausrichtung festlegen
- Bildlaufleisten zur Laufzeit hinzufügen
- Das Clipboard-Objekt hinzufügen
- Text markieren
- Den gesamten Text markieren
- Text ausschneiden, kopieren und einfügen
- Markierten Text löschen
- Menüeinträge deaktivieren
- Ein Popup-Menü bereitstellen
- Das Ereignis *OnPopup* behandeln

Die Textausrichtung festlegen

In einer RTF- oder Memokomponente kann Text linksbündig, rechtsbündig oder zentriert ausgerichtet werden. Die Textausrichtung eines Eingabefeldes legen Sie mit Hilfe der Eigenschaft *Alignment* fest. Diese Einstellung wird jedoch nur verwendet, wenn die Eigenschaft *WordWrap* (Wortumbruch) den Wert *True* hat. Bei deaktiviertem Umbruch steht kein rechter Rand für die Ausrichtung zur Verfügung.

Der folgende Code ordnet der Menüoption *Zeichen / Links* eine *OnClick*-Ereignisbehandlungsroutine zu. Dieselbe Routine wird den Optionen *Rechts* und *Zentriert* im Menü *Zeichen* zugeordnet.

```
procedure TEditForm.AlignClick(Sender: TObject);
begin
      Left1.Checked := False;  { Alle drei Markierungen entfernen }
      Right1.Checked := False;
      Center1.Checked := False;
      with Sender as TMenuItem do Checked := True;  { Das angeklickte Element markieren }
      with Editor do  { und die entsprechende Ausrichtung festlegen }
         if Left1.Checked then
         Alignment := taLeftJustify
         else if Right1.Checked then
         Alignment := taRightJustify
         else if Center1.Checked then
         Alignment := taCenter;
end;
```

Bildlaufleisten zur Laufzeit hinzufügen

RTF- und Memokomponenten können bei Bedarf horizontale und vertikale Bildlaufleisten enthalten. Ist der Wortumbruch aktiviert, wird nur eine vertikale Bildlaufleiste benötigt. Bei deaktiviertem Wortumbruch kann auch eine horizontale Bildlaufleiste angezeigt werden, da die Textzeilen nicht durch den rechten Rand des Editors begrenzt sind.

So fügen Sie zur Laufzeit Bildlaufleisten hinzu:

1. Stellen Sie fest, ob Text über den rechten Fensterrand hinausgehen kann. Prüfen Sie zu diesem Zweck, ob der Wortumbruch aktiviert ist. Sie können auch testen, ob eine Textzeile die Breite des Steuerelements überschreitet.
2. Setzen Sie die Eigenschaft *ScrollBars* auf den gewünschten Wert.

Das nachfolgend wiedergegebene Quelltextfragment ordnet der Menüoption *Zeichen / Wortumbruch* eine *OnClick*-Ereignisbehandlungsroutine zu.

```
procedure TEditForm.WordWrap1Click(Sender: TObject);
begin
      with Editor do
      begin
         WordWrap := not WordWrap;  { Wortumbruch aktivieren oder deaktivieren }
         if WordWrap then
         ScrollBars := ssVertical  { Bei Wortumbruch nur vertikale Leiste }
```

```
        else
          ScrollBars := ssBoth;  { Ohne Wortumbruch beide Leisten }
          WordWrap1.Checked := WordWrap;  { Menüoption markieren }
      end;
end;
```

RTF- und Memokomponenten behandeln Bildlaufleisten unterschiedlich. Bei RTF-Komponenten können die Bildlaufleisten ausgeblendet werden, wenn der gesamte Text innerhalb des Steuerelements angezeigt werden kann. Bei Memokomponenten werden die Bildlaufleisten immer angezeigt, wenn sie aktiviert sind.

Das Clipboard-Objekt hinzufügen

In den meisten Textverarbeitungsprogrammen können markierte Textblöcke in andere Dokumente oder Anwendungen kopiert werden. Das Clipboard-Objekt kapselt die Windows-Zwischenablage und stellt Methoden zum Ausschneiden, Kopieren und Einfügen von Text (und anderen Formaten, einschließlich Grafiken) zur Verfügung. Das Clipboard-Objekt ist in der Unit *Clipbrd* deklariert.

So fügen Sie einer Anwendung das Clipboard-Objekt *hinzu:*

1. Wählen Sie die Unit aus, in die das Objekt eingefügt werden soll.
2. Suchen Sie nach dem reservierten Wort **implementation**.
3. Fügen Sie *Clipbrd* in die **uses**-Klausel unterhalb von **implementation** ein.
   - Enthält der implementation-Abschnitt bereits eine **uses**-Klausel, fügen Sie an deren Ende *Clipbrd* hinzu.
   - Ist noch keine **uses**-Klausel vorhanden, fügen Sie die folgende Zeile hinzu:

```
uses Clipbrd;
```

In einer Anwendung mit einem untergeordneten Fenster könnte die **uses**-Klausel im **implementation**-Abschnitt der Unit beispielsweise folgendermaßen aussehen:

```
uses
     MDIFrame, Clipbrd;
```

Text markieren

Bevor Text in die Zwischenablage kopiert werden kann, muß er markiert werden. Eingabefelder heben markierten Text automatisch hervor. Text, den der Benutzer markiert, wird hervorgehoben angezeigt.

Die folgende Tabelle enthält die Eigenschaften, die zum Bearbeiten markierter Textblöcke verwendet werden können.

Tabelle 6.1 Eigenschaften markierter Textblöcke

Eigenschaft	Beschreibung
SelText	Ein String mit dem markierten Text.

Tabelle 6.1 Eigenschaften markierter Textblöcke (Fortsetzung)

Eigenschaft	Beschreibung
SelLength	Die Länge des markierten Textes.
SelStart	Der Beginn der Markierung.

Den gesamten Text markieren

Mit Hilfe der Methode *SelectAll* können Sie den gesamten Inhalt einer RTF- oder Memokomponente auswählen. Dies ist besonders hilfreich, wenn nicht der ganze Text im Steuerelement sichtbar ist. In den meisten anderen Fällen wird der Text vom Benutzer mit der Tastatur oder der Maus markiert.

Um den gesamten Inhalt einer RTF- oder Memokomponente auszuwählen, rufen Sie die Methode *SelectAll* des Steuerelements auf.

Ein Beispiel:

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
      RichEdit1.SelectAll;  { Gesamten Text der RTF-Komponente markieren }
end;
```

Text ausschneiden, kopieren und einfügen

Wenn die Unit *Clipbrd* in eine Anwendung eingebunden wird, können Text, Grafiken und Objekte mit Hilfe der Windows-Zwischenablage ausgeschnitten, kopiert und eingefügt werden. Eingabefelder kapseln die Windows-Standardkomponenten und verfügen daher über integrierte Methoden zur Interaktion mit der Zwischenablage. Informationen über die Verwendung der Zwischenablage mit Grafiken finden Sie im Abschnitt »Die Zwischenablage und Grafiken« auf Seite 7-23.

Mit den Methoden *CutToClipboard, CopyToClipboard* und *PasteFromClipboard* können Sie im Quelltext auf die Zwischenablage zugreifen.

Im folgenden Beispiel werden den *OnClick*-Ereignissen der Menübefehle *Bearbeiten / Ausschneiden, Bearbeiten / Kopieren* und *Bearbeiten / Einfügen* die entsprechenden Behandlungsroutinen zugeordnet.

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
```

Markierten Text löschen

Sie können ausgewählten Text aus einer Textkomponente löschen, ohne ihn in die Zwischenablage einzufügen. Rufen Sie zu diesem Zweck die Methode *ClearSelection* auf. Befindet sich beispielsweise die Option *Löschen* im Menü *Bearbeiten*, kann folgender Quelltext verwendet werden:

```
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
```

Menüeinträge deaktivieren

Häufig ist es sinnvoll, Menüeinträge zu deaktivieren, ohne sie aus dem Menü zu entfernen. Wenn beispielsweise in einem Texteditor kein Text markiert ist, werden die Befehle *Ausschneiden, Kopieren* und *Löschen* im Menü *Bearbeiten* grau dargestellt. Der geeignete Zeitpunkt zum Aktivieren oder Deaktivieren eines Menüeintrags ist der Moment, in dem der Benutzer das Menü öffnet.

Um einen bestimmten Menüeintrag zu deaktivieren, setzen Sie dessen Eigenschaft *Enabled* auf *False.*

Im folgenden Beispiel wird dem Ereignis *OnClick* des Menübefehls *Bearbeiten* eine Behandlungsroutine zugeordnet. Die Routine prüft, ob in der RTF-Komponente Editor Text markiert ist, und setzt die Eigenschaft *Enabled* der Befehle *Ausschneiden, Kopieren* und *Löschen*. Der Befehl *Einfügen* wird aktiviert, wenn sich Text in der Zwischenablage befindet.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
      HasSelection: Boolean;  { Temporäre Variable deklarieren }
begin
      Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);{Einfügen aktivieren oder deaktivieren}
      HasSelection := Editor.SelLength > 0;  { True, wenn Text markiert ist }
      Cut1.Enabled := HasSelection;  { Befehle aktivieren, wenn HasSelection True ist }
      Copy1.Enabled := HasSelection;
      Delete1.Enabled := HasSelection;
end;
```

Die Methode *HasFormat* des Clipboard-Objekts gibt einen Booleschen Wert zurück, der angibt, ob die Zwischenablage Objekte, Text oder Grafiken in einem bestimmten Format enthält. Durch den Aufruf von *HasFormat* mit dem Parameter CF_TEXT können Sie feststellen, ob die Zwischenablage Text enthält. Anschließend wird der Befehl *Einfügen* entsprechend aktiviert oder deaktiviert.

Informationen über die Verwendung des Clipboard-Objekts mit Grafiken finden Sie in Kapitel 7, »Mit Grafiken arbeiten«.

Ein Popup-Menü bereitstellen

Popup-Menüs (oder lokale Menüs) sind benutzerfreundliche Komponenten, die in fast jeder Anwendung verwendet werden. Mit ihnen können Benutzer an der aktuellen Mausposition durch Klicken mit der rechten Maustaste eine Liste gängiger Befehle anzeigen.

In einem Texteditor wäre beispielsweise ein Popup-Menü mit den Befehlen *Ausschneiden*, *Kopieren* und *Einfügen* des Menüs *Bearbeiten* sinnvoll. Für diese Optionen können dieselben Ereignisbehandlungsroutinen wie für die entsprechenden Einträge im Menü *Bearbeiten* verwendet werden. Zugriffstasten oder Tastenkürzel brauchen Sie für Popup-Menüs nicht zu definieren.

Mit der Formulareigenschaft *PopupMenu* legen Sie fest, welches Popup-Menü beim Klicken mit der rechten Maustaste angezeigt wird. Da auch die einzelnen Steuerelemente über diese Eigenschaft verfügen, können Sie die Einstellung des Formulars außer Kraft setzen, um für bestimmte Komponenten ein anderes Menü anzuzeigen.

So fügen Sie einem Formular ein Popup-Menü hinzu:

1. Plazieren Sie eine *PopupMenu*-Komponente im Formular.
2. Definieren Sie die Einträge des Popup-Menüs im Menü-Designer.
3. Weisen Sie der Eigenschaft *PopupMenu* des Formulars oder eines Steuerelements das zuvor definiertet Popup-Menü zu.
4. Weisen Sie dem *OnClick*-Ereignis der verschiedenen Menüoptionen die entsprechenden Behandlungsroutinen zu.

Das Ereignis OnPopup

In manchen Situationen müssen die Einträge eines Popup-Menüs angepaßt werden, bevor das Menü eingeblendet wird (wie beim Aktivieren oder Deaktivieren normaler Menüoptionen). Bei einem normalen Menü erreichen Sie dies mit Hilfe des Ereignisses *OnClick* des zugehörigen Hauptmenüeintrags (siehe »Menüeinträge deaktivieren« auf Seite 6-11).

Da ein Popup-Menü jedoch keine übergeordnete Menüleiste hat, muß das Ereignis in der Menükomponente selbst behandelt werden. Genau für diesen Zweck verfügt die Komponente über das Ereignis *OnPopup*.

So passen Sie die Optionen eines Popup-Menüs an, bevor es angezeigt wird:

1. Wählen Sie ein Popup-Menü aus.
2. Weisen Sie seinem Ereignis *OnPopup* eine Behandlungsroutine zu.
3. In der Routine können Sie die gewünschten Menüoptionen aktivieren, deaktivieren, ausblenden oder einblenden.

Im folgenden Beispiel wird die im Abschnitt »Menüeinträge deaktivieren« auf Seite 6-11 beschriebene Ereignisbehandlungsroutine *Edit1Click* dem Ereignis

OnPopup eines Popup-Menüs zugeordnet. Für jede Menüoption kommt eine weitere Programmzeile hinzu.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
      HasSelection: Boolean;
begin
      Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
      Paste2.Enabled := Paste1.Enabled; {Neue Zeile}
      HasSelection := Editor.SelLength <> 0;
      Cut1.Enabled := HasSelection;
      Cut2.Enabled := HasSelection;  {Neue Zeile}
      Copy1.Enabled := HasSelection;
      Copy2.Enabled := HasSelection; {Neue Zeile}
      Delete1.Enabled := HasSelection;
end;
```

Grafiken zu Steuerelementen hinzufügen

Die Windows-Steuerelemente für Listenfelder, Kombinationsfelder und Menüs unterstützen den Stil *Owner-Draw*. Bei dieser Einstellung wird zum Anzeigen der Elemente nicht die Windows-Standardmethode verwendet. Die Elemente werden statt dessen zur Laufzeit vom übergeordneten Objekt der Komponente (normalerweise das Formular) gezeichnet. Dieser Stil wird verwendet, um Grafiken anstelle von Text oder zusätzlich zu Text anzuzeigen. Weitere Informationen zu diesem Thema finden Sie unter »Grafiken zu Menüeinträgen hinzufügen« auf Seite 5-23.

Alle Owner-Draw-Steuerelemente enthalten Listen mit Elementen. Standardmäßig enthalten die Listen Strings, die von Windows als Text angezeigt werden. Sie können den Listenelementen aber auch Objekte zuordnen.

Im allgemeinen werden Owner-Draw-Steuerelemente in Delphi folgendermaßen erstellt:

1. Den Owner-Draw-Stil festlegen
2. Grafikobjekte zu einer Stringliste hinzufügen
3. Owner-Draw-Elemente anzeigen

Den Owner-Draw-Stil festlegen

Listen- und Kombinationsfelder verfügen über die Eigenschaft *Style*, die festlegt, ob das Steuerelement standardmäßig (Stil *Standard*) oder vom Eigentümer gezeichnet wird. Bei Datengittern kann das Standardverfahren mit Hilfe der Eigenschaft *DefaultDrawing* aktiviert oder deaktiviert werden.

Listen- und Kombinationsfelder unterstützen zwei weitere Owner-Draw-Werte: *fixed* und *variable* (siehe die nachfolgende Tabelle). Owner-Draw-Datengitter sind immer

fixed. Obwohl jede Zeile oder Spalte eine eigene Größe haben kann, wird die Zellengröße vor dem Zeichnen des Gitters bestimmt.

Tabelle 6.2 Fester und variabler Owner-Draw-Stil

Owner-Draw-Stil	Beschreibung	Beispiele
Fixed	Alle Elemente erhalten die mit der Eigenschaft *ItemHeight* festgelegte Größe.	LbOwnerDrawFixed, csOwnerDrawFixed
Variable	Die Höhe jedes Elements wird zur Laufzeit an die anzuzeigenden Daten angepaßt.	LbOwnerDrawVariable, csOwnerDrawVariable

Grafikobjekte zu einer Stringliste hinzufügen

Jeder Stringliste kann zusätzlich zu den Strings eine Liste von Objekten gespeichert werden.

In einer Dateiverwaltung können beispielsweise zusammen mit den Laufwerksbuchstaben Grafiken angezeigt werden, die den Typ des Laufwerks angeben. Dazu müssen Sie der Anwendung die Grafiken hinzufügen und sie anschließend an die entsprechende Position in der Stringliste kopieren. Diese Operationen werden in den nächsten Abschnitten beschrieben.

Grafiken zu einer Anwendung hinzufügen

Ein Image-Steuerelement ist ein nichtvisuelles Steuerelement, mit dem Grafiken (z. B. Bitmaps) in einem Formular angezeigt werden können. Sie können mit diesen Komponenten aber auch verborgene Grafiken speichern, die Sie in Ihrer Anwendung verwenden wollen. Um beispielsweise Bitmaps für Owner-Draw-Steuerelemente in verborgenen Image-Steuerelementen zu speichern, gehen Sie folgendermaßen vor:

1. Plazieren Sie die gewünschten Image-Steuerelemente im Hauptformular.
2. Weisen Sie der Eigenschaft *Name* der Steuerelemente einen Wert zu.
3. Setzen Sie die Eigenschaft *Visible* der Steuerelemente auf *False*.
4. Weisen Sie der Eigenschaft *Picture* jedes Steuerelements die gewünschte Bitmap-Grafik zu. Verwenden Sie dazu den Grafikeditor des Objektinspektors.

Die Image-Steuerelemente werden zur Laufzeit nicht angezeigt.

Grafiken zu einer Stringliste hinzufügen

Nachdem Sie Ihrer Anwendung Grafiken hinzugefügt haben, können Sie diese den Strings einer Stringliste zuordnen. Die Grafikobjekte können entweder zusammen mit den Strings eingefügt oder vorhandenen Strings zugeordnet werden. Wenn alle benötigten Daten verfügbar sind, sollten Objekte und Strings immer gemeinsam bearbeitet werden.

Das folgende Beispiel zeigt, wie Grafiken in einer Stringliste plaziert werden. Der Quelltext ist Teil einer Dateiverwaltung, in der neben den Laufwerksbuchstaben Bit-

maps für den Typ des Laufwerks angezeigt werden. Die Ereignisbehandlungsroutine für *OnCreate* sieht folgendermaßen aus:

```
procedure TFMForm.FormCreate(Sender: TObject);
var
      Drive: Char;
      AddedIndex: Integer;
begin
      for Drive := 'A' to 'Z' do  { Alle möglichen Laufwerke untersuchen }
      begin
         case GetDriveType(Drive + ':/') of  { Positive Werte bedeuten gültige Laufwerke }
         DRIVE_REMOVABLE:  { Register hinzufügen }
            AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
         DRIVE_FIXED:  { Register hinzufügen }
            AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
         DRIVE_REMOTE:  { Register hinzufügen }
            AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
         end;
         if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then  { Aktuelles Laufwerk? }
         DriveTabSet.TabIndex := AddedIndex;  { Register aktivieren}
      end;
end;
```

Owner-Draw-Elemente anzeigen

Steuerelemente mit dem Stil Owner-Draw werden von Windows nicht direkt auf dem Bildschirm angezeigt. Statt dessen werden für jedes sichtbare Element der Komponente Ereignisse generiert, die Sie in Ihrer Anwendung behandeln können, um die Elemente anzuzeigen.

Führen Sie für jedes sichtbare Element in einem Owner-Draw-Steuerelement die folgenden Aktionen durch. Verwenden Sie für alle Elemente eine einzige Ereignisbehandlungsroutine.

1 Legen Sie bei Bedarf die Größe des Elements fest.

   Wenn alle Elemente die gleiche Größe haben (z. B. bei einem Listenfeld mit dem Stil *lsOwnerDrawFixed*), können Sie diesen Schritt überspringen.

2 Zeichnen Sie das Element.

Größe von Owner-Draw-Elementen festlegen

Bevor Ihre Anwendung die Elemente eines variablen Owner-Draw-Steuerelements zeichnen kann, generiert Windows das Ereignis *OnMeasureItem*, das der Anwendung mitteilt, wo das Element im Steuerelement angezeigt wird.

Windows ermittelt die Größe des Elements (im allgemeinen ist es gerade groß genug, um den Text des Elements in der aktuellen Schrift anzuzeigen). Sie können das Ereignis in Ihrer Anwendung behandeln und das von Windows übergebene Rechteck ändern. Wenn Sie beispielsweise den Text eines Elements durch ein Bitmap ersetzen wollen, passen Sie das Rechteck an die Größe der Grafik an. Sollen Grafik und Text

zusammen angezeigt werden, ändern Sie das Rechteck entsprechend der Größe der beiden Elemente.

Sie können die Größe eines Owner-Draw-Elements ändern, indem Sie eine Ereignisbehandlungsroutine für *OnMeasureItem* bereitstellen. Je nach Steuerelement ist der Name des Ereignisses unterschiedlich. Bei Listen- und Kombinationsfeldern heißt es *OnMeasureItem*. Bei Datengittern gibt es kein entsprechendes Ereignis.

Der Ereignisbehandlungsroutine für *OnMeasureItem* müssen zwei wichtige Parameter übergeben werden: der Index und die Größe des Elements. Die Größe ist variabel und kann in der Anwendung geändert werden. Die Positionen der nachfolgenden Elemente sind von der Größe der vorhergehenden Elemente abhängig.

Wird beispielsweise bei einem Owner-Draw-Listenfeld die Höhe des ersten Elements auf fünf Pixel gesetzt, beginnt das zweite Element mit dem sechsten Pixel von oben usw. Bei Listen- und Kombinationsfeldern kann nur die Höhe der Elemente geändert werden. Die Breite entspricht immer der Breite des Steuerelements.

Die Zellengröße von Owner-Draw-Datengittern kann beim Zeichnen nicht geändert werden. Sie müssen die Größe der einzelnen Zeilen und Spalten mit den Eigenschaften *ColWidths* und *RowHeights* festlegen.

In der folgenden Behandlungsroutine für Ereignis *OnMeasureItem* eines Owner-Draw-Listenfeldes wird die Höhe der einzelnen Elemente entsprechend der Größe des zugehörigen Bitmaps geändert:

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
      var TabWidth: Integer);  { Beachten Sie, daß TabWidth ein var-Parameter ist }
var
      BitmapWidth: Integer;
begin
      BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
      { Breite um die Breite des Bitmaps + 2 erhöhen }
      Inc(TabWidth, 2 + BitmapWidth);
end;
```

Hinweis Mit den in der Eigenschaft *Object*s gespeicherten Elementen müssen Sie eine Typumwandlung durchführen. Da die Eigenschaft *Objects* vom Typ *TObject* ist, kann sie jede Art von Objekten aufnehmen. Wenn Sie Objekte aus dem Array abrufen, müssen Sie diese wieder in den ursprünglichen Elementtyp konvertieren.

Alle Owner-Draw-Elemente anzeigen

Wenn in einer Anwendung ein Owner-Draw-Steuerelement angezeigt oder aktualisiert werden muß, generiert Windows für jedes sichtbare Element der Komponente ein entsprechendes Ereignis.

Wenn Sie alle Elemente des Steuerelements anzeigen wollen, erstellen Sie eine Behandlungsroutine für dieses Ereignis.

Die Namen der Ereignisse beginnen immer mit *OnDraw* (z. B. *OnDrawItem* oder *OnDrawCell*).

Einer Behandlungsroutine für ein *OnDraw*-Ereignis müssen drei Parameter übergeben werden: der Index des anzuzeigenden Elements, das Rechteck, in dem es angezeigt wird, und der Elementstatus (z. B. ob das Element den Fokus hat). Die Anwendung gibt das Element im angegebenen Rechteck aus.

Im folgenden Beispiel werden Elemente in einer Liste angezeigt, in der jedem String ein Bitmap zugeordnet ist. Die Routine wird dem Ereignis *OnDrawItem* des Listenfeldes zugeordnet:

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
      R: TRect; Index: Integer; Selected: Boolean);
var
      Bitmap: TBitmap;
begin
      Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
      with TabCanvas do
      begin
      Draw(R.Left, R.Top + 4, Bitmap);  { Bitmap anzeigen }
      TextOut(R.Left + 2 + Bitmap.Width,  { Text positionieren }
         R.Top + 2, DriveTabSet.Tabs[Index]);  { und rechts neben dem Bitmap ausgeben }
      end;
end;
```

Kapitel

7

Mit Grafiken und Multimedia arbeiten

Mit Grafiken und Multimedia-Elementen kann das Erscheinungsbild von Anwendungen optimiert werden. Delphi stellt verschiedene Möglichkeiten zur Implementierung dieser Merkmale in Anwendungen bereit. Ein grafisches Element können Sie beispielsweise während des Entwurfs als gezeichnetes Bild einfügen oder mit grafischen Steuerelementen erstellen. Alternativ können diese Elemente zur Laufzeit dynamisch gezeichnet werden. Außerdem enthält Delphi eine Reihe spezieller Komponenten zum Abspielen von Audio- und Videoclips, die der Implementierung von Multimedia-Merkmalen dienen.

Grafikprogrammierung im Überblick

In den Grafikkomponenten der VCL ist die Windows-GDI (Graphics Device Interface) gekapselt, wodurch die Grafikprogrammierung unter Windows wesentlich vereinfacht wird.

Wenn Sie in einer Delphi-Anwendung Grafikelemente zeichnen, erfolgt dies nicht direkt auf dem betreffenden Objekt, sondern auf dessen Zeichenfläche (*Canvas*). Die Zeichenfläche ist eine Eigenschaft des Objekts, stellt aber ihrerseits selbst ein Objekt dar. Ein Hauptvorteil des Objekts *Canvas* liegt im effizienten Umgang mit Ressourcen. Außerdem übernimmt es die Verwaltung des Gerätekontexts, so daß Ihre Programme immer dieselben Methoden verwenden können, unabhängig davon, ob Sie auf dem Bildschirm oder auf dem Drucker zeichnen, oder ob Sie Bitmaps oder Metadateien bearbeiten. Da Zeichenflächen nur zur Laufzeit verfügbar sind, erfolgt die Arbeit mit diesen Objekten ausschließlich über den Quelltext.

Hinweis Da *TCanvas* den Windows-Gerätekontext als Ressourcen-Manager kapselt, können auf der Zeichenfläche alle Windows-GDI-Funktionen eingesetzt werden. Der Wert der Eigenschaft *Handle* der Zeichenfläche ist mit dem Gerätekontext identisch.

Wie Grafiken in einer Anwendung dargestellt werden, hängt vom Typ des Objekts ab, auf dessen Zeichenfläche gezeichnet wird. Wenn Sie direkt auf die Zeichenfläche eines Steuerelements zeichnen, wird das Bild sofort angezeigt. Dagegen wird ein

Bild, das nicht auf dem Bildschirm gezeichnet wird (sondern z. B. auf einer *TBitmap*-Zeichenfläche), erst angezeigt, wenn ein Steuerelement das Bitmap in die Zeichenfläche des Steuerelements kopiert. Bitmaps, die nach dem Zeichnen einem Bild-Steuerelement zugewiesen wurden, erscheinen somit erst auf dem Bildschirm, nachdem das Steuerelement seine OnPaint-Botschaft verarbeitet hat.

Wenn Sie mit Grafikelementen arbeiten, begegnen Ihnen häufig die Begriffe *Zeichnen* und *Bild aufbauen*:

- *Zeichnen* ist die Erzeugung eines spezifischen Grafikelements, beispielsweise einer Linie oder einer Figur, durch Quelltext. Über den Quelltext weisen Sie ein Objekt an, ein bestimmtes Grafikelement an einer bestimmten Stelle seiner Zeichenfläche zu plazieren, indem eine Zeichenmethode der Zeichenfläche aufgerufen wird.
- *Bild aufbauen* beschreibt die Erzeugung des Erscheinungsbildes eines Objekts. In der Regel ist Bild aufbauen mit Zeichnen verbunden. Das heißt, daß ein Objekt auf *OnPaint*-Ereignisse meistens mit dem Zeichnen von Grafikelementen reagiert. Ein Eingabefeld zeichnet sich zum Beispiel selbst, indem es ein Rechteck zeichnet und darin den Text ausgibt. Ein Shape-Steuerelement zeichnet sich hingegen in Form eines einzigen Grafikelements.

Die Beispiele am Anfang dieses Kapitels erläutern, wie verschiedene Grafikelemente gezeichnet werden. Dies geschieht jedoch immer als Reaktion auf *OnPaint*-Ereignisse. Spätere Abschnitte zeigen, wie die gleichen Zeichenvorgänge als Reaktion auf andere Ereignisse durchgeführt werden können.

Den Bildschirm aktualisieren

Windows ermittelt gelegentlich, daß Bildschirmobjekte aktualisiert werden müssen, und generiert zu diesem Zweck *WM_PAINT*-Botschaften, die von der VCL an *OnPaint*-Ereignisse weitergeleitet werden. Wenn Sie die Methode *Refresh* verwenden, ruft die VCL die *OnPaint*-Ereignisbehandlungsroutine auf, die Sie für das betreffende Objekt geschrieben haben. Standardmäßig erhält die OnPaint-Ereignisbehandlungsroutine in einem Formular den Namen *FormPaint*. Sie können eine Komponente oder ein Formular mit der Methode *Refresh* aktualisieren. So können Sie beispielsweise *Refresh* in der Ereignisbehandlungsroutine für *OnResize* des Formulars aufrufen, um Grafiken neu anzuzeigen oder in einem Formular einen Hintergrund zu zeichnen.

Unter Windows erfolgt das Neuzeichnen des Client-Bereichs eines Fensters, das aktualisiert werden muß, nicht automatisch, wie dies bei einigen anderen Betriebssystemen der Fall ist. Was unter Windows auf den Bildschirm gezeichnet wird, ist permanent. Wird ein Formular oder Steuerelement vorübergehend verdeckt, beispielsweise beim Verschieben eines Fensters, muß das Formular oder Steuerelement den verdeckten Bereich neu zeichnen, wenn er wieder sichtbar wird. Weitere Informationen zur Botschaft *WM_PAINT* finden Sie in der Online-Hilfe zu Windows.

Wenn Sie das Steuerelement *TImage* verwenden, führt die VCL das Zeichnen und Aktualisieren der in *TImage* enthaltenen Grafik automatisch durch. Da beim Zeichnen auf *TImage*-Objekt ein persistentes Bild erzeugt wird, müssen Sie nichts unternehmen, um das enthaltene Bild neu zu zeichnen. Im Gegensatz hierzu ist die Zeichenfläche von *TPaintBox* direkt dem Bildschirmgerät zugeordnet, so daß alles, was auf

die Zeichenfläche von *PaintBox* gezeichnet wird, nicht persistent ist. Dies trifft für nahezu alle Steuerelemente zu, auch für das Formular selbst. Deshalb müssen Sie, wenn Sie mit dem Konstruktor von *TPaintBox* zeichnen oder das Bild aufbauen, diesen Quelltext auch in die *OnPaint*-Ereignisbehandlungsroutine einfügen, damit das Bild neu gezeichnet wird, wenn der Client-Bereich aktualisiert werden muß.

Grafikobjekt-Typen

Die VCL unterstützt die in Tabelle 7.3 gezeigten Grafikobjekte. Diese Objekte verfügen über Methoden zum Zeichnen auf der Zeichenfläche sowie zum Laden und Speichern von Grafikdateien. Erstere werden im Abschnitt »Grafikobjekte zeichnen« auf Seite 7-10, letztere unter »Grafikdateien laden und speichern« auf Seite 7-20 beschrieben.

Tabelle 7.1 Grafikobjekt-Typen

Objekt	Beschreibung
Picture	Dieser Objekttyp kann jedes beliebige Grafikbild aufnehmen. Mit der Methode *Register* können Sie weitere Grafikdateiformate hinzufügen. Setzen Sie diesen Objekttyp zur universellen Behandlung von Dateien ein, z. B. zum Anzeigen in einem Bild-Steuerelement.
Bitmap	Dieser leistungsfähige Objekttyp kann zum Erstellen, Manipulieren (Skalieren, Bildlauf, Rotieren und Bildaufbau) und Speichern von Bildern als Dateien auf einem Datenträger eingesetzt werden. Mit diesem Typ lassen sich einfach und schnell Kopien von einem Bitmap erstellen, da nicht das Bild selbst, sondern nur das *Handle* kopiert wird.
Clipboard	Dieser Objekttyp bildet den Container für jedes Text- oder Grafikelement, das in einer Anwendung ausgeschnitten oder kopiert bzw. eingefügt wird. Verwenden Sie diesen Objekttyp, um Daten entsprechend dem jeweiligen Format abzurufen, Referenzzähler und das Öffnen und Schließen der Zwischenablage zu verwalten und Formate für Objekte in der Zwischenablage zu ändern und zu verwalten.
Icon	Dieses Grafikobjekt repräsentiert den Wert, der aus einer Windows-Symboldatei (ICO) geladen wurde.
Metafile	Dieser Objekttyp enthält eine Metadatei, die die erforderlichen Operationen zum Aufbau eines Bildes enthält, nicht jedoch die tatsächlichen Bitmap-Pixel des Bildes. Metadateien bieten Vorteile hinsichtlich der Skalierbarkeit, ohne die Detaildarstellung des Bildes negativ zu beeinflussen. Häufig benötigen sie wesentlich weniger Speicherplatz als Bitmaps, insbesondere für hochauflösende Geräte wie z. B. Drucker. Metadateien werden jedoch langsamer als Bitmaps gezeichnet. Metadateien sollten eingesetzt werden, wenn Vielseitigkeit wichtiger ist als die Ausführungsgeschwindigkeit.

Häufig verwendete Eigenschaften und Methoden des Objekts Canvas

In der folgenden Tabelle sind die gebräuchlichsten Eigenschaften des Objekts *Canvas* aufgeführt. Eine vollständige Liste der Eigenschaften und Methoden finden Sie bei der Beschreibung der Komponente *TCanvas* in der Online-Hilfe.

Tabelle 7.2 Häufig verwendete Eigenschaften des Objekts Canvas

Eigenschaft	Beschreibung
Font	Legt die Schrift fest, die beim Schreiben von Text in das Bild verwendet wird. Über die Eigenschaften des Objekts *TFont* können die Schriftart, die Farbe, die Größe und die Auszeichnung für die Schrift bestimmt werden.
Brush	Legt die Farbe und das Muster fest, das auf der Zeichenfläche zum Füllen von grafischen Formen und von Hintergründen verwendet wird. Über die Eigenschaften des Objekts *TBrush* können Sie die Farbe und das Muster bzw. das Bitmap bestimmen, das beim Füllen von Leerraum in der Zeichenfläche benutzt wird.
Pen	Legt die Art des Stiftes fest, der zum Zeichnen von Linien und Umrissen in der Zeichenfläche benutzt wird. Über die Eigenschaften des Objekts *TPen* können Sie Farbe, Stil, Strichstärke und Modus für den Stift bestimmen.
PenPos	Legt die aktuelle Zeichenposition des Stiftes fest.
Pixels	Legt die Pixelfarbe innerhalb des aktuellen Clipping-Rechtecks *ClipRect* fest.

Diese Eigenschaften werden unter »Häufig verwendete Eigenschaften und Methoden des Objekts Canvas« auf Seite 7-4 ausführlicher beschrieben.

In der folgenden Tabelle sind die gebräuchlichsten Methoden des Objekts *Canvas* aufgeführt:

Tabelle 7.3 Häufig verwendete Methoden des Objekts Canvas

Methode	Beschreibung
Arc	Zeichnet einen Bogen am Umfang einer Ellipse, die durch das angegebene Rechteck begrenzt ist.
Chord	Zeichnet eine geschlossene Form, die sich aus dem Schnitt einer Linie mit einer Ellipse ergibt.
CopyRect	Kopiert einen Teil eines Bildes aus einer anderen Zeichenfläche in die aktuelle Zeichenfläche.
Draw	Zeichnet die im Parameter *Graphic* angegebene Grafik an den mit (*X, Y*) festgelegten Koordinaten auf der Zeichenfläche.
Ellipse	Zeichnet die durch ein umschließendes Rechteck festgelegte Ellipse auf der Zeichenfläche.
FillRect	Füllt das angegebene Rechteck auf der Zeichenfläche unter Verwendung des aktuellen Pinsels.
FloodFill	Füllt einen Bereich der Zeichenfläche unter Verwendung des aktuellen Pinsels.
FrameRect	Zeichnet ein Rechteck und verwendet dabei für den Rahmen den Pinsel der Zeichenfläche.

Tabelle 7.3 Häufig verwendete Methoden des Objekts Canvas (Fortsetzung)

Methode	Beschreibung
LineTo	Zeichnet auf der Zeichenfläche eine Linie von der Stiftposition (*PenPos*) bis zu dem in *X* und *Y* angegebenen Punkt und setzt die neue Stiftposition auf (*X, Y*).
MoveTo	Setzt die aktuelle Zeichenposition auf den Punkt (*X,Y*).
Pie	Zeichnet auf der Zeichenfläche ein tortenstückförmiges Segment der von dem Rechteck (*X1, Y1*) und (*X2, Y2*) umgebenen Ellipse.
Polygon	Zeichnet eine Folge von Linien auf der Zeichenfläche. Dazu werden die in *Points* übergebenen Punkte durch Linien miteinander verbunden. Zuletzt wird die Form durch eine Linie zwischen dem letzten und dem ersten Punkt geschlossen.
PolyLine	Zeichnet unter Verwendung des aktuellen Stiftes eine Folge von Linien auf der Zeichenfläche. Dazu werden die in *Points* übergebenen Punkte miteinander verbunden.
Rectangle	Zeichnet ein Rechteck auf der Zeichenfläche. Die obere linke Ecke des Rechtecks wird in (*X1, Y1*), die untere rechte Ecke in (*X2, Y2*) angegeben. Verwenden Sie *Rectangle*, um ein Rechteck mit dem in *Pen* festgelegten Stift und der in *Brush* angegebenen Füllung zu erstellen.
RoundRect	Zeichnet ein Rechteck mit abgerundeten Ecken auf der Zeichenfläche.
StretchDraw	Zeichnet eine Grafik so auf der Zeichenfläche, daß das Bild genau in das angegebene Rechteck paßt. Gegebenenfalls werden die Größe oder das Seitenverhältnis des Grafikbildes entsprechend angepaßt.
TextHeight, TextWidth	Ermittelt die Höhe bzw. die Breite eines Strings in der aktuellen Schrift. Bei der Höhe wird der Abstand zwischen den Zeilen berücksichtigt.
TextOut	Schreibt einen String beginnend beim Punkt (*X,Y*) auf die Zeichenfläche und setzt die neue Stiftposition auf das Ende des Strings.
TextRect	Schreibt einen String in einen angegebenen Bereich. Teile des Strings, die außerhalb des Bereichs liegen, werden nicht angezeigt.

Diese Methoden werden unter »Grafikobjekte zeichnen« auf Seite 7-10 ausführlicher beschrieben.

Eigenschaften des Objekts Canvas verwenden

Über die Eigenschaften des Objekts *Canvas* können Sie den Stift zum Zeichnen von Linien, den Pinsel zum Füllen von Formen, eine Schrift zur Ausgabe von Text und die Pixel zur Darstellung des Bildes festlegen.

In diesem Abschnitt werden folgende Themen behandelt:

- Stifte verwenden
- Pinsel verwenden
- Pixel lesen und setzen

Stifte verwenden

Die Eigenschaft *Pen* einer Zeichenfläche steuert die Darstellung von Linien. Dies schließt auch Linien ein, die als Umrißlinien von Formen gezeichnet werden. Das

Zeichnen einer geraden Linie bedeutet eigentlich nur, daß eine Reihe von Pixeln, die zwischen zwei Punkten liegen, verändert wird.

Der Stift selbst hat vier Eigenschaften, die Sie ändern können: *Color, Width, Style* und *Mode.*

- *Color* (Eigenschaft): ändert die Stiftfarbe.
- *Width* (Eigenschaft): ändert die Strichstärke von Stiften.
- *Style* (Eigenschaft): ändert den Stiftstil.
- *Mode* (Eigenschaft): ändert den Stiftmodus.

Die Werte dieser Eigenschaften bestimmen, wie der Stift die Pixel in der Linie verändert. Per Voreinstellung ist jeder Stift schwarz, die Linienstärke beträgt 1 Pixel, der Stil ist eine durchgehende Linie, und der Modus »Kopieren« ist aktiviert. Damit wird alles überschrieben, was sich auf der Zeichenfläche befindet.

Stiftfarben ändern

Die Farbe eines Stiftes kann wie jede andere Farbeigenschaft (*Color*) zur Laufzeit gesetzt werden. Die Stiftfarbe bestimmt die Farbe, in der der Stift Linien zeichnen soll. Dazu gehören sowohl Begrenzungslinien von Formen als auch normale Linien und Polygonzüge. Sie ändern die Stiftfarbe, indem Sie der Eigenschaft *Color* des Stiftes einen Wert zuweisen.

Damit der Anwender eine neue Farbe für den Stift auswählen kann, müssen Sie der Werkzeugleiste des Stiftes ein Farbgitter hinzufügen. Ein Farbgitter kann sowohl Vordergrund- als auch Hintergrundfarben setzen. Für einen Stiftstil ohne Gitter ist die Hintergrundfarbe zu berücksichtigen, die in den Lücken zwischen den Liniensegmenten gezeichnet wird. Die Hintergrundfarbe wird über die Farbeigenschaft des Pinsels festgelegt.

Da der Anwender eine neue Farbe auswählt, indem er auf das Farbgitter klickt, weist der folgende Quelltext dem Stift als Reaktion auf ein *OnClick*-Ereignis eine neue Farbe zu:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

Strichstärke von Stiften ändern

Die Strichstärke in Pixel bestimmt die Dicke der Linien, die der Stift zeichnet.

Hinweis Unter Windows 95 werden breitere Strichstärken als 1 immer als durchgezogene Linien gezeichnet, ganz gleich, welchen Wert die Eigenschaft *Style* hat.

Um die Strichstärke des Stiftes zu ändern, weisen Sie seiner Eigenschaft *Width* einen numerischen Wert zu.

Angenommen, Sie haben zum Setzen der Strichstärke eine Bildlaufleiste auf der Werkzeugleiste des Stiftes plaziert. Außerdem möchten Sie eine Beschriftung aktualisieren, die sich neben der Bildlaufleiste befindet, so daß der Anwender eine Rück-

meldung über die aktuelle Strichstärke erhält. Wenn Sie die Einstellung der Bildlaufleiste verwenden, um die Strichstärke des Stiftes festzulegen, müssen Sie die Strichstärke jedesmal aktualisieren, wenn sich die Einstellung ändert.

Die Behandlungsroutine für das Ereignis *OnChange* der Bildlaufleiste könnte folgendermaßen aussehen:

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;{ Strichstärke direkt festlegen }
 PenSize.Caption := IntToStr(PenWidth.Position);{ Für die Beschriftung in String umwandeln }
end;
```

Stiftstile ändern

Die Eigenschaft *Style* eines Stiftes bestimmt, ob durchgehende, strichlierte oder punktierte Linien usw. gezeichnet werden.

Hinweis Windows 95 unterstützt keine gestrichelten oder punktierten Linien für Stifte mit einer Strichstärke, die breiter als 1 Pixel ist. In diesem Fall wird, unabhängig vom Wert der Eigenschaft *Style*, immer eine durchgezogene Linie gezeichnet.

Beim Festlegen der Stifteigenschaften bietet es sich an, zur Behandlung der Ereignisse der verschiedenen Steuerelemente eine gemeinsame Ereignisbehandlungsroutine einzusetzen. Durch Überprüfung des Parameters *Sender* kann ermittelt werden, welches Steuerelement das Ereignis erhalten hat.

Gehen Sie beispielsweise folgendermaßen vor, um eine *OnClick*-Ereignisbehandlungsroutine für sechs Stiftstil-Schaltflächen auf der Werkzeugleiste eines Stiftes zu erzeugen:

1. Markieren Sie alle sechs Stiftstil-Schaltflächen, und wählen Sie im Objektinspektor in der Registerkarte *Ereignisse* das Ereignis *OnClick*. Geben Sie anschließend in das Feld daneben `SetPenStyle` ein.

   Delphi erzeugt eine leere *OnClick*-Ereignisbehandlungsroutine namens *SetPenStyle* und ordnet sie dem Ereignis *OnClick* aller sechs Schaltflächen zu.

2. Ergänzen Sie die Ereignisbehandlungsroutine um Anweisungen, die den Schriftstil zuweisen. Jede Zuweisung hängt vom Wert des Parameters *Sender* ab, der das Steuerelement bezeichnet, von dem das Ereignis ausgelöst wurde:

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
   else if Sender = DashPen then Style := psDash
   else if Sender = DotPen then Style := psDot
   else if Sender = DashDotPen then Style := psDashDot
   else if Sender = DashDotDotPen then Style := psDashDotDot
   else if Sender = ClearPen then Style := psClear;
  end;
end;
```

Den Stiftmodus ändern

Über die Eigenschaft *Mode* eines Stiftes legen Sie fest, wie die Stiftfarbe mit der Farbe auf der Zeichenfläche kombiniert werden soll. So könnte z. B. immer Schwarz als Stiftfarbe festgelegt werden. Es wäre aber auch möglich, die Komplementärfarbe zur Hintergrundfarbe der Zeichenfläche oder der Stiftfarbe zu verwenden usw. Einzelheiten finden Sie bei der Beschreibung von *TPen* in der Online-Hilfe.

Stiftposition ermitteln

Die aktuelle Zeichenposition (also die Position, an der die nächste Zeichenoperation beginnt) wird als Stiftposition bezeichnet. Die Zeichenfläche speichert ihre Stiftposition in einer Eigenschaft namens *PenPos* . Die Stiftposition wirkt sich nur auf das Zeichnen von Linien aus. Bei Formen und Text müssen Sie alle erforderlichen Koordinaten angeben.

Um die Stiftposition festzulegen, rufen Sie die Methode *MoveTo* der Zeichenfläche auf. Mit der folgenden Anweisung können Sie die Stiftposition beispielsweise in die obere linke Ecke der Zeichenfläche bewegen:

```
Canvas.MoveTo(0, 0);
```

Hinweis Das Zeichnen einer Linie mit der Methode *LineTo* verschiebt außerdem die aktuelle Position zum Endpunkt der Linie.

Pinsel verwenden

Die Eigenschaft *Brush* einer Zeichenfläche gibt an, wie Bereiche gefüllt werden. Dies schließt den Füllbereich von Formen ein. Wenn Sie einen Bereich mit Hilfe eines Pinsels füllen, wird eine große Anzahl benachbarter Pixel auf eine bestimmte Art verändert.

Der Pinsel hat drei Eigenschaften, die geändert werden können:

- *Color* (Eigenschaft): ändert die Füllfarbe
- *Style* (Eigenschaft): ändert den Pinselstil
- *Bitmap* (Eigenschaft): definiert ein Bitmap als Pinselmuster

Die Werte dieser Eigenschaften legen fest, wie die Zeichenfläche Formen oder andere Bereiche ausfüllt. Per Voreinstellung ist jeder Pinsel weiß, der Stil ist »gefüllt«, und es wird kein Bitmap als Muster verwendet.

Pinselfarben ändern

Die Farbe eines Pinsels legt fest, mit welcher Farbe die Zeichenfläche die Formen ausfüllt. Um die Füllfarbe zu ändern, weisen Sie der Eigenschaft *Color* des Pinsels einen Wert zu. Ein Pinsel stellt die Hintergrundfarbe für Text und Linienzeichnungen dar. Sie werden deshalb normalerweise die Hintergrundfarbe als Pinselfarbe verwenden.

Wie die Stiftfarbe kann auch die Pinselfarbe als Antwort auf einen Mausklick gesetzt werden, der auf einem Farbgitter in der Werkzeugleiste des Pinsels ausgeführt wird (siehe »Stiftfarben ändern« auf Seite 7-6):

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

Pinselstile ändern

Der Pinselstil legtfest, mit welchen Mustern die Zeichenfläche Formen ausfüllt. Sie können die Pinselfarbe mit anderen bereits auf der Zeichenfläche vorhandenen Farben kombinieren. Die vordefinierten Stile umfassen Vordergrundfarbe, Transparent und verschiedene Linien- und Gittermuster.

Um den Stil eines Pinsels zu ändern, setzen Sie seine Eigenschaft *Style* auf einen der vordefinierten Werte *bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross* oder *bsDiagCross.*

Im folgenden Beispiel werden Pinselstile festgelegt, indem für die acht Pinselstil-Schaltflächen eine gemeinsame Behandlungsroutine für das Ereignis *OnClick* verwendet wird. Dazu markieren Sie alle acht Schaltflächen und geben im Objektinspektor der Ereignisbehandlungsroutine für *OnClick* den Namen *SetBrushStyle.* Hier der Quelltext der Behandlungsroutine:

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
```

Die Pinseleigenschaft Bitmap setzen

Mit der Eigenschaft *Bitmap* eines Pinsels können Sie ein Bitmap-Bild für den Pinsel festlegen, das beim Füllen von Formen und anderen Bereichen als Muster verwendet werden soll.

Im folgenden Beispiel wird ein Bitmap aus einer Datei geladen und dem Pinsel der Eigenschaft *Canvas* des Formulars *Form1* zugewiesen:

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
```

```
    finally
      Form1.Canvas.Brush.Bitmap := nil;
      Bitmap.Free;
    end;
  end;
```

Hinweis Der Pinsel wird nicht zum Eigentümer des Bitmap-Objekts, das seiner Eigenschaft *Bitmap* zugewiesen wurde. Sie müssen daher sicherstellen, daß das Bitmap-Objekt während der Lebensdauer des Pinsels gültig bleibt, und es anschließend selbst freigeben.

Pixel lesen und setzen

Jede Zeichenfläche besitzt eine indizierte Eigenschaft namens *Pixels*, welche die einzelnen Farbpunkte repräsentiert, aus denen sich das Bild auf der Zeichenfläche zusammensetzt. Sie brauchen auf diese Eigenschaft nur in seltenen Fällen direkt zuzugreifen. Sie steht nur zur Verfügung, um bequem kleinere Aktionen ausführen zu können, wie etwa das Suchen oder Zuweisen einer Pixelfarbe.

Hinweis Das Festlegen und Ermitteln von Pixeln ist um ein Vielfaches langsamer als das Ausführen von Grafikoperationen in den entsprechenden Bereichen. Verwenden Sie für den Zugriff auf die Bild-Pixel eines allgemeinen Arrays nicht die Array-Eigenschaft *Pixel*. Informationen über einen schnellen Zugriff auf Bild-Pixel finden Sie bei der Erläuterung der Eigenschaft *ScanLine* von *TBitmap*.

Grafikobjekte zeichnen

In diesem Abschnitt wird erläutert, wie mit Hilfe von Zeichenflächenmethoden Grafikobjekte gezeichnet werden können. Folgende Themen werden behandelt:

- Linien und Linienzüge zeichnen
- Formen zeichnen
- Abgerundete Rechtecke zeichnen
- Polygone zeichnen

Linien und Linienzüge zeichnen

Eine Zeichenfläche kann gerade Linien und Linienzüge darstellen. Eine gerade Linie besteht aus einer Pixel-Linie, die zwei Punkte verbindet. Ein Linienzug ist eine Folge gerader Linien, deren Enden miteinander verbunden sind. Die Zeichenfläche zeichnet alle Linien mit Hilfe ihres Stiftes.

Linien zeichnen

Um auf einer Zeichenfläche eine gerade Linie zu zeichnen, verwenden Sie die Methode *LineTo* der Zeichenfläche.

LineTo zeichnet eine Linie von der aktuellen Stiftposition zum angegebenen Punkt und definiert den Endpunkt der Linie als aktuelle Position. Die Zeichenfläche zeichnet die Linie mit Hilfe ihres Stiftes.

Die folgende Methode zeichnet beispielsweise bei jedem Bildaufbau in einem Formular Diagonalen, die sich überkreuzen:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
 begin
   MoveTo(0, 0);
   LineTo(ClientWidth, ClientHeight);
   MoveTo(0, ClientHeight);
   LineTo(ClientWidth, 0);
 end;
end;
```

Linienzüge zeichnen

Die Zeichenfläche kann nicht nur einzelne Linien, sondern auch ganze Linienzüge zeichnen. Ein Linienzug ist eine Folge von mehreren miteinander verbundenen Liniensegmenten.

Um auf einer Zeichenfläche einen Linienzug darzustellen, rufen Sie die Methode *Polyline* der Zeichenfläche auf.

Der Parameter, der an die Methode *PolyLine* übergeben wird, ist ein Array von Punkten. Sie können sich die Erstellung eines Linienzugs als Aufruf von *MoveTo* für den ersten und *LineTo* für jeden folgenden Punkt vorstellen. Ein Linienzug läßt sich mit Polyline allerdings schneller zeichnen als mit einzelnen Aufrufen von *MoveTo* und *LineTo*, weil die Methode einen Großteil der wiederholten Aufrufe überflüssig macht.

Die folgende Methode zeichnet eine Raute in einem Formular:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
   PolyLine([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

Dieses Beispiel nutzt die Fähigkeit von Delphi, offene Array-Parameter dynamisch zu erstellen. Sie können natürlich jedes beliebige Punkte-Array übergeben. Die einfachste und schnellste Möglichkeit, das Array zu erzeugen, besteht jedoch darin, die Punkte in Klammern einzuschließen und das gesamte Objekt als Parameter zu übergeben. Weitere Informationen zu diesem Thema finden Sie in der Online-Hilfe.

Formen zeichnen

Zeichenflächen verfügen über Methoden zum Zeichnen verschiedener Formen. Die Zeichenfläche zeichnet den Umriß einer Form mit ihrem Stift und füllt sie dann mit ihrem Pinsel aus. Die Linie, die den Umriß der Form bildet, wird vom aktuellen *Pen-Objekt* bestimmt.

Dieser Abschnitt behandelt die folgenden Themen:

- Rechtecke und Ellipsen zeichnen
- Abgerundete Rechtecke zeichnen
- Polygone zeichnen

Rechtecke und Ellipsen zeichnen

Um ein Rechteck oder eine Ellipse auf einer Zeichenfläche zu zeichnen, rufen Sie die Methode *Rectangle* bzw. *Ellipse* der Zeichenfläche auf und übergeben die Koordinaten des begrenzenden Rechtecks.

Die Methode *Rectangle* zeichnet das begrenzende Rechteck. Die Methode *Ellipse* zeichnet eine Ellipse, die jede Seite des Rechtecks berührt.

Mit der folgenden Methode wird zunächst ein Rechteck gezeichnet, das den oberen linken Quadranten eines Formulars ausfüllt, und danach eine Ellipse im gleichen Bereich:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
 Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

Abgerundete Rechtecke zeichnen

Um ein abgerundetes Rechteck auf einer Zeichenfläche darzustellen, rufen Sie die Methode *RoundRect* der Zeichenfläche auf.

Die ersten vier Parameter, die an *RoundRect* übergeben werden, bezeichnen genau wie bei den Methoden *Rectangle* oder *Ellipse* ein begrenzendes Rechteck. *RoundRect* erwartet zwei weitere Parameter, die festlegen, wie die abgerundeten Ecken gezeichnet werden sollen.

Mit der folgenden Methode wird ein Rechteck im oberen linken Quadranten eines Formulars gezeichnet. Die abgerundeten Ecken des Rechtecks werden als Kreissegmente mit einem Durchmesser von 10 Pixeln dargestellt:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

Polygone zeichnen

Um ein Polygon mit einer beliebigen Anzahl von Seiten auf einer Zeichenfläche zu plazieren, rufen Sie die Methode *Polygon* der Zeichenfläche auf.

Polygon übernimmt als einzigen Parameter ein Array von Punkten und verbindet diese Punkte mit Hilfe des Stiftes. Der letzte Punkt wird anschließend mit dem ersten Punkt verbunden, um das Polygon zu schließen. Nach dem Zeichnen der Linien füllt *Polygon* die Fläche innerhalb des Polygons mit dem Pinsel aus.

Mit der folgenden Methode wird beispielsweise ein gleichseitiges Dreieck in der unteren rechten Hälfte eines Formulars gezeichnet:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
   Point(ClientWidth, ClientHeight)]);
end;
```

Behandlung mehrerer Zeichenobjekte in einer Anwendung

Normalerweise stehen in Werkzeugleisten oder Schalterleisten verschiedene Zeichenmethoden (für Rechtecke, Formen Linien usw.) zur Verfügung. Anwendungen reagieren auf das Anklicken von Mauspalettenschaltern und bestimmen das gewünschte Zeichenobjekt. In diesem Abschnitt werden folgende Themen erläutert:

- Benötigte Zeichenwerkzeuge ermitteln
- Werkzeuge mit Hilfe von Mauspalettenschaltern wechseln
- Zeichenwerkzeuge verwenden

Benötigte Zeichenwerkzeuge ermitteln

Ein Grafikprogramm muß die Möglichkeit haben, zu erkennen, welche Art von Zeichenwerkzeug (z. B. Linie, Rechteck, Ellipse oder abgerundetes Rechteck) ein Benutzer zu einem bestimmten Zeitpunkt verwenden will. Sie könnten zu diesem Zweck jedem Werkzeugtyp eine Nummer zuweisen, müßten sich dann jedoch merken, welches Werkzeug eine Zahl repräsentiert. Dies könnten Sie erreichen, indem Sie jeder Zahl einen aussagekräftigen Konstantennamen zuweisen. Im Quelltext könnten Sie dann jedoch nicht unterscheiden, welche Zahlen sich im gültigen Bereich befinden und vom richtigen Typ sind. Object Pascal bietet eine Lösung für dieses Problem: Sie deklarieren einen Aufzählungstyp.

Ein Aufzählungstyp ist eine effektive Möglichkeit, Konstanten sequentielle Werte zuzuweisen. Da es sich außerdem um eine Typdeklaration handelt, können Sie die Typprüfung von Object Pascal verwenden, um sicherzustellen, daß nur gültige Werte zugewiesen werden.

Um einen Aufzählungstyp zu deklarieren, verwenden Sie das reservierte Wort type, gefolgt von einem Typbezeichner, einem Gleichheitszeichen und (in Klammern eingeschlossen und durch Kommas getrennt) den Bezeichnern für die Werte des Typs.

Der folgende Quelltext deklariert beispielsweise einen Aufzählungstyp für jedes in einer Grafikanwendung vorhandene Zeichenwerkzeug:

```
type
  TDrawingTool = (zwLine, zwRectangle, zwEllipse, zwRoundRect);
```

Per Konvention beginnen Typbezeichner mit dem Buchstaben T. Eine Gruppe gleichartiger Konstanten (wie in einem Aufzählungstyp) beginnt mit einem Präfix aus zwei Buchstaben (wie *zw* für »Zeichenwerkzeug«).

Die Deklaration des Typs *TDrawingTool* entspricht der Deklaration einer Gruppe von Konstanten:

```
const
  zwLine = 0;
 zwRectangle = 1;
 zwEllipse = 2;
  zwRoundRect = 3;
```

Der Hauptunterschied besteht darin, daß bei der Deklaration eines Aufzählungstyps den Konstanten nicht nur ein Wert, sondern auch ein Typ zugewiesen wird. Dies ermöglicht die Verwendung der Typprüfung von Object Pascal und trägt dazu bei, Fehler zu vermeiden. Einer Variablen des Typs *TDrawingTool* kann nur eine der Konstanten *zwLine* bis *zwRoundRect* zugewiesen werden. Jeder Versuch, eine andere Zahl zuzuweisen (auch wenn sie im Bereich 0..3 liegt), wird vom Compiler zurückgewiesen.

Im folgenden Quelltext wird einem Formular ein Feld hinzugefügt, das zum Speichern des aktuellen Zeichenwerkzeugs dient:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
 TForm1 = class(TForm)
    ...     { Methodendeklarationen }
 public
    Drawing: Boolean;
   Origin, MovePt: TPoint;
   DrawingTool: TDrawingTool;{ Feld zum Speichern des aktuellen Werkzeugs }
 end;
```

Werkzeuge mit Hilfe von Mauspalettenschaltern wechseln

Für jedes Zeichenwerkzeug muß eine *OnClick*-Ereignisbehandlungsroutine bereitgestellt werden. Angenommen, eine Anwendung verfügt über Mauspalettenschalter für alle vier Zeichenwerkzeuge (Linien, Rechtecke, Ellipsen und abgerundete Rechtecke). In diesem Fall könnten Sie den *OnClick*-Ereignissen der vier Zeichenwerkzeug-Schaltflächen die folgenden Behandlungsroutinen zuordnen und *DrawingTool* dort auf den entsprechenden Wert setzen:

```
procedure TForm1.LineButtonClick(Sender: TObject); { Linien-Schaltfläche}
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject); { Rechteck-Schaltfläche}
begin
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject); { Ellipsen-Schaltfläche }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject); { Schaltfläche für abgerundete
                                                            Rechtecke }
begin
```

```
    DrawingTool := dtRoundRect;
end;
```

Zeichenwerkzeuge verwenden

Nachdem nun die Zeichenwerkzeuge erkannt werden, müssen Sie festlegen, wie die verschiedenen Formen gezeichnet werden sollen. Die einzigen Methoden, die Zeichenaktionen ausführen, sind die Behandlungsroutinen für *OnMouseMove* und *OnMouseUp*. Es können aber nur Linien gezeichnet werden, unabhängig davon, welches Zeichenwerkzeug ausgewählt ist.

Um verschiedene Zeichenwerkzeuge zu verwenden, müssen Sie im Quelltext festlegen, wie mit dem ausgewählten Werkzeug gezeichnet werden soll. Die entsprechende Anweisung wird den Ereignisbehandlungsroutinen der einzelnen Werkzeuge hinzugefügt.

In diesem Abschnitt werden folgende Themen behandelt:

- Formen zeichnen
- Gemeinsamer Quelltext für Ereignisbehandlungsroutinen

Formen zeichnen

Das Zeichnen von Formen ist genauso einfach wie das Zeichnen von Linien: Für jede Form ist eine einzelne Anweisung erforderlich. Es werden nur die Koordinaten benötigt.

Der folgende Quelltext ist eine neue Fassung der Ereignisbehandlungsroutine für *OnMouseUp*, die Formen für alle vier Zeichenwerkzeuge zeichnet:

```
procedure TForm1.FormMouseUp(Sender: TObject);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
       Canvas.LineTo(X, Y)
      end;
   dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
   dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
   dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                  (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
 Drawing := False;
end;
```

Nun muß noch die Behandlungsroutine für das Ereignis *OnMouseMove* zum Zeichnen von Formen aktualisiert werden:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;
```

```
    case DrawingTool of
       dtLine: begin
                 Canvas.MoveTo(Origin.X, Origin.Y);
                Canvas.LineTo(MovePt.X, MovePt.Y);
                Canvas.MoveTo(Origin.X, Origin.Y);
                Canvas.LineTo(X, Y);
               end;
      dtRectangle: begin
                     Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                    Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                   end;
      dtEllipse: begin
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                  Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                 end;
      dtRoundRect: begin
                     Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                   end;
     end;
     MovePt := Point(X, Y);
    end;
   Canvas.Pen.Mode := pmCopy;
  end;
```

Normalerweise würden alle Teile dieses Quelltextes, die ständig wiederholt werden, in einer separaten Routine abgelegt. Im nächsten Abschnitt wird der Quelltext für das Zeichnen von Formen in einer einzelnen Routine zusammengefaßt, auf die alle Behandlungsroutinen für Mausereignisse zugreifen können.

Gemeinsamer Quelltext für Ereignisbehandlungsroutinen

Wenn mehrere Ereignisbehandlungsroutinen dieselben Programmteile verwenden, können Sie die Anwendung effizienter gestalten, indem Sie den betreffenden Quelltext in eine Routine verlagern, auf die alle Ereignisbehandlungsroutinen zugreifen können.

Um eine Methode in ein Formular aufzunehmen, gehen Sie folgendermaßen vor:

1 Fügen sie dem Formularobjekt die Methodendeklaration hinzu.

   Sie können die Deklaration entweder im **public**- oder im **private**-Teil am Ende der Deklaration des Formularobjekts hinzufügen. Wenn der gemeinsame Quelltext nur Details für die Behandlung bestimmter Ereignisse betrifft, ist es am sichersten, wenn Sie die Methode als **private** deklarieren.

2 Schreiben Sie die Methodenimplementierung im **implementation**-Abschnitt der Unit.

Der Kopf der Methodenimplementierung muß genau mit der Deklaration übereinstimmen, d. h. die Parameter müssen dieselbe Reihenfolge haben.

Der nachfolgende Quelltext fügt dem Formular eine Methode namens *DrawShape* hinzu und ruft sie in jeder Behandlungsroutine auf. Zunächst wird die Deklaration von *DrawShape* in die Deklaration des Formularobjekts eingefügt:

```
type
  TForm1 = class(TForm)
    ...     { Deklaration von Feldern und Methoden }
 public
    { Public-Deklarationen }
   procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

Danach wird im implementation-Abschnitt der Unit die Methode *DrawShape* implementiert:

```
implementation
{$R *.FRM}
...   { Hier können weitere Methodenimplementationen stehen }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
    Pen.Mode := AMode;
   case DrawingTool of
      dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
         LineTo(BottomRight.X, BottomRight.Y);
        end;
     dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
        (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
    end;
  end;
end;
```

Die anderen Ereignisbehandlungsroutinen werden so geändert, daß sie *DrawShape* aufrufen:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy); { Die endgültige Form zeichnen }
 Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor); { Die vorhergehende Form löschen }
   MovePt := Point(X, Y);  { Die aktuelle Position speichern }
   DrawShape(Origin, MovePt, pmNotXor); { Die aktuelle Form speichern }
  end;
end;
```

In einer Grafik zeichnen

Sie benötigen keine Komponenten, um in Ihrer Anwendung Grafikobjekte zu bearbeiten. Sie können Grafikobjekte konstruieren, zeichnen, speichern und löschen, ohne jemals etwas auf den Bildschirm zu zeichnen. Es wird nur selten vorkommen, daß Ihre Anwendungen direkt auf ein Formular zeichnen. Meist zeichnet eine Anwendung in einer Grafik und zeigt diese mit Hilfe eines Bild-Steuerelements der VCL in einem Formular an.

Sobald Sie die Zeichnung der Anwendung in die Grafik des Bild-Steuerelements eingefügt haben, lassen sich Drucken-, Zwischenablage- sowie Laden- und Speichern-Operationen für die Grafikobjekte sehr einfach implementieren. Bei Grafikobjekten kann es sich um Bitmap-Dateien, Metadateien, Symbole oder um beliebige installierte Grafikklassen handeln, z. B. um JPEG-Grafiken.

Hinweis Da Sie ein Bild zeichnen, das sich nicht auf dem Bildschirm befindet (z. B. im Fall einer *TBitmap*-Zeichenfläche), wird das Bild erst sichtbar, wenn ein Steuerelement die Zeichnung aus dem Bitmap in seine Zeichenfläche kopiert. Mit anderen Worten: Wenn Sie ein Bitmap zeichnen und es einem Bild-Steuerelement zuweisen, wird das Bild nur angezeigt, wenn das Steuerelement die Möglichkeit hat, seine Zeichenbotschaften zu verarbeiten. Zeichnen Sie dagegen direkt in die *Canvas*-Eigenschaft eines Steuerelements, wird das Bildobjekt sofort angezeigt.

Bildlauffähige Grafiken

Die Grafik muß nicht dieselbe Größe wie das Formular haben. Sie kann auch größer oder kleiner sein. Wenn Sie dem Formular ein Bildlauffeld (eine *TScrollBox*-Komponente) hinzufügen und die Grafik darin plazieren, können Sie Grafiken anzeigen, die größer als das Formular oder sogar größer als der Bildschirm sind. Zu diesem Zweck fügen Sie zunächst eine *TScrollbox*-Komponente und danach das Bild-Steuerelement ein.

Bild-Steuerelemente hinzufügen

Ein Bild-Steuerelement ist eine Container-Komponente, mit deren Hilfe Sie Bitmap-Objekte anzeigen können. Mit einem Bild-Steuerelement kann ein Bitmap gespeichert werden, das nicht die ganze Zeit über sichtbar sein muß oder das von einer Anwendung zur Generierung anderer Bilder benötigt wird.

Hinweis Im Abschnitt »Grafikkomponenten« auf Seite 2-25 wird erläutert, wie Grafiken in Steuerelementen verwendet werden können.

Einfügen des Steuerelements

Sie können ein Bild-Steuerelement überall im Formular plazieren. Ein Bild-Steuerelement besitzt die Fähigkeit, seine Größe an die des Bildes anzupassen. Wenn Sie diese Möglichkeit nutzen wollen, brauchen Sie nur die linke obere Ecke festzulegen. Falls es sich bei dem Bild-Steuerelement um einen unsichtbaren Platzhalter für ein Bitmap handelt, können Sie es wie eine nicht-visuelle Komponente an einer beliebigen Stelle plazieren.

Wenn das Bild-Steuerelement in einem Bildlauffeld abgelegt wird, das bereits am Client-Bereich des Formulars ausgerichtet ist, werden die Bildlaufleisten für den Zugriff auf die nicht sichtbaren Teile des Bildes eingeblendet. Anschließend legen Sie die Eigenschaften des Bild-Steuerelements fest.

Die Anfangsgröße eines Bitmaps festlegen

Wenn Sie ein Bild-Steuerelement einfügen, handelt es sich dabei nur um einen Container. Sie können aber zur Entwurfszeit die Eigenschaft *Picture* des Bild-Steuerelements mit einer statischen Grafik belegen. Das Steuerelement kann sein Bild aber auch zur Laufzeit aus einer Datei laden, wie es im Abschnitt »Grafikdateien laden und speichern« auf Seite 7-20 beschrieben wird.

Um beim Start der Anwendung ein leeres Bitmap zu erstellen, gehen Sie folgendermaßen vor:

1. Verknüpfen Sie eine Behandlungsroutine mit dem Ereignis *OnCreate* des Formulars, das das Bild enthält.
2. Erstellen Sie ein Bitmap-Objekt, und weisen Sie es der Eigenschaft *Picture.Graphic* des Bild-Steuerelements zu.

Im folgenden Beispiel befindet sich das Bild im Hauptformular der Anwendung (*Form1*). Aus diesem Grund ordnet der Quelltext dem Ereignis *OnCreate* von *Form1* eine Behandlungsroutine zu:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap; { Temporäre Variable für das Bitmap }
begin
  Bitmap := TBitmap.Create; { Das Bitmap-Objekt erstellen }
 Bitmap.Width := 200; { Anfangsbreite... }
 Bitmap.Height := 200; { ...und Anfangshöhe zuweisen }
 Image.Picture.Graphic := Bitmap; { Das Bitmap dem Bild-Steuerelement zuweisen }
end;
```

Durch die Zuweisung des Bitmaps an die Eigenschaft *Graphic* des Bildes wird das Bildobjekt zum Eigentümer des Bitmaps. Da das Bildobjekt das Bitmap nach Gebrauch freigibt, brauchen Sie nicht selbst für die Freigabe zu sorgen. Sie können dem Bild ein anderes Bitmap zuweisen (siehe »Bilder ersetzen« auf Seite 7-22). Das Bild übernimmt das neue Bitmap dann nach der Freigabe des alten.

Wenn Sie jetzt die Anwendung starten, sehen Sie den Client-Bereich des Formulars als weiße Fläche, die das Bitmap darstellt. Wenn Sie die Größe des Fensters so ändern, daß der Client-Bereich nicht das gesamte Bild anzeigt, blendet die Bildlauffeld-Komponente automatisch Bildlaufleisten ein. Mit Hilfe dieser Leisten können Sie den Rest des Bildes anzeigen. Wenn Sie aber versuchen, in das Bild zu zeichnen, geschieht nichts, weil die Anwendung noch immer in dem Formular zeichnet, das sich jetzt hinter dem Bild und dem Bildlauffeld befindet.

Zeichnen im Bitmap

Um in einem Bitmap zu zeichnen, verwenden Sie die Zeichenfläche des Bild-Steuerelements und verbinden die Behandlungsroutinen für Mausereignisse mit den ent-

sprechenden Ereignissen des Bild-Steuerelements. Normalerweise werden Sie hierfür Bereichsoperationen verwenden (Füllungen, Rechtecke, Linienzüge usw.). Diese ermöglichen ein schnelles und effektives Zeichnen.

Zum Zugriff auf einzelne Pixel steht Ihnen die Bitmap-Eigenschaft *ScanLine* zur Verfügung. Im Normalfall setzen Sie das Pixel-Format des Bitmaps auf 24 Bit und behandeln den Zeiger, der von *ScanLine* zurückgegeben wird, als RGB-Array. Andernfalls müssen Sie das native Format der Eigenschaft *ScanLine* kennen. Das folgende Beispiel illustriert, wie mit der Eigenschaft *ScanLine* Pixel zeilenweise ermittelt werden können:

```
procedure TForm1.Button1Click(Sender: TObject);
// Dieses Beispiel demonstriert das direkte Zeichnen im Bitmap.
var
  x,y : integer;
  BitMap : TBitMap;
  P : PByteArray;
begin
  BitMap := TBitMap.create;
  try
    BitMap.LoadFromFile('C:\Program Files\Borland\Delphi 4\Images\Splash\256color\
factory.bmp');
    for y := 0 to BitMap.height -1 do
    begin
      P := BitMap.ScanLine[y];
      for x := 0 to BitMap.width -1 do
        P[x] := y;
    end;
  canvas.draw(0,0,BitMap);
  finally
    BitMap.free;
  end;
end;
```

Grafikdateien laden und speichern

Grafikbilder, die nur für die Dauer der Programmausführung existieren, sind von sehr begrenztem Wert. In der Regel möchten Sie ein Bild immer wieder verwenden oder eine neu erstellte Grafik für den späteren Gebrauch speichern. Das Bild-Steuerelement der VCL vereinfacht das Laden und Speichern von Bildern in Dateien.

Die VCL-Komponenten zum Laden, Speichern und Ersetzen von Grafikbildern unterstützen eine Vielzahl verschiedener Grafikformate, z. B. Bitmap-Dateien, Metadateien, Symbole usw. Sie unterstützen auch installierbare Grafik-Klassen.

Das Laden und Speichern von Grafikdateien erfolgt ähnlich wie bei allen anderen Dateien und wird in den folgenden AbschnittenThemen beschrieben:

- Bilder aus Dateien laden
- Bilder in Dateien speichern
- Bilder ersetzen

Bilder aus Dateien laden

Wenn Sie ein Bild verändern oder zur Weiterbearbeitung an eine andere Person oder Anwendung weitergeben möchten, ist es wichtig, daß Ihre Anwendung Bilder aus einer Datei laden kann.

Um eine Grafikdatei in ein Bild-Steuerelement zu laden, rufen Sie die Methode *LoadFromFile* des Objekts Picture im Bild-Steuerelement auf.

Der folgende Quelltext übernimmt einen Dateinamen aus einem Dialogfeld *Datei öffnen* in ein Bild-Steuerelement namens Image:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    CurrentFile := OpenDialog1.FileName;
   Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

Bilder in Dateien speichern

VCL-Bildobjekte können Grafiken in unterschiedlichen Formaten laden und speichern. Sie können aber auch eigene Formate für Grafikdateien erstellen und diese in Bildobjekten speichern bzw. aus diesen laden.

Um den Inhalt eines Bild-Steuerelements in einer Datei zu speichern, rufen Sie die Methode *SaveToFile* des Objekts Picture im Bild-Steuerelement auf.

Die Methode *SaveToFile* benötigt den Namen der Datei, in der das Objekt gespeichert wird. Wenn das Bild neu erstellt wurde oder ein vorhandenes Bild unter einem neuen Namen gespeichert werden soll, muß die Anwendung vor dem Speichern einen Dateinamen vom Benutzer anfordern. Wie dies bewerkstelligt wird, zeigt der nächste Abschnitt.

Die beiden folgenden Ereignisbehandlungsroutinen sind den Menüeinträgen *Datei / Speichern* und *Datei / Speichern unter* zugeordnet. Sie verarbeiten das Speichern bereits benannter, unbenannter und existierender Dateien mit neuen Namen.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile) { Speichern, wenn Name bereits vorhanden, }
 else SaveAs1Click(Sender); { andernfalls Namen anfordern }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then { Dateinamen anfordern }
  begin
    CurrentFile := SaveDialog1.FileName; { Eingegebenen Namen speichern, }
   Save1Click(Sender); { dann normal speichern }
  end;
end;
```

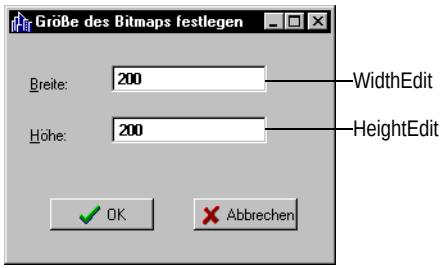
Bilder ersetzen

Sie können die Grafik in einem Bild-Steuerelement jederzeit durch eine andere ersetzen. Wenn Sie einem Bild, dem bereits eine Grafik zugeordnet ist, eine neue Grafik zuweisen, ersetzt diese die vorhandene Grafik.

Um ein Bild in einem Bild-Steuerelement zu ersetzen, weisen Sie dem Objekt *Picture* des Bild-Steuerelements die neue Grafik zu.

Beim Erstellen einer neuen Grafik gehen Sie genauso vor wie beim Erzeugen einer Anfangsgrafik (siehe »Die Anfangsgröße eines Bitmaps festlegen« auf Seite 7-19). Sie sollten aber dem Benutzer auch die Wahl einer anderen Größe ermöglichen. Eine einfache Möglichkeit ist ein Dialogfeld wie in Abbildung 7.1.

Abbildung 7.1 Das Dialogfeld zur Größenänderung von Bitmaps aus der Unit *BMPDlg*.



Dieses spezielle Dialogfeld wird in der Unit *BMPDlg* erzeugt, die im Projekt *GraphEx* enthalten ist (im Verzeichnis EXAMPLES\DOC\GRAPHEX).

Nehmen Sie die Dialogfeld-Unit in die uses-Klausel der Hauptformular-Unit auf. Anschließend weisen Sie dem Ereignis *OnClick* für den Menüeintrag *Datei / Neu* eine Ereignisbehandlungsroutine zu. Das folgende Beispiel demonstriert dieses Vorgehen:

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap; { Temporäre Variable für das neue Bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit; { Das Feld muß für die Breiteneinstellung fokussiert sein }
   WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width); { Aktuelle Abmessungen... }
   HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height); { ...als Vorgabe verwenden }
   if ShowModal <> idCancel then { Wenn der Benutzer das Dialogfeld nicht verläßt,... }
    begin
      Bitmap := TBitmap.Create; { ...neues Bitmap-Objekt erstellen }
     Bitmap.Width := StrToInt(WidthEdit.Text); { Angegebene Breite verwenden }
     Bitmap.Height := StrToInt(HeightEdit.Text); { Angegebene Höhe verwenden }
     Image.Picture.Graphic := Bitmap; { Grafik durch neues Bitmap ersetzen }
     CurrentFile := ''; { Unbenannte Datei anzeigen }
    end;
  end;
end;
```

Hinweis Durch das Zuweisen eines neuen Bitmaps an die Eigenschaft *Graphic* des Bildobjekts wird das vorhandene Bitmap entfernt und durch ein neues ersetzt. Das Bildobjekt wird Eigentümer des neuen Bitmaps. Die VCL gibt die Ressourcen des gelöschten Bitmaps automatisch frei.

Die Zwischenablage und Grafiken

Sie können die Windows-Zwischenablage verwenden, um Grafiken in Ihren Anwendungen zu kopieren und einzufügen oder mit anderen Anwendungen auszutauschen. Das Zwischenablageobjekt (*Clipboard*) der VCL erleichtert Ihnen den Umgang mit verschiedenen Arten von Informationen, einschließlich Grafiken.

Damit Sie das Zwischenablageobjekt in Ihrer Anwendung einsetzen können, müssen Sie die Unit *Clipbrd* in die **uses**-Klausel jeder Unit einfügen, die Zugriff auf die Daten der Zwischenablage benötigt.

Grafiken in die Zwischenablage kopieren

Sie können jedes Bild, auch den Inhalt eines Bild-Steuerelements, in die Zwischenablage kopieren, so daß es für alle Windows-Anwendungen verfügbar ist.

Um ein Bild in die Zwischenablage zu kopieren, weisen Sie es mit Hilfe der Methode *Assign* dem Objekt *Clipboard* zu.

Der folgende Quelltext zeigt, wie die Grafik aus einem Bild-Steuerelement mit der Bezeichnung *Image* mit Hilfe des Menüeintrags *Bearbeiten* / *Kopieren* in die Zwischenablage kopiert wird.

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

Grafiken in die Zwischenablage ausschneiden

Das Ausschneiden einer Grafik in die Zwischenablage entspricht dem Kopieren einer Grafik, wobei jedoch die ursprüngliche Grafik gelöscht wird.

Um eine Grafik aus einem Bild in die Zwischenablage auszuschneiden, kopieren Sie sie zunächst in die Zwischenablage und löschen anschließend das Original.

Normalerweise ist beim Ausschneiden nur die Frage zu klären, wie der gelöschte Teil im Originalbild zu behandeln ist. Das Ausfüllen des Bereichs durch die Farbe Weiß ist der übliche Ansatz. Der folgende Quelltext mit einer Behandlungsroutine für das Ereignis *OnClick* des Menüeintrags *Bearbeiten / Ausschneiden* zeigt dieses Vorgehen:

```
procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender); { Bild in die Zwischenablage kopieren }
 with Image.Canvas do
  begin
```

```
    CopyMode := cmWhiteness; { Alles mit der Farbe Weiß füllen }
   ARect := Rect(0, 0, Image.Width, Image.Height); { Abmessungen des Bitmaps
ermitteln }
   CopyRect(ARect, Image.Canvas, ARect); { Bitmap auf sich selbst kopieren }
   CopyMode := cmSrcCopy; { Standardmodus wiederherstellen }
  end;
end;
```

Grafiken aus der Zwischenablage einfügen

Wenn die Windows-Zwischenablage eine Bitmap-Grafik enthält, können Sie diese in jedes andere Bildobjekt einfügen, auch in Bild-Steuerelemente und in die Oberfläche eines Formulars.

Um eine Grafik aus der Zwischenablage einzufügen, gehen Sie folgendermaßen vor:

1. Rufen Sie die Methode *HasFormat* der Zwischenablage auf, um festzustellen, ob sie eine Grafik enthält.

   HasFormat ist eine Boolesche Funktion, die *True* zurückgibt, wenn die Zwischenablage ein Objekt des als Parameter übergebenen Typs enthält. Um festzustellen, ob der Inhalt eine Grafik ist, übergeben Sie CF_BITMAP.

2. Verknüpfen Sie die Zwischenablage mit dem Ziel.

Der folgende Quelltext zeigt, wie als Reaktion auf das Anklicken des Menüeintrags *Bearbeiten / Einfügen* ein Bild aus der Zwischenablage in ein Bild-Steuerelement kopiert wird:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.HasFormat(CF_BITMAP) then    { Befindet sich ein Bild in der Zwischenablage? )
 begin
   Image.Picture.Bitmap.Assign(Clipboard);
      end;
end;
```

Die Grafik in der Zwischenablage könnte aus der Anwendung selbst stammen oder aus einem anderen Programm, z. B. Windows Paintbrush. In diesem Fall braucht das Format in der Zwischenablage nicht geprüft zu werden, da der Menüeintrag für das Einfügen deaktiviert sein muß, wenn sich in der Zwischenablage kein unterstütztes Format befindet.

Der Gummiband-Effekt: Beispiel

Dieser Abschnitt befaßt sich ausführlich mit der Implementierung des »Gummiband-Effekts« in einer Grafikanwendung. Darunter versteht man das Nachverfolgen der Mausbewegungen zur Laufzeit, während der Benutzer eine Grafik zeichnet. Die Quelltextbeispiele in diesem Abschnitt stammen aus einer Beispielanwendung, die sich im Verzeichnis EXAMPLES\DOC\GRAPHEX befindet. Die Anwendung zeichnet als Antwort auf Klick- und Ziehbewegungen Linien und Formen in der Zeichen-

fläche eines Fensters. Das Drücken der Maustaste startet den Zeichenvorgang, das Loslassen beendet ihn.

Zunächst wird gezeigt, wie man auf der Oberfläche des Hauptformulars zeichnet. Später wird das Zeichnen in einem Bitmap demonstriert.

In diesem Abschnitt werden folgende Themen behandelt:

- Auf Mausaktionen reagieren
- Felder einem Formularobjekt hinzufügen
- Verbesserte Liniendarstellung

Auf Mausaktionen reagieren

Ihre Anwendungen können auf folgende Mausaktionen reagieren: Maustaste gedrückt, Maus bewegt, Maustaste losgelassen. Außerdem können sie auf das Klicken (Drücken und Loslassen ohne Bewegung der Maus) reagieren, das auch durch Drükken einer Taste oder Tastenkombination simuliert werden kann (z. B. *Return* in einem modalen Dialogfeld).

Nachfolgend werden folgende Themen behandelt:

- Woraus besteht ein Mausereignis?
- Auf das Drücken einer Maustaste reagieren
- Auf das Loslassen einer Maustaste reagieren
- Auf das Bewegen der Maus reagieren

Woraus besteht ein Mausereignis?

In der VCL sind drei Mausereignisse definiert: *OnMouseDown~*, *OnMouseMove* und *OnMouseUp*.

Sobald eine VCL-Anwendung eine Mausaktion feststellt, wird die Ereignisbehandlungsroutine für das entsprechende Ereignis aufgerufen. Dieser werden fünf Parameter übergeben, deren Inhalte Sie verwenden können, um die Reaktion auf das Ereignis an Ihre Bedürfnisse anzupassen. Die fünf Parameter sind:

Tabelle 7.4 Mausereignis-Parameter

Parameter	Bedeutung
Sender	Das Objekt, das die Mausaktion festgestellt hat.
Button	Gibt an, welche Maustaste gedrückt wurde: *mbLeft*, *mbMiddle* oder *mbRight*.
Shift	Gibt den Zustand der Tasten *Alt*, *Strg* und *Umschalt* zum Zeitpunkt der Mausaktion an.
X, Y	Die Koordinaten, an denen das Ereignis aufgetreten ist.

Meist sind die Koordinaten die wichtigste Information für die Behandlung von Mausereignissen. In bestimmten Fällen werden Sie aber auch durch einen Lesezugriff auf *Button* feststellen müssen, welche Maustaste das Ereignis ausgelöst hat.

Hinweis Delphi verwendet dieselben Kriterien wie Windows, um herauszufinden, welche Maustaste gedrückt wurde. Wenn Sie daher standardmäßig die linke und die rechte Maustaste vertauscht haben, bewirkt ein Klick mit der rechten Maustaste, daß der Parameter *Button* den Wert *mbLeft* annimmt.

Auf das Drücken einer Maustaste reagieren

Jedesmal wenn eine Maustaste gedrückt wird, empfängt das Objekt unter dem Mauszeiger ein *OnMouseDown-Ereignis*. Das Objekt kann dann auf das Ereignis reagieren.

Zu diesem Zweck ordnen Sie dem Ereignis *OnMouseDown* eine Ereignisbehandlungsroutine zu.

Die VCL erzeugt im Formular eine leere Behandlungsroutine für das Ereignis *OnMouseDown:*

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

Das folgende Beispiel zeigt Text an der Stelle an, an der die Maustaste gedrückt wurde. Dafür werden die *X*- und *Y*-Koordinaten verwendet, die der Methode als Parameter übergeben wurden. Die Methode *TextOut* der Zeichenfläche wird aufgerufen, um den Text darzustellen:

Der folgende Quelltext blendet den String »Hier!« an der Stelle im Formular ein, an der mit der Maus geklickt wurde:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.TextOut(X, Y, 'Hier!');{ Text bei (X, Y) ausgeben }
end;
```

Wenn die Anwendung läuft und Sie im Formular die Maustaste drücken, wird der String »Hier!« am angeklickten Punkt positioniert. Der folgende Quelltext weist der aktuellen Zeichenposition die Koordinaten zu, an denen der Benutzer die Maustaste gedrückt hat:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.MoveTo(X, Y);{ Stiftposition festlegen }
end;
```

Das Drücken der Maustaste legt hier die Stiftposition und somit den Ausgangspunkt der Linie fest. Um eine Linie zu einem Punkt zu zeichnen, an dem der Benutzer die Maustaste wieder losläßt, müssen Sie auf das Ereignis *OnMouseUp* reagieren.

Auf das Loslassen einer Maustaste reagieren

Ein *OnMouseUp*-Ereignis tritt immer dann ein, wenn der Benutzer eine Maustaste losläßt. Normalerweise wird dieses Ereignis an das Objekt weitergeleitet, das sich

beim Drücken der Maustaste unter dem Mauszeiger befindet. Dabei muß es sich nicht um dasselbe Objekt handeln, auf dem sich der Mauszeiger beim Loslassen der Maustaste befindet. Dies ermöglicht Ihnen beispielsweise, eine Linie so zu zeichnen, als ob sie jenseits der Formularbegrenzung fortgesetzt würde.

Um auf das Loslassen einer Maustaste zu reagieren, definieren Sie eine Behandlungsroutine für das Ereignis *OnMouseUp*.

Das folgende Beispiel zeigt eine einfache Behandlungsroutine für das Ereignis *OnMouseUp*. Es zeichnet eine Linie zu einem Punkt, an dem die Maustaste losgelassen wurde:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ Linie zeichnen von PenPos zu (X, Y) }
end;
```

Dieser Quelltext erlaubt dem Benutzer, durch Klicken, Ziehen und Loslassen der Maustaste Linien zu zeichnen. Die Linie wird für den Benutzer erst sichtbar, wenn er die Maustaste losläßt.

Auf das Bewegen der Maus reagieren

Das Ereignis *OnMouseMove* tritt auf, wenn der Benutzer die Maus bewegt. Das Ereignis wird dem Objekt übergeben, das sich unter dem Mauszeiger befindet, wenn die Maustaste gedrückt wird. Sie haben damit die Möglichkeit, dem Benutzer durch Zeichnen temporärer Linien die Mausbewegung anzuzeigen.

Um auf das Bewegen der Maus zu reagieren, definieren Sie eine Ereignisbehandlungsroutine für *OnMouseMove*. Das folgende Beispiel verwendet *OnMouseMove*-Ereignisse, um temporäre Formen auf einem Formular zu zeichnen, während der Anwender die Maustaste gedrückt hält. Der Anwender kann dadurch die Mausbewegung besser nachvollziehen. Die Behandlungsroutine für das Ereignis *OnMouseMove* zeichnet im Formular eine Linie zu der Position, an der das Ereignis aufgetreten ist:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ Linie zur aktuellen Position zeichnen }
end;
```

Die Bewegung der Maus über das Formular bewirkt das Zeichnen einer Linie, die dem Mauszeiger folgt, und zwar auch dann, wenn die Maustaste noch nicht gedrückt wurde.

OnMouseMove-Ereignisse treten auch dann ein, wenn die Maustaste nicht gedrückt wurde.

Wenn Sie abfragen möchten, ob eine Maustaste gedrückt wurde, müssen Sie dem Formularobjekt ein Objektfeld hinzufügen.

Felder einem Formularobjekt hinzufügen

Wenn Sie feststellen wollen, ob eine Maustaste gedrückt wurde, müssen Sie dem Formularobjekt ein Objektfeld hinzufügen. Sobald Sie ein Formular um eine Komponente erweitern, fügt Delphi dem Formularobjekt ein entsprechendes Feld hinzu, so daß Sie über den Namen des Feldes auf die Komponente zugreifen können. Sie können auch eigene Felder zu Formularen hinzufügen, indem Sie die Typdeklaration in der Formular-Unit entsprechend ändern.

Im folgenden Beispiel wird einem Formular ein Boolesches Feld hinzugefügt, dessen Wert auf *True* gesetzt wird, sobald der Benutzer eine Maustaste drückt.

Um einem Objekt ein Feld hinzuzufügen, ändern Sie die Typdefinition des Objekts, geben den Feldbezeichner an und schreiben dann nach der **public**-Anweisung am Ende der Deklaration weiter.

Alle Deklarationen vor der **public**-Anweisung »gehören« Delphi: An dieser Stelle werden Felder eingefügt, die Steuerelemente und Methoden vertreten, die auf Ereignisse reagieren.

Im folgenden Quelltext wird zunächst in der Objektdeklaration eines Formulars ein Boolesches Feld mit der Bezeichnung *Drawing* eingefügt. Danach werden zwei Felder vom Typ *TPoint* hinzugefügt (*Origin* und *MovePt*), die zum Speichern von Punktpositionen dienen.

```
type
  TForm1 = class(TForm)
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
 public
    Drawing: Boolean; { Feld, das angibt, ob eine Taste gedrückt wurde }
    Origin, MovePt: TPoint; { Feld zum Speichern von Punkten }
  end;
```

Das Feld *Drawing* gibt an, ob gerade gezeichnet wird. Es muß auf *True* gesetzt werden, wenn der Benutzer die Maustaste drückt, und auf *False*, wenn er sie losläßt:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True; { Das Zeichnen-Flag setzen }
 Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
 Drawing := False; { Das Zeichnen-Flag zurücksetzen }
end;
```

Anschließend kann in der Ereignisbehandlungsroutine für *OnMouseMove* festgelegt werden, daß nur gezeichnet wird, wenn *Drawing* den Wert *True* hat:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then { Nur Zeichnen, wenn das Zeichnen-Flag gesetzt ist }
   Canvas.LineTo(X, Y);
end;
```

Dies bewirkt, daß nur zwischen den Ereignissen *OnMouseDown* und *OnMouseUp* gezeichnet wird. Trotzdem erhalten Sie anstelle einer geraden eine »gekritzelte« Linie, die den Mausbewegungen direkt folgt.

Das Problem liegt in der Behandlungsroutine für das Ereignis *OnMouseMove,* die bei jeder Mausbewegung *LineTo* aufruft. Dadurch ändert sich die Stiftposition, und beim Loslassen der Maustaste ist der eigentliche Ausgangspunkt der Linie nicht mehr bekannt. Deshalb wird die Linie nicht gerade.

Verbesserte Liniendarstellung

Mit Hilfe von Feldern, in denen die verschiedenen Punkte zwischengespeichert werden, können Sie die Liniendarstellung in der Anwendung verbessern.

Festhalten des Ausgangspunktes

Beim Zeichnen von Linien muß der Ausgangspunkt der Linie im Feld *Origin* gespeichert werden.

Dem Feld *Origin* wird der Punkt zugewiesen, an dem das Ereignis *OnMouseDown* auftritt. Die Behandlungsroutine für das Ereignis *OnMouseUp* kann dann aus dem Inhalt von *Origin* den Ausgangspunkt ermitteln:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y); { Ausgangspunkt der Linie speichern }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y); { Stift auf Ausgangspunkt setzen }
 Canvas.LineTo(X, Y);
 Drawing := False;
end;
```

Nach diesen Änderungen zeichnet die Anwendung die abschließende Linie noch einmal. Der »Gummiband-Effekt«, d. h. das Zeichnen einer Hilfslinie zur Anzeige der Mausbewegung, wird aber noch nicht unterstützt.

Bewegungen zwischenspeichern

Das Problem bei der jetzigen Version der Behandlungsroutine für das Ereignis *OnMouseMove* ist, daß sie die Linie zur aktuellen Mausposition ausgehend von der letzten Mausposition zeichnet, nicht ausgehend von der Anfangsposition. Dieser Fehler

läßt sich aber einfach beseitigen, indem die Zeichenposition auf den Ausgangspunkt gesetzt und anschließend eine Linie zur aktuellen Mausposition gezeichnet wird:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y); { Stiftposition auf Ausgangspunkt setzen }
   Canvas.LineTo(X, Y);
  end;
end;
```

Auf diese Weise erhalten Sie Hilfslinien zur aktuellen Mausposition, die allerdings nicht mehr verschwinden und sich gegenseitig überdecken. Deshalb muß jede Linie gelöscht werden, bevor die nächste gezeichnet wird. Zu diesem Zweck ist es erforderlich, die Position der vorherigen Linien zwischenzuspeichern. Dazu dient das Feld *MovePt*.

MovePt muß auf den Endpunkt jeder Hilfslinie gesetzt werden, so daß Sie *MovePt* und *Origin* verwenden können, um diese Linie zu löschen, bevor die nächste gezeichnet wird:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);
 MovePt := Point(X, Y); { Bewegung speichern }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor; { XOR-Modus zum Zeichnen/Löschen verwenden }
   Canvas.MoveTo(Origin.X, Origin.Y); { Stift auf Ausgangspunkt zurücksetzen }
   Canvas.LineTo(MovePt.X, MovePt.Y); { Alte Linie löschen }
   Canvas.MoveTo(Origin.X, Origin.Y); { Stift wieder auf Ausgangspunkt setzen }
   Canvas.LineTo(X, Y); { Neue Linie zeichnen }
  end;
 MovePt := Point(X, Y); { Neuen Endpunkt speichern }
 Canvas.Pen.Mode := pmCopy;
end;
```

Sobald Sie nun eine Linie zeichnen, wird der erwünschte »Gummiband-Effekt« sichtbar. Der Wechsel in den Stiftmodus *pmNotXor* kombiniert die Hilfslinie mit den Hintergrund-Pixeln. Wenn Sie die Linie löschen, werden die Pixel wieder in ihren ursprünglichen Zustand versetzt. Nach dem Zeichnen der Hilfslinie wechseln Sie zurück in den Stiftmodus *pmCopy* (den Standardwert), um sicherzustellen, daß der Stift die endgültige Linie zeichnen kann, sobald die Maustaste losgelassen wird.

Mit Multimedia arbeiten

In Delphi können Sie Ihren Anwendungen Multimedia-Komponenten hinzufügen. Dazu stehen die Komponenten *TAnimate*! in der Registerkarte *Win32* und *TMediaplayer* in der Registerkarte *System* der Komponentenpalette zur Verfügung. Mit Hilfe von *TAnimate* können in einer Anwendung Videoclips ohne Ton abgespielt werden. Die Komponente *TMediaplayer* ermöglicht den Zugriff auf Audio- und / oder Videoclips.

Weitere Informationen zu *TAnimate* und *TMediaplayer* finden Sie in der Online-Hilfe.

Dieses Kapitel enthält Informationen zu folgenden Themen:

- Einer Anwendung Videoclips ohne Ton hinzufügen
- Einer Anwendung Audio- und / oder Videoclips hinzufügen

Einer Anwendung Videoclips ohne Ton hinzufügen

Mit Hilfe der Komponente *TAnimate* können Sie in einer Anwendung Videoclips ohne Ton abspielen.

So fügen Sie einer Anwendung einen Videoclip hinzu:

1. Doppelklicken Sie auf die Komponente *Animate* in der Registerkarte *Win32* der Komponentenpalette. Dadurch wird ein Steuerelement in das Formular eingefügt, in dem der Videoclip angezeigt werden kann.
2. Wählen Sie im Objektinspektor die Eigenschaft *Name* aus, und geben Sie einen neuen Namen für die Komponente ein (wobei Sie natürlich die Namenskonventionen für Delphi-Bezeichner beachten müssen). Dieser Name wird für den Zugriff auf das Steuerelement benötigt.

   Arbeiten Sie immer direkt im Objektinspektor, wenn Sie während des Entwurfs Eigenschaften festlegen und Ereignisbehandlungsroutinen erstellen.
3. Führen Sie anschließend eine der folgenden Aktionen durch:

   - Wählen Sie die Eigenschaft *Common AVI* aus, und klicken Sie in der Dropdown-Liste auf die gewünschte AVI-Option.
   - Wählen Sie die Eigenschaft *FileName* aus, und klicken Sie in der Wertespalte auf die Ellipsen-Schaltfläche (…). Wählen Sie im Dialogfeld *AVI öffnen* die gewünschte AVI-Datei, und klicken Sie anschließend auf *öffnen*.
   - Weisen Sie der Eigenschaft *ResName*~ oder *ResID* die Ressource mit dem Videoclip zu. Geben Sie anschließend mit *ResHandle* das Modul an, in dem sich die von *ResName* oder *ResID* angegebene Ressource befindet. Diese Eigenschaften stehen nicht im Objektinspektor zur Verfügung und müssen im Quelltext zugewiesen werden.

   Die AVI-Datei wird in den Speicher geladen. Wenn das erste Einzelbild der AVI-Datei bereits angezeigt werden soll, bevor der Clip mit Hilfe seiner Eigenschaft

Active oder seiner Methode *Play* abgespielt wird, setzen Sie die Eigenschaft *Open* auf *True*.

4 Legen Sie mit der Eigenschaft *Repetitions* fest, wie oft der Videoclip abgespielt werden soll. Beim Wert 0 wird der Clip so lange abgespielt, bis die Methode *Stop* der Komponente aufgerufen wird.

5 Legen Sie beliebige andere Einstellungen der *TAnimate*-Komponente fest. Soll beispielsweise die Wiedergabe mit einem anderen Einzelbild beginnen, weisen Sie der Eigenschaft *StartFrame* den gewünschten Wert zu.

6 Setzen Sie die Eigenschaft *Active* im Objektinspektor auf *True*, oder erstellen Sie eine Ereignisbehandlungsroutine, die den AVI-Clip bei einem bestimmten Ereignis zur Laufzeit startet. Soll der Clip beispielsweise beim Klicken auf eine Schaltfläche aktiviert werden, programmieren Sie diese Aktion in der *OnClick*-Ereignisbehandlungsroutine des Schalters. Sie können die Wiedergabe auch mit der Methode *Play* starten.

Hinweis Wenn Sie Änderungen am Formular oder seinen Komponenten vornehmen, nachdem Sie *Active* auf *True* gesetzt haben, erhält die Eigenschaft automatisch den Wert *False*, und die Aktivierung muß entweder vor dem Programmstart oder zur Laufzeit erneut durchgeführt werden.

Beispiel für das Hinzufügen von Videoclips ohne Ton

Nehmen wir an, Sie wollen beim Start Ihrer Anwendung einen Begrüßungsbildschirm mit einem animierten Logo anzeigen und ihn nach dem Abspielen des Clips wieder schließen.

Erstellen Sie zuerst ein neues Projekt. Speichern Sie die Datei UNIT1.PAS unter dem Namen FRMLOGO.PAS und die Datei PROJECT1.DPR unter LOGO.DPR. Fahren Sie dann folgendermaßen fort:

1 Doppelklicken Sie auf die Komponente *Animate* in der Registerkarte *Win32* der Komponentenpalette.

2 Geben Sie im Objektinspektor für die Eigenschaft *Name* den Wert *Logo1* ein.

3 Klicken Sie in der Wertespalte der Eigenschaft *FileName* auf die Ellipsen-Schaltfläche (…), um das Dialogfeld *AVI öffnen* anzuzeigen. Wechseln Sie in das Verzeichnis ..\DEMOS\COOLSTUF, wählen Sie die Datei COOL.AVI aus, und klicken Sie auf *Öffnen*.

Die Datei COOL.AVI wird in den Speicher geladen.

4 Verschieben Sie das Steuerelement mit der Maus in die rechte obere Ecke des Formulars.

5 Setzen Sie seine Eigenschaft *Repetitions* auf *5*.

6 Klicken Sie im Formular, um es zu aktivieren. Setzen Sie seine Eigenschaften *Name* und *Caption* auf *LogoForm1* bzw. *Logo-Fenster*. Verkleinern Sie nun das Formular, bis sich das Steuerelement etwa in der Mitte befindet.

7 Doppelklicken Sie auf das Formularereignis *OnActivate*. Nehmen Sie in die Ereignisbehandlungsroutine die folgende Anweisung auf, die den AVI-Clip abspielt, wenn das Formular aktiviert wird:

```
Logo1.Active := True;
```

8 Doppelklicken Sie auf die Komponente *Label* in der Registerkarte *Standard* der Komponentenpalette. Setzen Sie Ihre Eigenschaft *Caption* auf *Willkommen bei Cool Images 4.0*. Wählen Sie anschließend die Eigenschaft *Font* aus, und klicken Sie auf die Ellipsen-Schaltfläche (…) in der Wertespalte, um das Dialogfeld *Schriftart* zu öffnen. Nehmen Sie hier folgende Einstellungen vor: Schriftschnitt = Fett, Schriftgrad = 18, Farbe = Aquamarin. Bestätigen Sie mit OK. Positionieren Sie die Komponente in der Mitte des Formulars.

9 Klicken Sie auf die *Animate*-Komponente, um sie zu aktivieren. Doppelklicken Sie auf das Ereignis *OnStop*. Nehmen Sie in die Ereignisbehandlungsroutine die folgende Zeile auf, um das Formular nach dem Abspielen des Clips zu schließen:

```
LogoForm1.Close;
```

10 Wählen Sie *Start / Start*, oder klicken Sie in der Symbolleiste auf den grünen Pfeil, um die Anwendung zu starten.

Einer Anwendung Audio- und/oder Videoclips hinzufügen

Mit Hilfe der Komponente *TMediaPlayer* können Sie in Ihren Anwendungen auf Audio- und / oder Videoclips zugreifen. Die Komponente öffnet zuerst ein Mediengerät und führt dann verschiedene Operationen (Wiedergabe, Stop, Pause, Aufnahme usw.) mit den Clips durch. Mediengeräte können Hardware (z. B. ein DAT-Laufwerk) oder Software (z. B. ein Sequenzer) sein.

So fügen Sie einer Anwendung einen Audio- und / oder Videoclip hinzu:

1 Doppelklicken Sie auf die Komponente *MediaPlayer* in der Registerkarte *System* der Komponentenpalette.

2 Wählen Sie im Objektinspektor die Eigenschaft *Name* aus, und geben Sie einen neuen Namen für die Komponente ein (wobei Sie natürlich die Namenskonventionen für Delphi-Bezeichner beachten müssen). Dieser Name wird für den Zugriff auf das Steuerelement benötigt.

   Arbeiten Sie immer direkt im Objektinspektor, wenn Sie während des Entwurfs Eigenschaften festlegen und Ereignisbehandlungsroutinen erstellen.

3 Wählen Sie die Eigenschaft *DeviceType* aus, und klicken Sie in der Dropdown-Liste auf den gewünschten Gerätetyp (bei *dtAutoSelect* wird der Typ anhand der Namenserweiterung der mit *FileName* angegebenen Mediendatei ermittelt). Weitere Informationen zu den verschiedenen Gerätetypen und ihren Funktionen finden Sie in der nachfolgenden Tabelle.

4 Bei einem dateibasierten Mediengerät geben Sie in der Eigenschaft *FileName* den Namen der zu öffnenden Mediendatei an. Klicken Sie dazu auf die Ellipsen-Schaltfläche (...) in der Wertespalte, um das Dialogfeld *Öffnen* anzuzeigen. Wählen Sie hier die gewünschte Datei aus, und klicken Sie auf *öffnen*. Bei nicht dateibasier-

ten Mediengeräten muß zur Laufzeit das betreffende Medium (z. B. Diskette, Kassette, DAT usw.) eingelegt werden.

5 Setzen Sie die Eigenschaft *AutoOpen* auf *True*. Dadurch wird das angegebene Mediengerät automatisch geöffnet, wenn das Formular zur Laufzeit erstellt wird. Hat *AutoOpen* den Wert *False*, muß das Gerät explizit mit der Methode *Open* geöffnet werden.

6 Setzen Sie die Eigenschaft *AutoEnable* auf *True*. Dadurch werden die Schaltflächen des Steuerelements zur Laufzeit automatisch so aktiviert oder deaktiviert, wie es dem verwendeten Gerät entspricht. Sie können auch auf die Eigenschaft *EnabledButtons* doppelklicken und die Schaltflächen anschließend einzeln aktivieren (*True*) oder deaktivieren (*False*).

Der Benutzer kann zur Laufzeit mit den Schaltflächen der Komponente den Betriebsmodus des Multimediagerätes steuern (Wiedergabe, Pause, Stop usw.). Im Quelltext ist dies mit Hilfe der entsprechenden *Methoden* möglich.

7 Sie können das Steuerelement durch Ziehen mit der Maus oder mit den Ausrichtungsoptionen der Eigenschaft *Align* beliebig im Formular positionieren.

Die Komponente kann auch verwendet werden, wenn sie nicht in der Anwendung angezeigt wird. Setzen Sie dazu ihre Eigenschaft *Visible* auf *False*, und steuern Sie das Gerät mit den entsprechenden Methoden (*Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *Start Recording* und *Eject*).

8 Legen Sie beliebige andere Einstellungen der Medienwiedergabe-Komponente fest. Wenn Sie beispielsweise ein Anzeigefenster benötigen, weisen Sie der Eigenschaft *Display* das Steuerelement zu, in dem die Mediendaten angezeigt werden. Bei Geräten, die mehrere Spuren unterstützen, können Sie der Eigenschaft *Tracks* die gewünschte Spur zuweisen.

Tabelle 7.5 Gerätetypen und ihre Funktionen

Gerätetyp	Software/Hardware	Medium	Spuren	Anzeigefenster
dtAVIVideo	AVI-Wiedergabe für Windows	AVI-Dateien	Nein	Ja
dtCDAudio	CD-Wiedergabe für Windows oder ein CD-Laufwerk	Audio-CDs	Ja	Nein
dtDAT	DAT-Laufwerk	DAT	Ja	Nein
dtDigitalVideo	DV-Wiedergabe für Windows	AVI-, MPG- und MOV-Dateien	Nein	Ja
dtMMMovie	MM-Movie-Player	MM-Film	Nein	Ja
dtOverlay	Overlay-Gerät	Analogvideo	Nein	Ja
dtScanner	Scanner	Wiedergabe wird nicht unterstützt (bei Aufnahme wird gescannt)	Nein	Nein
dtSequencer	MIDI-Sequenzer für Windows	MIDI-Dateien	Ja	Nein

Tabelle 7.5 Gerätetypen und ihre Funktionen (Fortsetzung)

Gerätetyp	Software/Hardware	Medium	Spuren	Anzeigefenster
dtVCR	Videokassettenrecorder	Videokassetten	Nein	Ja
dtWaveAudio	Wave-Audio-Player für Win dows	WAV-Dateien	Nein	Nein

Beispiel für das Hinzufügen von Audio- und/oder Videoclips

Im folgenden Beispiel wird der AVI-Videoclip einer Multimedia-Werbung für Delphi abgespielt. Erstellen Sie zunächst ein neues Projekt, um dieses Beispiel auszuführen. Speichern Sie dann die Datei UNIT1.PAS unter dem Namen FRMAD.PAS und die Datei PROJECT1.DPR unter DELPHIAD.DPR. Fahren Sie dann folgendermaßen fort:

1. Doppelklicken Sie auf die Komponente *MediaPlayer* in der Registerkarte *System* der Komponentenpalette.
2. Geben Sie im Objektinspektor für die Eigenschaft *Name* den Wert *VideoPlayer1* ein.
3. Wählen Sie die Eigenschaft *DeviceType* aus, und klicken Sie in der Dropdown-Liste auf *dtAVIVideo.*
4. Klicken Sie bei der Eigenschaft *FileName* auf die Ellipsen-Schaltfläche (...) in der Wertespalte, um das Dialogfeld *Öffnen* anzuzeigen. Wechseln Sie hier in das Verzeichnis ..\DEMOS\COOLSTUF, und wählen Sie die Datei SPEEDIS.AVI aus. Klicken Sie auf *Öffnen.*
5. Setzen Sie *AutoOpen* auf *True* und *Visible* auf *False.*
6. Doppelklicken Sie auf die Komponente *Animate* in der Registerkarte *Win32* der Komponentenpalette. Diese Komponente wird als Anzeigefenster für den Videoclip verwendet. Setzen Sie *AutoSize* auf *False*, *Height* auf 175 und *Width* auf 200. Ziehen Sie das Steuerelement in die linke obere Ecke des Formulars.
7. Klicken Sie auf die *MediaPlayer*-Komponente, um sie wieder zu aktivieren. Wählen Sie für die Eigenschaft *Display* das Objekt *Animate1* aus der Dropdown-Liste.
8. Aktivieren Sie nun das Formular. Geben Sie für seine Eigenschaft *Name* den Wert Delphi_Ad ein. Passen Sie anschließend die Größe des Formulars an die Größe der Animate-Komponente an.
9. Doppelklicken Sie auf das Formularereignis *OnActivate.* Nehmen Sie in die Ereignisbehandlungsroutine die folgende Anweisung auf, die den AVI-Clip abspielt, wenn das Formular aktiviert wird:

   ```
   Videoplayer1.Play;
   ```

10. Wählen Sie *Start / Start*, um den AVI-Clip wiederzugeben.

Kapitel

8

Multithread-Anwendungen entwickeln

Die VCL bietet mehrere Objekte, welche die Entwicklung von Multithread-Anwendungen erleichtern. Multithread-Anwendungen zeichnen sich dadurch aus, daß sie aus mehreren gleichzeitig ausgeführten Ablaufsträngen bestehen, den sogenannten *Threads*. Die Realisierung mehrerer solcher Threads erfordert zwar eine gewisse Sorgfalt bei der Planung, führt aber zu deutlichen Verbesserungen in Programmen:

- **Flaschenhälse werden vermieden.** In einem Programm mit nur einem Thread muß die gesamte Ausführung unterbrochen werden, wenn auf den Abschluß eines langsamen Prozesses gewartet wird. Solche Prozesse sind das Schreiben von Dateien auf Datenträger, die Kommunikation mit anderen Rechnern oder die Anzeige von Multimediadaten. Die CPU befindet sich bis zum Ende solcher Prozesse im Leerlauf. Wenn eine Anwendung dagegen mehrere Threads umfaßt, kann die Ausführung der anderen Threads fortgesetzt werden, während das Ergebnis eines langsamen Prozesses noch nicht vorliegt.
- **Strukturierung des Programmverhaltens.** Oft läßt sich das Verhalten eines Programms in mehreren parallelen Prozessen organisieren, die unabhängig voneinander funktionieren. Mit Hilfe von Threads kann die Ausführung eines bestimmten Quelltextabschnitts für jeden dieser parallelen Zweige gleichzeitig gestartet werden. Ebenso können verschiedenen Programm-Tasks unterschiedliche Prioritäten zugeordnet werden, damit kritische Tasks mehr CPU-Zeit erhalten.
- **Multiprocessing.** Wenn eine Anwendung auf einem System mit mehreren Prozessoren läuft, kann die Leistung gesteigert werden, indem man die Verarbeitung auf einzelne Threads verteilt und diese gleichzeitig auf verschiedenen Prozessoren ausführen läßt.

Hinweis Nicht alle Betriebssysteme bieten echtes Multiprocessing, auch wenn die zugrundeliegende Hardware dieses Konzept unterstützt. So wird beispielsweise unter Windows 95 Multiprocessing auch dann nur simuliert, wenn die verwendete Hardware Multiprocessing unterstützt.

Thread-Objekte definieren

In den meisten Anwendungen wird ein Ausführungs-Thread durch ein Thread-Objekt repräsentiert. Solche Objekte vereinfachen die Entwicklung von Multithread-Anwendungen, indem sie die gängigsten Einsatzmöglichkeiten von Threads kapseln.

Hinweis Thread-Objekte bieten keine Kontrolle über die Sicherheitsattribute oder die Stack-Größe der Threads. Um auf diese Faktoren Einfluß zu nehmen, verwenden Sie die Funktion *BeginThread*. Einige der im Abschnitt »Threads koordinieren« auf Seite 8-7 beschriebenen Objekte und Methoden zur Thread-Synchronisierung können zusammen mit *BeginThread* eingesetzt werden. Eine ausführliche Beschreibung von *BeginThread* finden Sie in der Online-Hilfe.

Um ein Thread-Objekt in einer Anwendung zu implementieren, erzeugen Sie einen neuen Nachkommen von *TThread*. Wählen Sie dazu zunächst im Hauptmenü den Befehl *Datei / Neu*, und doppelklicken Sie anschließend im Dialogfeld *Objektgalerie* auf das Symbol *Thread-Objekt*. Sie werden aufgefordert, einen Klassennamen für das neue Thread-Objekt einzugeben. Nach Angabe des Namens erstellt Delphi eine neue Unit-Datei, um den Thread zu implementieren.

Hinweis Im Gegensatz zu den meisten Dialogfeldern der IDE, in denen die Eingabe eines Klassennamens erforderlich ist, wird im Dialogfeld *Neues Thread-Objekt* dem eingegebenen Klassennamen nicht automatisch der Buchstabe *T* vorangestellt.

Die automatisch generierte Datei enthält das Quelltextgerüst für das neue Thread-Objekt. Für einen Thread mit dem Namen *TMyThread* sieht die Datei beispielsweise folgendermaßen aus:

```
unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Private-Deklarationen }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Hier fügen Sie den Quelltext für den Thread ein }
end;
end.
```

Ihre Aufgabe ist es, den Quelltext für die Methode *Execute* einzufügen. Wie Sie dabei vorgehen, wird in den folgenden Abschnitten erläutert.

Den Thread initialisieren

Um den Quelltext zur Initialisierung der neuen Thread-Klasse zu erstellen, überschreiben Sie die Methode *Create*. Fügen Sie einen neuen Konstruktor zur Deklaration der Thread-Klasse hinzu, und initialisieren Sie in dessen Implementierung den Thread. Sie können hier z.B. die Standard-Priorität des Threads festlegen und angeben, ob der Thread nach seiner Ausführung automatisch freigegeben werden soll.

Eine Standard-Priorität zuweisen

Die Priorität ist maßgebend für den Anteil der CPU-Zeit, die dem Thread vom Betriebssystem zugeordnet wird. Threads mit hoher Priorität übernehmen zeitkritische Aufgaben. Zur Festlegung der Priorität dient die Eigenschaft *Priority*. Die Werte dieser Eigenschaft liegen auf einer siebenstufigen Skala, wie Tabelle 8.1 zeigt.

Tabelle 8.1 Prioritätsstufen für Threads

Wert	Bedeutung
tpIdle	Der Thread wird nur dann ausgeführt, wenn sich das System im Leerlauf befindet. Windows unterbricht niemals einen anderen Thread zugunsten eines Threads mit der Prioritätsstufe *tpIdle*.
tpLowest	Die Priorität des Threads liegt zwei Punkte unter der normalen Stufe.
tpLower	Die Priorität des Threads liegt einen Punkt unter der normalen Stufe.
tpNormal	Der Thread besitzt normale Priorität.
tpHigher	Die Priorität des Threads liegt eine Stufe über der normalen Stufe.
tpHighest	Die Priorität des Threads liegt zwei Stufen über der normalen Stufe.
tpTimeCritical	Der Thread erhält höchste Priorität.

Warnung Wenn eine rechenintensive Operation hohe Priorität erhält, kann dies die Ausführung der anderen Threads in der Anwendung blockieren. Daher sollten nur solche Threads höchste Prioritätsstufen erhalten, die den Großteil der Zeit auf externe Ereignisse warten.

Der folgende Quelltext zeigt den Konstruktor eines Threads mit niedriger Priorität, der im Hintergrund Verarbeitungsschritte ausführen soll, ohne die Leistung der übrigen Anwendung zu vermindern:

```
constructor TMyThread.Create(CreateSuspended: Boolean);
{
  inherited Create(CreateSuspended);
  Priority := tpIdle;
}
```

Den Freigabezeitpunkt von Threads festlegen

Sobald Threads ihre Operation beendet haben, können sie in der Regel einfach freigegeben werden. In solchen Fällen ist es am einfachsten, die Freigabe dem Thread-Objekt selbst zu überlassen. Dazu setzen Sie die Eigenschaft *FreeOnTerminate*auf *True*.

Es gibt nun aber Situationen, in denen die Beendigung eines Threads mit anderen Threads koordiniert werden muß. So kann es beispielsweise sein, daß Sie den Ergebniswert eines Threads abwarten müssen, bevor ein anderer Thread ausgeführt werden kann. Sie werden dann kaum die Absicht haben, den ersten Thread freizugeben, bevor der zweite Thread den Ergebniswert erhalten hat. Diese Situation läßt sich handhaben, indem Sie *FreeOnTerminate* auf *False* setzen und dann den ersten Thread explizit freigeben.

Die Thread-Funktion schreiben

Die Methode *Execute* fungiert als Thread-Funktion. Man kann sich diese Methode als Programm vorstellen, das von Ihrer Anwendung gestartet wird. Der Unterschied liegt nur darin, daß beide denselben Prozeßbereich verwenden. Das Schreiben einer Thread-Funktion gestaltet sich allerdings ein wenig komplexer als das Schreiben eines separaten Programms. Es muß sichergestellt sein, daß kein Speicher überschrieben wird, der im Zugriff anderer Threads der Anwendung liegt. Andererseits kann über den gemeinsam genutzten Speicher die Kommunikation zwischen den Threads ablaufen.

Der VCL-Haupt-Thread

Wenn auf Objekte der VCL-Objekthierarchie zurückgegriffen wird, ist nicht garantiert, daß deren Eigenschaften und Methoden thread-sicher sind. Das heißt, daß beim Zugriff auf Eigenschaften oder beim Ausführen von Methoden bestimmte Aktionen stattfinden, die auf Speicher zugreifen, der nicht vor dem Zugriff anderer Threads geschützt ist. Aus diesem Grund ist ein VCL-Haupt-Thread für den Zugriff von VCL-Objekten separiert. Dieser Thread behandelt alle Windows-Botschaften, die von den Komponenten Ihrer Anwendung stammen.

Indem alle Objekte nur innerhalb dieses einen Threads auf ihre Eigenschaften zugreifen und ihre Methoden ausführen, werden Interferenzen unter den Objekten ausgeschlossen. Um den VCL-Haupt-Thread zu verwenden, erstellen Sie eine eigene Routine, welche die nötigen Aktionen übernimmt. Rufen Sie diese Routine dann in der Methode *Synchronize* des Threads auf. Beispiel:

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;
⋮
procedure TMyThread.Execute;
begin
⋮
  Synchronize(PushTheButton);
⋮
end;
```

Die Methode *Synchronize* wartet, bis der VCL-Haupt-Thread in die Botschaftsschleife eintritt, und führt dann die übergebene Methode aus.

Hinweis Da die Methode *Synchronize* die Botschaftsschleife verwendet, kann sie nicht in Konsolenanwendungen eingesetzt werden. In Konsolenanwendungen muß deshalb der Zugriffsschutz für VCL-Objekte mit anderen Mechanismen implementiert werden (z.B. mit kritischen Abschnitten).

Der VCL-Haupt-Thread braucht allerdings nicht immer verwendet zu werden. Einige Objekte sind thread-sensitiv. Wenn bekannt ist, daß die Methoden eines Objekts thread-sicher sind, kann auf die Methode *Synchronize* verzichtet und die Ausführung beschleunigt werden. Es muß dann nämlich nicht gewartet werden, bis der VCL-Thread in die Botschaftsschleife eintritt. Die Methode *Synchronize* wird in den folgenden Fällen nicht benötigt:

- Datenzugriffskomponenten sind thread-sicher, wenn jeder Thread über eine eigene Datenbanksitzung verfügt. Die einzige Ausnahme sind Treiber für die Datenbank Access. Diese Treiber sind in die ADO-Bibliothek von Microsoft integriert, und diese ist nicht thread-sicher.

  Auch bei der Verwendung von Datenzugriffskomponenten müssen alle Aufrufe, die sich auf datensensitive Steuerelemente beziehen, in der Methode *Synchronize* gekapselt werden. Beispielsweise müssen alle Aufrufe gekapselt werden, die ein Datensteuerelement über die Eigenschaft *DataSet* der Datenquelle mit einer Datenmenge verknüpfen. Dagegen braucht der Zugriff auf die Daten in einem Feld der Datenmenge nicht gekapselt zu werden.

  Weitere Informationen zu Datenbanksitzungen und Threads finden Sie unter »Mehrere Sitzungen verwalten« auf Seite 16-17.

- Grafikobjekte sind thread-sicher. Sie benötigen den VCL-Haupt-Thread, um auf die folgenden Objekte zuzugreifen: *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile* und *TIcon*. Canvas-Objekte (Zeichenflächen) können außerhalb der Methode *Synchronize* verwendet werden, indem man die Objekte sperrt (siehe »Objekte sperren« auf Seite 8-7).
- Da Listenobjekte grundsätzlich nicht thread-sicher sind, steht die thread-sichere Version *TThreadList* bereit, die Sie anstelle von *TList* verwenden können.

Lokale Thread-Variablen verwenden

Die Thread-Funktion und alle von ihr aufgerufenen Routinen besitzen ebenso, wie alle anderen Object-Pascal-Routinen eigene lokale Variablen. Diese Routinen können außerdem auf globale Variablen zugreifen. Das ist die Basis für einen leistungsfähigen Mechanismus zur Kommunikation zwischen den Threads.

In bestimmten Situationen müssen Variablen verwendet werden, die für alle Routinen in einem Thread global sind, aber nicht mit anderen Instanzen derselben Thread-Klasse gemeinsam genutzt werden sollen. Dies läßt sich über die Deklaration von Variablen erreichen, die innerhalb des Threads lokal sind. Lokale Thread-Variablen werden in einem **threadvar**-Abschnitt deklariert:

```
threadvar
  x : integer;
```

Hier wird eine Integer-Variable deklariert, die aus Sicht anderer Threads der Anwendung als privat gilt, jedoch innerhalb des Threads global sichtbar ist.

Der **threadvar**-Abschnitt kann nur für globale Variablen benutzt werden. Die Verwendung von Zeigern und Funktionen als Thread-Variablen ist nicht zulässig. Auch Typen, bei denen mit einer Copy-on-write-Semantik gearbeitet wird, wie etwa lange Strings, sind als Thread-Variablen nicht verwendbar.

Die Beendigung mit anderen Threads prüfen

Ein Thread-Objekt startet, sobald die Methode *Execute* ausgeführt wird (siehe »Thread-Objekte ausführen« auf Seite 8-11). Der Thread bleibt aktiv, solange die Methode *Execute* läuft. Dahinter steht das Prinzip, daß der Thread eine bestimmte Aufgabe ausführt und nach deren Erledigung stoppt. Zuweilen ist es allerdings notwendig, daß ein Thread beendet wird, wenn eine bestimmte externe Bedingung erfüllt ist.

Es ist möglich, Threads so zu konfigurieren, daß sie den Zeitpunkt anzeigen, zu dem ein anderer Thread seine Ausführung beenden soll. Diese Signalisierung erfolgt über die Eigenschaft *Terminated*. Wenn ein Thread versucht, einen anderen zur Beendigung zu veranlassen, ruft er die Methode *Terminate* dieses anderen Threads auf. *Terminate* setzt die Eigenschaft des Threads, der beendet werden soll, auf *True*. Nun liegt es bei der *Execute*- Methode dieses Threads, die Methode *Terminate* so zu implementieren, daß sie die Eigenschaft *Terminated* abfragt und entsprechend reagiert. Das folgende Beispiel zeigt eine solche Möglichkeit:

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

Clean-up-Quelltext schreiben

Der Quelltext, der nach Beendigung eines Threads die nötigen Aufräumarbeiten übernimmt (sogenannter Clean-up-Quelltext), kann zentralisiert werden. Kurz vor Abschluß des Threads wird ein Ereignis *OnTerminate* ausgelöst. Am besten wird sämtlicher Clean-up-Quelltext in der Ereignisbehandlungsroutine für *OnTerminate* plaziert. Nur dann ist sichergestellt, daß er immer ausgeführt wird, unabhängig von dem Ausführungspfad, der auf die Methode *Execute* folgt.

Die Ereignisbehandlungsroutine für *OnTerminate* wird nicht als Teil des Threads ausgeführt. Statt dessen läuft sie im Kontext des VCL-Haupt-Threads der Anwendung. Das impliziert folgende Bedingungen:

- Lokale Thread-Variablen können in einer Ereignisbehandlungsroutine für *OnTerminate* nicht verwendet werden, es sei denn, es sind die Werte des VCL-Haupt-Threads verlangt.
- Von der Ereignisbehandlungsroutine für *OnTerminate* kann sicher auf sämtliche Komponenten und VCL-Objekte zugegriffen werden, ohne sich um eventuelle Kollisionen mit anderen Threads zu kümmern.

Details zum VCL-Haupt-Thread finden Sie unter »Der VCL-Haupt-Thread« auf Seite 8-4.

Threads koordinieren

Während der Entwicklung des Quelltextes, der beim Start des Threads ausgeführt werden soll, muß das Verhalten der anderen Threads berücksichtigt werden, die eventuell gleichzeitig laufen. Insbesondere muß verhindert werden, daß zwei Threads gleichzeitig auf dasselbe globale Objekt oder dieselbe globale Variable zugreifen. Außerdem kann es sein, daß der Quelltext eines Threads von den Ausführungsergebnissen eines anderen Threads abhängt.

Gleichzeitigen Zugriff vermeiden

Um beim Zugriff auf globale Objekte oder Variablen eine Kollision mit anderen Threads zu vermeiden, muß die Ausführung anderer Threads so lange blockiert werden, bis der Quelltext des aktuellen Threads eine Operation abgeschlossen hat. Dabei ist große Sorgfalt nötig, damit andere Threads nicht unnötig gesperrt werden. Dies würde die Systemleistung herabsetzen und die Vorteile zunichte machen, die sich aus dem Einsatz mehrerer Threads ergeben.

Objekte sperren

Einige Objekte verfügen über einen integrierten Sperrmechanismus, der die Ausführung anderer Threads verhindert.

Beispielsweise besitzen Zeichenflächenobjekte (*TCanvas* und seine Nachkommen) eine Methode namens *Lock* , die den Zugriff anderer Threads auf die Zeichenfläche erst dann gestattet, wenn die Methode *Unlock* aufgerufen wird.

Die Objekthierarchie der VCL enthält außerdem das thread-sichere Listenobjekt *TThreadList*. Ein Aufruf von *TThreadList.LockList* gibt einerseits das Listenobjekt zurück und hindert andererseits weitere Threads am Zugriff auf die Liste, bis die Methode *UnlockList* aufgerufen wird. Aufrufe von *TCanvas.Lock* oder *TThreadList.LockList* können sicher geschachtelt werden. Die Sperre wird erst dann wieder aufgehoben, wenn sämtliche Aufrufe von *LockList* innerhalb desselben Threads durch einen Aufruf von *UnlockList* neutralisiert wurden.

Kritische Abschnitte

Wenn ein Objekt nicht über integrierte Sperrmechanismen verfügt, kann auf einen sogenannten kritischen Abschnitt ausgewichen werden. Kritische Abschnitte funktionieren wie Schranken, die immer nur einen einzigen Thread passieren lassen. Für den praktischen Einsatz muß eine globale Instanz von *TCriticalSection* erzeugt werden. *TCriticalSection* besitzt zwei Methoden: *Acquire* (verhindert, daß andere Threads den Abschnitt ausführen) und *Release* (entfernt den Block).

Jeder kritische Abschnitt ist dem globalen Speicher zugeordnet, der geschützt werden soll. Vor dem Zugriff auf diesen globalen Speicher muß der Thread mit der Methode *Acquire* sicherstellen, daß kein anderer Thread den Speicher verwendet. Nach Ausführung der Operationen ruft der Thread die Methode *Release* auf, so daß wie-

derum andere Threads mit einem Aufruf von *Acquire* auf den globalen Speicher zugreifen können.

Wichtiger Hinweis

Kritische Abschnitte erreichen nur dann die vorgesehene Wirkung, wenn sie von jedem Thread für den Zugriff auf den verknüpften globalen Speicher verwendet werden. Sobald es Threads gibt, welche den kritischen Abschnitt ignorieren und ohne Aufruf von *Acquire* auf den globalen Speicher zugreifen, können sich Kollisionen ergeben.

Angenommen, in einer Anwendung ist zur Realisierung eines kritischen Abschnitts die globale Variable *LockXY* eingerichtet, welche den Zugriff auf die globalen Variablen X und Y verhindern soll. Jeder Thread, der *X* oder *Y* verwendet, muß indirekt vorgehen, also den kritischen Abschnitt aufrufen:

```
LockXY.Acquire; { Andere Threads blockieren }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

TMultiReadExclusiveWriteSynchronizer-Objekte

Wenn der Schutz des globalen Speichers mit einem kritischen Abschnitt implementiert wird, kann immer nur ein Thread auf den geschützten Speicher zugreifen. Ein solcher umfassender Schutz ist aber häufig gar nicht nötig. Ein typisches Beispiel dafür sind Objekte oder Variablen, auf die zwar viele Lesezugriffe durchgeführt werden, denen aber nur sehr selten ein Wert zugewiesen wird. Wenn mehrere Threads gleichzeitig Lesezugriffe auf denselben Speicherbereich ausführen, besteht keine Gefahr eines Konflikts. Kollisionen sind nur möglich, wenn ein Thread einen Schreibzugriff ausführt.

Ein globaler Speicherbereich, auf den zwar verschiedene Threads oft Lesezugriffe ausführen, in den aber nur selten geschrieben wird, kann mit *TMultiReadExclusiveWriteSynchronizer* geschützt werden. Dieses Objekt agiert wie ein kritischer Abschnitt, der mehreren Threads das Lesen des geschützten Speichers gestattet, solange kein Thread in diesen Speicher schreibt. In den durch *TMultiReadExclusiveWriteSynchronizer* geschützten Speicher können nur Threads schreiben, die über einen exklusiven Zugriff verfügen.

Um einen globalen Speicherbereich auf diese Weise zu schützen, erzeugen Sie eine globale Instanz von *TMultiReadExclusiveWriteSynchronizer* und verknüpfen diese mit dem betreffenden Speicher. Jeder Thread, der aus diesem Speicher lesen will, muß die Methode *BeginRead* aufrufen. *BeginRead* stellt sicher, daß kein anderer Thread gerade in diesen Speicher schreibt. Wenn ein Thread den Lesezugriff auf den geschützten Speicher beendet, ruft er die Methode *EndRead* auf. Wenn ein Thread in den geschützten Speicher schreiben will, muß er vorher die Methode *BeginWrite* aufrufen. Damit wird sichergestellt, daß kein anderer Thread gerade einen Lese- oder Schreibzugriff auf den Speicher ausführt. Ein Thread beendet seinen Schreibzugriff auf den geschützten Speicher durch einen Aufruf der Methode *EndWrite*. Erst dann können andere Threads wieder lesend auf den Speicher zugreifen.

Wichtiger Hinweis Wie ein kritischer Abschnitt erreicht auch ein *TMultiReadExclusiveWriteSynchronizer*-Objekt nur dann die gewünschte Wirkung, wenn er von allen Threads zum Zugriff auf den globalen Speicher verwendet wird. Sobald es Threads gibt, die das *TMultiReadExclusiveWriteSynchronizer*-Objekt ignorieren und ohne einen Aufruf von *BeginRead* oder *BeginWrite* auf den globalen Speicher zugreifen, können sich Kollisionen ergeben.

Weitere Techniken für die gemeinsame Nutzung von Speicher

Bei der Verwendung von Objekten in der VCL-Objekthierarchie sollte der Quelltext immer im VCL-Haupt-Thread ausgeführt werden. Damit ist sichergestellt, daß das Objekt nicht indirekt auf Speicherbereiche zugreift, die auch von VCL-Objekten in anderen Threads verwendet werden. Weitere Informationen über den VCL-Haupt-Thread finden Sie unter »Der VCL-Haupt-Thread« auf Seite 8-4.

Wenn der globale Speicher nicht von mehreren Threads gemeinsam genutzt werden muß, sollten Sie lokale Thread-Variablen anstelle von globalen Variablen in Erwägung ziehen. Threads müssen dann keine anderen Threads sperren oder auf deren Beendigung warten. Details zu diesen Variablen finden Sie unter »Lokale Thread-Variablen verwenden« auf Seite 8-5.

Die Ausführung anderer Threads abwarten

Wenn die Ausführung eines Threads ausgesetzt werden muß, bis ein anderer die vorgesehenen Aktionen durchgeführt hat, sind verschiedene Methoden verfügbar. Sie können abwarten, bis ein anderer Thread vollständig abgeschlossen ist oder bis ein anderer Thread signalisiert, daß er eine bestimmte Aufgabe ausgeführt hat.

Warten, bis ein Thread vollständig ausgeführt ist

Wenn die vollständige Ausführung eines anderen Threads abgewartet werden muß, wird die Methode *WaitFor* dieses anderen Threads aufgerufen. *WaitFor* wird erst beendet, wenn der andere Thread beendet ist, das heißt, entweder nach Ausführung seiner Methode *Execute* oder nach einer Exception. Im folgenden Beispiel wird gewartet, bis ein anderer Thread ein Thread-Listenobjekt eingefügt hat. Erst dann erfolgt der Zugriff auf die Objekte in der Liste:

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
  end;
  ThreadList1.UnlockList;
end;
```

In diesem Beispiel wird nur dann auf die Listenelemente zugegriffen, wenn die Methode *WaitFor* signalisiert hat, daß die Liste erfolgreich gefüllt wurde. Dieser Rückgabewert muß von der *Execute*-Methode des Threads zugewiesen werden, auf den gewartet wurde. Da das Ergebnis der Thread-Ausführung allerdings nur für die

Threads relevant ist, welche die Methode *WaitFor* aufrufen, und nicht für Quelltext, der *Execute* aufruft, gibt die Methode *Execute* keinen Wert zurück. Statt dessen setzt sie die Eigenschaft *ReturnValue*. *ReturnValue* wird von der Methode *WaitFor* zurückgeliefert, wenn diese von anderen Threads aufgerufen wird. Die Rückgabewerte sind Integer, deren Bedeutung von der Anwendung bestimmt wird.

Warten, bis eine Aufgabe ausgeführt ist

Es gibt Situationen, in denen nicht gewartet werden muß, bis ein Thread vollständig ausgeführt ist, sondern nur solange, bis er eine bestimmte Operation abgeschlossen hat. In solchen Fällen finden Ereignisobjekte Verwendung. Diese Objekte (*TEvent*) sollten mit globalem Gültigkeitsbereich erzeugt werden, so daß sie wie Signale funktionieren, die von jedem Thread erkannt werden können.

Wenn in einem Thread eine Operation abgeschlossen wurde, auf der andere Threads basieren, wird *TEvent.SetEvent* aufgerufen. *SetEvent* aktiviert das Signal, so daß andere Threads, welche dieses Signal auswerten, den Abschluß einer Operation feststellen können. Zur Deaktivierung des Signals verwenden Sie die Methode *ResetEvent*.

Das folgende Beispiel basiert auf einer Situation, in der nicht nur auf die Ausführung eines einzelnen Threads, sondern auf die vollständige Ausführung mehrerer Threads gewartet werden muß. Da nicht bekannt ist, welcher Thread zuletzt beendet wird, kann nicht einfach die Methode *WaitFor* eines der Threads aufgerufen werden. Statt dessen wird ein Zähler implementiert, den jeder Thread bei seiner Beendigung erhöht. Der letzte Thread signalisiert dann durch das Auslösen eines Ereignisses, daß die Ausführung aller Threads abgeschlossen ist.

Nachfolgend finden Sie den letzten Teil der Ereignisbehandlungsroutine für *OnTerminate*. Sie wird für alle Threads aufgerufen, die beendet werden müssen. *CounterGuard* ist ein globaler, kritischer Abschnitt, der den gleichzeitigen Zugriff mehrerer Threads auf den Zähler verhindert. *Counter* ist eine globale Variable, in der die Zahl der Threads gespeichert wird, die vollständig ausgeführt wurden.

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
  ...
  CounterGuard.Acquire; { Zähler mit einer Sperre belegen }
  Dec(Counter);   { Wert der globalen Zähler-Variable verringern }
  if Counter = 0 then
    Event1.SetEvent; { Signalisieren, ob es sich um den letzten Thread handelt }
  CounterGuard.Release; { Sperre vom Zähler entfernen }
  ...
end;
```

Andere Threads fragen über die Methode *WaitFor* den Signalzustand ab. *WaitFor* wartet eine bestimmte Zeit, ob das Signal gesetzt wird, und gibt einen der Werte aus der folgenden Tabelle zurück.

Tabelle 8.2 Rückgabewerte der Methode WaitFor

Wert	Bedeutung
wrSignaled	Das Signal des Ereignisses war gesetzt.
wrTimeout	Die festgelegte Zeitspanne ist verstrichen, ohne daß das Signal gesetzt wurde.

Tabelle 8.2 Rückgabewerte der Methode WaitFor (Fortsetzung)

Wert	Bedeutung
wrAbandoned	Das Ereignisobjekt wurde freigegeben, bevor das Timeout-Intervall abgelaufen war.
wrError	Während der Wartezeit ist ein Fehler aufgetreten.

Der folgende Quelltext zeigt, wie der Haupt-Thread die einzelnen Aufgaben-Threads startet und nach deren Beendigung seine Operation wieder aufnimmt:

```
Event1.ResetEvent; { Ereignis vor der Ausführung der Threads zurücksetzen }
for i := 1 to Counter do
  TaskThread.Create(False); { Aufgaben-Threads erzeugen und ausführen }
if Event1.WaitFor(20000) != wrSignaled then
  raise Exception;
{ Alle Aufgaben-Threads sind beendet. Jetzt den Haupt-Thread fortsetzen }
```

Hinweis Wenn die Wartezeit nicht durch einen festen Wert beschränkt sein soll, übergeben Sie der Methode *WaitFor* den Parameterwert INFINITE. Allerdings ist hier Vorsicht geboten, denn wenn das Signal aus irgendwelchen Gründen nie empfangen wird, bleibt der Thread einfach stehen.

Thread-Objekte ausführen

Nach der Implementierung einer Thread-Klasse, das heißt, nach der Ausstattung mit einer *Execute*-Methode, kann die Thread-Klasse in der Anwendung zum Ausführen der Methode *Execute* verwendet werden. Zuerst ist eine Instanz der Thread-Klasse erforderlich. Es lassen sich Instanzen erzeugen, die sofort starten, und solche, die sich so lange in einem passiven Zustand befinden, bis die Methode *Resume* aufgerufen wird. Wenn der erzeugte Thread sofort starten soll, setzen Sie den Parameter *CreateSuspended* des Konstruktors wie in der folgenden Zeile auf *False*:

```
SecondProcess := TMyThread.Create(false); { Thread erzeugen und starten }
```

Wichtiger Hinweis Erzeugen Sie nicht zu viele Threads in einer Anwendung. Der Aufwand für die Koordination kann die Systemleistung herabsetzen. Auf einem Rechner mit Einzelprozessor sind 16 Threads pro Prozeß die empfohlene Obergrenze. Voraussetzung ist, daß die meisten Threads auf externe Ereignisse warten. Wenn alle Threads aktiv sind, muß die Anzahl reduziert werden.

Zur Ausführung von parallelem Quelltext lassen sich mehrere Instanzen desselben Thread-Typs erzeugen. Dies könnte zum Beispiel als Reaktion auf eine bestimmte Benutzeraktion erfolgen, damit jeder Thread die erwartete Antwort generieren kann.

Die Standard-Priorität überschreiben

Wenn sich der Anteil an CPU-Zeit, den ein Thread erhält, direkt aus der Aufgabe des Threads ergibt, wird die Priorität im Konstruktor festgelegt. Details dazu enthält der Abschnitt »Den Thread initialisieren« auf Seite 8-3. Wenn die Thread-Priorität allerdings vom Zeitpunkt der Ausführung abhängt, wird der Thread in einem passiven,

unterbrochenen Zustand erzeugt und die Priorität erst vor dem Start des Threads festgelegt:

```
SecondProcess :=  TMyThread.Create(True); { Thread erzeugen, aber nicht starten }
SecondProcess.Priority := tpLower; { Priorität niedriger als normal }
SecondProcess.Resume; { Jetzt den Thread starten }
```

Threads starten und stoppen

Bevor die Ausführung eines Threads endgültig abgeschlossen ist, kann der Thread beliebig oft gestoppt und wieder gestartet werden. Zur vorübergehenden Unterbrechung wird die Methode *Suspend* aufgerufen. Zur Fortsetzung dient die Methode *Resume*. Nach dem Aufruf von *Suspend* wird ein interner Zähler erhöht, so daß Aufrufe von *Suspend* und *Resume* verschachtelt werden können. Der Thread wird erst dann weiter ausgeführt, wenn alle Unterbrechungsanforderungen durch einen korrespondierenden Aufruf von *Resume* neutralisiert wurden.

Die vorzeitige Beendigung eines Threads kann mit der Methode *Terminate* ausgelöst werden. Dadurch wird die Eigenschaft *Terminated* des Threads auf *True* gesetzt. Die Implementierung der Methode *Execute* sollte die Eigenschaft *Terminated* regelmäßig abfragen, damit die Ausführung angehalten wird, sobald die Eigenschaft den Wert *True* annimmt.

Threads in verteilten Anwendungen

Bei der Enwicklung von verteilten Multithread-Anwendungen kommt der Thread-Koordination besondere Bedeutung zu. Dabei ist vor allem zu beachten, wie die einzelnen Threads der Anwendung von anderen Prozessen beeinflußt werden.

In verteilten Multithread-Anwendungen ist es normalerweise Aufgabe der Serveranwendung, die Mechanismen für die Thread-Verwaltung bereitzustellen. Bei der Entwicklung eines Servers ist deshalb zu berücksichtigen, wie die Anfragen der Clients bearbeitet werden.

Wenn für jede Client-Anfrage ein eigener Thread vorhanden ist, müssen Kollisionen zwischen den einzelnen Client-Threads ausgeschlossen werden. Neben den üblichen Aspekten hinsichtlich der Koordination mehrerer Threads muß eventuell auch sichergestellt werden, daß jeder Client eine eigene, konsistente Ansicht der Anwendung zur Verfügung hat. Wenn z.B. ein Client die Anwendung mehrfach aufruft und dabei jedesmal einen anderen Thread benutzt, können zum Speichern von Daten, die während mehrerer Client-Anfragen erhalten bleiben müssen, keine Thread-Variablen verwendet werden. Wenn Clients die Werte von Objekteigenschaften oder globalen Variablen ändern, wirkt sich dies nicht nur auf ihre eigene Ansicht des Objekts bzw. der Variablen aus, sondern auf die Ansichten aller anderen Clients.

Threads und botschaftsbasierte Server

Botschaftsbasierte Server empfangen die Anfrage eines Client, führen eine entsprechende Aktion aus und senden eine Botschaft an den Client zurück. Beispiele für botschaftsbasierte Server sind Internet-Server-Anwendungen und einfache Dienste, die mit Hilfe von Sockets entwickelt wurden.

Bei der Entwicklung botschaftsbasierter Server wird normalerweise für jede Client-Botschaft ein eigener Thread definiert. Sobald die Anwendung eine Client-Botschaft empfängt, spaltet sie einen Thread ab, der die Botschaft verarbeitet. Die Ausführung dieses Threads wird erst beendet, nachdem eine Antwort an den Client gesendet wurde. Während der Einsatz von globalen Objekten und Variablen sehr sorgfältig geplant werden muß, ist die Steuerung der Thread-Erzeugung und Ausführung relativ einfach, da alle Client-Botschaften im Haupt-Thread der Anwendung empfangen und von dort weitergeleitet werden.

Threads und verteilte Objekte

Bei der Entwicklung von Servern für verteilte Objekte ist die Thread-Verwaltung komplizierter. Im Gegensatz zu botschaftsbasierten Servern, bei denen Botschaften an einer zentralen Stelle empfangen und von dort weitergeleitet werden, richten Clients Aufrufe an Server-Objekte, indem sie eine ihrer Methoden aufrufen oder auf eine ihrer Eigenschaften zugreifen. Es ist für Server-Anwendungen deshalb ungleich schwerer, für jede einzelne Client-Anfrage einen separaten Thread abzuspalten.

Anwendungen (EXE-Dateien) schreiben

In einer EXE-Datei, die ein oder mehrere Objekte für Remote-Clients implementiert, gehen Client-Anfragen in Form einzelner Threads ein. Wie dies genau funktioniert, hängt davon ab, ob die Clients über COM oder über CORBA auf das Objekt zugreifen.

- **Unter COM** gehen Client-Anfragen als Teil der Botschaftsschleife der Anwendung ein. Aus diesem Grund muß in Programmteilen, die nach dem Start der Haupt-Botschaftsschleife der Anwendung ausgeführt werden, ein Schutzmechanismus existieren, der den Zugriff anderer Threads auf Objekte und den globalen Speicher verhindert. Wenn Delphi in einer Umgebung ausgeführt wird, die DCOM unterstützt, können Client-Anfragen erst dann eingehen, wenn der gesamte Quelltext im Initialisierungsabschnitt Ihrer Units ausgeführt ist. Bei einer Ausführung in anderen Umgebungen sind Sie selbst dafür verantwortlich, daß der Quelltext im Initialisierungsabschnitt Ihrer Units thread-sicher ist.
- **Unter CORBA** können Sie ein Thread-Modell im Assistenten wählen, das einen neuen CORBA-Server startet. Sie haben die Wahl zwischen Einzel- und Multithread-Modellen. In beiden Fällen besitzt jede Client-Verbindung einen eigenen Thread. Informationen, die über mehrere Client-Aufrufe hinweg erhalten bleiben müssen, können in Thread-Variablen gespeichert werden, da alle Aufrufe für einen bestimmten Client denselben Thread verwenden. Bei Modellen mit einem einzelnen Thread hat immer nur ein Client-Thread Zugriff auf eine Objektinstanz. In

diesem Fall muß zwar der globale Speicher vor dem Zugriff anderer Threads geschützt werden, es ist jedoch sichergestellt, daß beim Zugriff auf die Daten der Objektinstanz (z.B. auf Eigenschaftswerte) keine Konflikte auftreten. Bei Verwendung eines Multithread-Modells können mehrere Clients gleichzeitig auf die Anwendung zugreifen. Wenn Objektinstanzen von mehreren Clients gemeinsam genutzt werden, müssen Sie sowohl für den Schutz der Instanzdaten als auch für den der globalen Daten sorgen.

Bibliotheken schreiben

Wenn das verteilte Objekt über eine ActiveX-Bibliothek implementiert wird, ist normalerweise die Technologie für die Thread-Verwaltung zuständig, die verteilte Objektaufrufe unterstützt (COM, DCOM oder MTS). Wenn Sie eine neue Server-Bibliothek mit dem entsprechenden Assistenten anlegen, werden Sie aufgefordert, ein Thread-Modell auszuwählen. Dieses Modell bestimmt, wie Threads für die Client-Anfragen zugewiesen werden. Sie haben die Wahl zwischen folgenden Modellen:

- **Einzel-Thread-Modell:** Client-Anfragen werden durch den Aufrufmechanismus einzeln verarbeitet. Die DLL braucht keinen Quelltext für die Thread-Verwaltung zu enthalten, da sie immer nur eine Client-Anfrage empfängt.
- **Einzel-Thread-Apartment-Modell**: Auf jedes durch einen Client instantiierte Objekt wird über einen einzelnen Thread zugegriffen. Sie müssen zwar den globalen Speicher vor dem Zugriff anderer Threads schützen, die Daten der Instanz (z.B. die Objekteigenschaften) sind jedoch thread-sicher. Da ein Client immer über denselben Thread auf die Objektinstanz zugreift, können Thread-Variablen eingesetzt werden. Dieses Modell wird als Apartment-Modell bezeichnet.
- **Aktivitäts-Modell:** Zwar wird auf jede Objektinstanz jeweils nur über einen einzelnen Thread zugegriffen, die Clients verwenden aber verschiedene Threads für die einzelnen Aufrufe. Die Instanzdaten sind sicher, aber der globale Speicher muß geschützt werden, und Thread-Variablen sind über mehrere Client-Aufrufe hinweg nicht konsistent. Dieses Modell wird (unter MTS) auch als Apartment-Modell oder als Freies Modell bezeichnet.
- **Multi-Thread-Apartment-Modell:** Auf jede Objektinstanz kann gleichzeitig über verschiedene Threads zugegriffen werden. Sowohl die Instanzdaten als auch der globale Speicher müssen geschützt werden. Thread-Variablen sind über mehrere Client-Aufrufe hinweg nicht konsistent. Dieses Modell wird auch als Freies Modell bezeichnet.
- **Einzel-/Multi-Thread-Apartment-Modell:** Dieses Modell ist mit dem Multi-Thread-Apartment-Modell identisch, stellt aber zusätzlich sicher, daß alle Callbacks von Clients über denselben Thread ausgeführt werden. Werte, die als Parameter an Callback-Funktionen übergeben werden, müssen deshalb nicht geschützt werden.

Hinweis In der Regel weist ein Experte Ihrem Objekt ein Thread-Modell zu. Wenn Sie mehrere COM-Objekte in eine EXE-Datei einbinden, wird COM von der Anwendung auf dem höchsten vorgesehenen Unterstützungsniveau initialisiert (wobei *Einzel-Thread* die niedrigste und *Beide* die höchste Stufe darstellt). Sie können jedoch diese vorgegebene Art der COM-Initialisierung manuell überschreiben, indem Sie in der

primären Quelldatei der betreffenden Anwendung vor dem Aufruf von *Application.Initialize* die globale Variable *CoInitFlags* ändern.

COM-basierte Systeme verwenden zur Synchronisation der Threads die Botschaftsschleife der Anwendung. Die einzige Ausnahme bildet das Multi-Thread-Apartment-Modell, das nur unter DCOM verfügbar ist. Aus diesem Grund muß sichergestellt werden, daß längere Aufrufe, die über eine COM-Schnittstelle ausgeführt werden, die Methode *ProcessMessages* des Anwendungsobjekts aufrufen. Wenn dieser Aufruf fehlt, können andere Clients nicht auf die Anwendung zugreifen, das heißt, die Bibliothek unterstützt nur noch einen einzigen Thread.

Fehlersuche in Multithread-Anwendungen

Bei der Fehlersuche in Multithread-Anwendungen ist es oft schwierig, den Status sämtlicher gleichzeitig ausgeführter Threads zu verfolgen oder den Thread zu identifizieren, der gerade aktiv ist, wenn das Programm an einem Haltepunkt unterbrochen wird. Das Dialogfeld *Thread-Status* ermöglicht es, alle Threads der Anwendung zu verfolgen und zu verwalten. Sie öffnen dieses Dialogfeld mit dem Menübefehl *Ansicht / Threads*.

Wenn ein Debugger-Ereignis eintritt (Haltepunkt, Exception oder Pause), wird der Status aller Threads angezeigt. Wenn Sie mit der rechten Maustaste im Dialogfeld klicken, wird ein lokales Menü eingeblendet, mit dessen Befehlen Sie die zum Thread gehörende Stelle im Quelltext lokalisieren oder einen anderen Thread zum aktuellen machen können. Der nächste Ausführungsbefehl bezieht sich dann auf diesen Thread.

Das Dialogfeld *Thread-Status* enthält eine nach IDs sortierte Liste aller Threads innerhalb der Anwendung. Wenn Sie Thread-Objekte verwenden, ist der Wert in der Spalte *Thread-ID* mit dem Wert der Eigenschaft *ThreadID* identisch. Andernfalls kann die Thread-ID der einzelnen Threads durch einen Aufruf der Methode *BeginThread* ermittelt werden.

Weitere Informationen zum Dialogfeld Thread-Status finden Sie in der Online-Hilfe.

Kapitel

9

Packages und Komponenten

Ein Package ist eine spezielle Art von dynamischer Link-Bibliothek (DLL), die von Delphi-Anwendungen, der IDE oder von beiden verwendet wird. *Laufzeit-Packages* stellen ihre Funktionen zur Laufzeit einer Anwendung zur Verfügung. *Entwurfszeit-Packages* dagegen werden verwendet, um Komponenten in der IDE zu installieren und bestimmte Eigenschaftseditoren für benutzerdefinierte Komponenten zu erzeugen. Jedes Package kann sowohl als Entwurfszeit- als auch als Laufzeit-Package fungieren. Entwurfszeit-Packages rufen sogar häufig Laufzeit-Packages auf. Um sie von anderen DLLs zu unterscheiden, werden Package-Bibliotheken in Dateien mit der Erweiterung BPL (Borland Package Library) gespeichert.

Packages enthalten ebenso wie andere Laufzeit-Bibliotheken Quelltext, der von verschiedenen Anwendungen gemeinsam genutzt werden kann. Eine der am häufigsten verwendeten Delphi-Komponenten befindet sich beispielsweise in einem Package namens *VCL50*. Beim Erzeugen einer Anwendung kann festgelegt werden, daß *VCL50* verwendet werden soll. Wird eine solche Anwendung compiliert, enthält die ausführbare Version nur den anwendungsspezifischen Quelltext und die zugehörigen Daten. Der gemeinsame Quelltext ist in der Datei VCL50.BPL gespeichert. Ein Rechner, auf dem mehrere durch Packages gesteuerte Anwendungen installiert sind, benötigt lediglich eine einzige Kopie der Datei VCL50.BPL. Auf diese können dann alle Anwendungen und die IDE gemeinsam zugreifen.

Delphi umfaßt mehrere vorcompilierte Laufzeit-Packages, einschließlich *VCL50*, in denen VCL-Komponenten gekapselt sind. Entwurfszeit-Packages dienen außerdem dazu, Komponenten in der IDE zu bearbeiten.

Eine Anwendung kann mit oder ohne Packages erstellt werden. Sollen jedoch der IDE benutzerdefinierte Komponenten hinzugefügt werden, müssen Sie diese vor der Installation in Entwurfszeit-Packages compilieren.

Sie haben auch die Möglichkeit, Ihre eigenen Laufzeit-Packages für die Verwendung in mehreren Anwendungen zu erzeugen. Wenn Sie Delphi-Komponenten schreiben, können Sie diese Komponenten vor der Installation in Entwurfszeit-Packages compilieren.

Packages sinnvoll einsetzen

Entwurfszeit-Packages vereinfachen die Verteilung und Installation von benutzerdefinierten Komponenten. Laufzeit-Packages sind zwar optional, bieten aber im Vergleich zur konventionellen Programmierung mehrere Vorteile. Durch das Compilieren häufig verwendeter Quelltextelemente in eine Laufzeitbibliothek können diese von mehreren Anwendungen verwendet werden. So haben beispielsweise alle Anwendungen und Delphi selbst über Packages Zugriff auf Standardkomponenten. Da nicht jede Anwendung über eine eigene Kopie der Komponentenbibliothek verfügt, sind die ausführbaren Dateien kleiner, was Systemressourcen und Platz auf den Datenträgern spart. Außerdem beschleunigen die Packages die Compilierung, da immer nur der Quelltext compiliert wird, der spezifisch für die Anwendung ist.

Packages und Standard-DLLs

Sie erzeugen ein Package, um eine benutzerdefinierte Komponente für die Delphi-IDE zur Verfügung zu stellen. Wollen Sie hingegen eine Bibliothek erstellen, die von allen Windows-Anwendungen aufgerufen werden kann, erzeugen Sie eine Standard-DLL. Dabei ist es unerheblich, mit welchem Entwicklungswerkzeug diese Anwendung erstellt wurde.

Die folgende Tabelle enthält die Dateitypen, die im Zusammenhang mit Packages relevant sind.

Tabelle 9.1 Compilierte Package-Dateien

Namenserweiterung	Inhalt
DPK	Die Quelldatei mit der Liste der Units, die im Package enthalten sind.
DCP	Ein binäres Abbild, das einen Package-Header und die Verkettung aller DCU-Dateien im Package enthält. In dieser Datei sind auch alle Symbolinformationen enthalten, die der Compiler benötigt. Für jedes Package wird eine DCP-Datei erzeugt. Der Basisname der DCP-Datei wird vom Basisnamen der DPK-Quelldatei abgeleitet. Die Entwicklung einer Anwendung mit Packages ist nur möglich, wenn eine DCP-Datei existiert.
DCU	Ein binäres Abbild für eine Unit-Datei, die in einem Package enthalten ist. Für jede Unit-Datei wird bei Bedarf eine DCU-Datei angelegt.
BPL	Das Laufzeit-Package. Diese Datei ist eine Windows-DLL mit Delphi-spezifischen Merkmalen. Der Basisname für die BPL-Datei wird aus dem Basisnamen der DPK-Quelldatei abgeleitet.

Hinweis Packages teilen ihre globalen Daten mit den anderen Modulen einer Anwendung.

Detaillierte Informationen über DLLs und Packages finden Sie in der Object Pascal Sprachreferenz.

Laufzeit-Packages

Laufzeit-Packages werden zusammen mit Delphi-Anwendungen weitergegeben und enthalten die nötige Funktionalität für die Ausführung der Anwendung.

Eine Anwendung, die Packages benutzt, kann nur dann ausgeführt werden, wenn auf dem Rechner sowohl die EXE-Dateien als auch die korrekten Versionen der BPL-Dateien (Packages) installiert sind.

Laufzeit-Packages in Anwendungen

Um Laufzeit-Packages in einer Anwendung zu benutzen, gehen Sie folgendermaßen vor:

1. Laden oder erstellen Sie ein Projekt in der IDE.
2. Wählen Sie den Menübefehl *Projekt / Optionen*.
3. Aktivieren Sie die Registerkarte *Packages*.
4. Aktivieren Sie das Kontrollfeld *Mit Laufzeit-Packages compilieren*, und geben Sie einen oder mehrere Package-Namen in das Feld darunter ein. In diesem Feld sind sämtliche Laufzeit-Packages aufgeführt, die bereits mit installierten Entwurfszeit-Packages verbunden sind. Um der Liste ein Package hinzuzufügen, klicken Sie auf *Hinzufügen* und geben den neuen Package-Namen im Dialogfeld *Laufzeit-Package hinzufügen* ein. Wenn Sie aus der Liste der verfügbaren Packages auswählen wollen, klicken Sie in der Registerkarte *Packages* auf *Hinzufügen* und anschließend im Dialogfeld *Laufzeit-Package hinzufügen* neben dem Feld *Package-Name* auf *Durchsuchen*.

   Wenn Sie im Dialogfeld *Laufzeit-Package hinzufügen* das Eingabefeld für den Suchpfad bearbeiten, ändern Sie den global für Delphi gültigen Bibliothekssuchpfad.

   Die Namen der Packages benötigen keine Dateinamenserweiterungen. Bei der Eingabe mehrerer Namen in das Feld *Laufzeit-Packages* trennen Sie die Namen durch Semikolons:

   ```
   VCL50;VCLDB50;VCLDBX50
   ```

Die aufgelisteten Laufzeit-Packages werden beim Compilieren automatisch zu Ihrer Anwendung gelinkt, wobei doppelt vorhandene Package-Namen ignoriert werden. Bei einem leeren Eingabefeld werden die Anwendungen ohne Packages compiliert.

Laufzeit-Packages werden jeweils nur für das laufende Projekt ausgewählt. Sie haben die Möglichkeit, die aktuelle Auswahl als Standardeinstellung für neue Projekte zu speichern, indem Sie das Kontrollfeld *Vorgabe* aktivieren.

Hinweis Auch in Anwendungen, die Packages verwenden, müssen die Namen der ursprünglichen Delphi-Units in der **uses**-Klausel der Quelldateien angegeben werden. Eine Quelldatei für das Hauptformular einer Anwendung könnte also wie folgt beginnen:

```
unit MainForm;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

Alle in diesem Beispiel aufgeführten Units sind im Package *VCL50* enthalten. Sie müssen diese Referenzen aber auch dann in die **uses**-Klausel aufnehmen, wenn Sie *VCL50* in Ihrer Anwendung verwenden. Andernfalls treten Compiler-Fehler auf. In generierten Quelldateien werden diese Units automatisch in die **uses**-Klausel eingefügt.

Packages dynamisch laden

Um ein Package zur Laufzeit zu laden, rufen Sie die Funktion *LoadPackage* auf. So könnte beispielsweise der folgende Code ausgeführt werden, wenn in einem Dialogfeld zur Auswahl von Dateien eine Datei ausgewählt wird.

```
with OpenDialog1 do
    if Execute then
      with PackageList.Items do
        AddObject(FileName, Pointer(LoadPackage(Filename)));
```

Um ein Package dynamisch aus dem Speicher zu entfernen, steht die Prozedur *UnloadPackage* zur Verfügung. Seien Sie allerdings vorsichtig bei der Freigabe der Instanzen von Klassen, die in dem Package definiert sind, sowie bei der Deregistrierung von Klassen, die durch das Package registriert wurden.

Benötigte Laufzeit-Packages auswählen

Im Lieferumfang von Delphi sind neben *VCL50* weitere vorcompilierte Laufzeit-Packages enthalten, die wichtige Komponenten und Elemente zur Verfügung stellen.

Das Package *VCL50* enthält die am häufigsten verwendeten Komponenten, Systemfunktionen und Elemente der Windows-Benutzeroberfläche, jedoch keine datenbankspezifischen und keine Windows-3.1-Komponenten. Diese werden in gesonderten Packages zur Verfügung gestellt. Die folgende Tabelle enthält die mitgelieferten Laufzeit-Packages und die darin enthaltenen Units.

Tabelle 9.2 Laufzeit-Packages von Delphi

Package	Units
VCL50.BPL	Ax, Buttons, Classes, Clipbrd, Comctrls, Commctrl, Commdlg, Comobj, Comstrs, Consts, Controls, Ddeml, Dialogs, Dlgs, Dsgnintf, Dsgnwnds, Editintf, Exptintf, Extctrls, Extdlgs, Fileintf, Forms, Graphics, Grids, Imm, IniFiles, Isapi, Isapi2, Istreams, Libhelp, Libintf, Lzexpand, Mapi, Mask, Math, Menu, Messages, Mmsystem, Nsapi, Ole2I, Oleconst, Olectnrs, Olectrls, Oledlg, Penwin, Printers, Proxies, Registry, Regstr, Richedit, Shellapi, Shlobj, Stdctrls, Stdvcl, Sysutils, Tlhelp32, Toolintf, Toolwin, Typinfo, Vclcom, Virtintf, Windows, Wininet, Winsock, Winspool, Winsvc
VCLX50.BPL	Checklst, Colorgrd, Ddeman, Filectrl, Mplayer, Outline, Tabnotbk, Tabs

Tabelle 9.2 Laufzeit-Packages von Delphi (Fortsetzung)

Package	Units
VCLDB50.BPL	Bde, Bdeconst, Bdeprov, Db, Dbcgrids, Dbclient, Dbcommon, Dbconsts, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables, Dsintf, Provider, SMintf
VCLDBX50.BPL	Dblookup, Report
DSS50.BPL	Mxarrays, Mxbutton, Mxcommon, Mxconsts, Mxdb, Mxdcube, Mxdssqry, Mxgraph, Mxgrid, Mxpivsrc, Mxqedcom, Mxqparse, Mxqryedt, Mxstore, Mxtables, Mxqvb
QRPT50.BPL	Qr2const, Qrabout, Qralias, Qrctrls, Qrdatasu, Qrexpbld, Qrextra, Qrprev, Qrprgres, Qrprntr, Qrqred32, Quickrpt
TEE50.BPL	Arrowcha, Bubblech, Chart, Ganttch, Series, Teeconst, Teefunci, Teengine, Teeprocs, Teeshape
TEEDB50.BPL	Dbchart, Qrtee
TEEUI50.BPL	Areaedit, Arrowedi, Axisincr, Axmaxmin, Baredit, Brushdlg, Bubbledi, Custedit, Dbeditch, Editchar, Flineedi, Ganttedi, Ieditcha, Pendlg, Pieedit, Shapeedi, Teeabout, Teegally, Teelisb, Teeprevi, Teexport
VCLSMP50.BPL	Sampreg, Smpconst

Um eine Client/Server-Datenbankanwendung zu erstellen, die Packages verwendet, werden lediglich die beiden Laufzeit-Packages *VCL50* und *VCLDB50* benötigt. Falls diese Anwendung zusätzlich Gliederungsdiagramme (*TOutline*) enthalten soll, benötigen Sie zusätzlich das Package *VCLX50*. Sie laden diese Packages folgendermaßen: Wählen Sie *Projekt / Packages*, öffnen Sie die Registerkarte *Packages*, und geben Sie anschließend in das Feld *Laufzeit-Packages* die folgende Liste ein:

```
VCL50;VCLDB50;VCLX50
```

Die Angabe von *VCL50* ist hier unnötig, da dieses Package bereits in der **requires**-Klausel von *VCLDB50* referenziert wird (siehe »Die requires-Klausel« auf Seite 9-10). Die Anwendung wird unabhängig davon compiliert, ob Sie *VCL50* angeben oder nicht.

Benutzerdefinierte Packages

Ein benutzerdefiniertes Package kann entweder von Ihnen selbst durch Codierung und Compilierung als BPL-Datei erzeugt oder aber als vorcompiliertes Package von anderen Entwicklern erworben werden. Sie verwenden ein solches Laufzeit-Package in einer Anwendung, indem Sie *Projekt / Optionen* wählen und in der Registerkarte *Packages* den Namen des Package in das entsprechende Feld eingeben. Um beispielsweise ein Statistik-Package namens STATS.BPL verfügbar zu machen, geben Sie folgendes in das Textfeld ein:

```
VCL50;VCLDB50;STATS
```

Auf die gleiche Weise können Sie ein selbsterzeugtes Package in die Liste der benötigten Laufzeit-Packages einfügen.

Entwurfszeit-Packages

Entwurfszeit-Packages werden verwendet, um Komponenten in der Komponentenpalette der IDE zu installieren und spezielle Eigenschaftseditoren für benutzerdefinierte Komponenten zu erzeugen.

Delphi umfaßt die folgenden vorinstallierten Packages mit Komponenten:

Table 9.3 Entwurfszeit-Packages von Delphi

Package	Registerkarte der Komponentenpalette
DCLSTD50.BPL	Standard, Zusätzlich, System, Win32, Dialoge
DCLTEE50.BPL	Zusätzlich (Komponente *TChart*)
DCLDB50.BPL	Datenzugriff, Datensteuerung
DCLMID50.BPL	Datenzugriff (MIDAS)
DCL31W50.BPL	Win 3.1
DCLNET50.BPL NMFAST50.BPL	Internet
DCLSMP50.BPL	Beispiele
DCLOCX50.BPL	ActiveX
DCLQRT50.BPL	QReport
DCLDSS50.BPL	Datenanalyse
IBSMP50.BPL	Beispiele (Komponente *IBEventAlerter*)
DCLINT50.BPL	(International-Tools—Ressourcen-DLL-Experte)
RCEXPERT.BPL	(Ressourcen-Experte)
DBWEBXPRT.BPL	(Web-Experte)

Diese Entwurfszeit-Packages rufen die in ihrer **requires**-Klausel aufgeführten Laufzeit-Packages auf (siehe auch »Die requires-Klausel« auf Seite 9-10). So referenziert beispielsweise *DCLSTD50* das Package *VCL50*. *DCLSTD50* selbst enthält zusätzliche Funktionalität, mit der die meisten Standardkomponenten in der Komponentenpalette verfügbar gemacht werden.

Zusätzlich zu den vorinstallierten Packages können Sie eigene oder von Fremdherstellern entwickelte Komponenten-Packages in der IDE installieren. Das Entwurfszeit-Package *DCLUSR50* stellt einen Standard-Container für neue Komponenten dar.

Komponenten-Packages installieren

In Delphi sind grundsätzlich alle Komponenten als Packages in der IDE installiert. Wenn Sie eigene Komponenten entwickeln, müssen Sie dafür ein neues Package erstellen und compilieren (siehe auch »Packages erstellen und bearbeiten« auf Seite 9-8). Der Quelltext Ihrer Komponenten muß sich an dem Muster orientieren, das in Teil IV, »Benutzerdefinierte Komponenten erzeugen« beschrieben ist.

Um eigene oder fremde Komponenten zu installieren oder wieder zu entfernen, gehen Sie folgendermaßen vor:

1. Zur Installation eines neues Package kopieren oder verschieben Sie dessen Dateien zuerst in ein lokales Verzeichnis. Wenn zum Package BPL-, DCP- und DCU-Dateien gehören, müssen unbedingt alle diese Dateien kopiert werden. Detaillierte Informationen über diese Dateien finden Sie im Abschnitt »Package-Dateien nach erfolgreicher Compilierung« auf Seite 9-14.

   Das Verzeichnis, in dem die beteiligten DCP- und DCU-Dateien gespeichert sind, muß sich im Bibliothekssuchpfad von Delphi befinden.

   Wenn das Package als DPC-Datei (Package-Sammlung) vorliegt, braucht nur diese Datei kopiert zu werden, da alle anderen Dateien in ihr enthalten sind. (Informationen über Package-Sammlungen finden Sie unter »Package-Sammlungen« auf Seite 9-15.)
2. Wählen Sie in der IDE den Menübefehl *Komponente / Packages installieren*, oder wählen Sie *Projekt / Optionen*, und öffnen Sie dann die Registerkarte *Packages*.
3. Unter *Entwurfszeit-Package* wird eine Liste der verfügbaren Packages angezeigt.
   - Sie installieren ein Package, indem sie in der Liste das Kontrollfeld neben dem Package aktivieren.
   - Sie deinstallieren ein Package, indem sie in der Liste das Kontrollfeld neben dem Package deaktivieren.
   - Um sich die Komponenten eines installierten Package anzeigen zu lassen, markieren Sie das Package und klicken auf die Schaltfläche *Komponenten*.
   - Um ein neues Package in die Liste aufzunehmen, klicken Sie auf die Schaltfläche *Hinzufügen* und suchen im zugehörigen Dialogfeld nach dem Verzeichnis, das die BPL- oder DPC-Dateien enthält (siehe Schritt 1). Markieren Sie die BPL- oder DPC-Datei, und klicken Sie auf *öffnen*. Wenn Sie eine DPC-Datei markieren, wird ein weiteres Dialogfeld angezeigt, in dem Sie Festlegungen für die Dekomprimierung von BPL-Dateien und anderen, in der Package-Sammlung enthaltenen Dateien treffen können.
   - Sie entfernen ein Package, indem Sie es markieren und auf die Schaltfläche *Entfernen* klicken.
4. Klicken Sie auf *OK*.

Die im Package enthaltenen Komponenten werden in der Komponentenpalette in den Registerkarten installiert, die in der Prozedur *RegisterComponents* der einzelnen Komponenten festgelegt wurden. Dabei werden auch die mit dieser Prozedur zugewiesenen Namen verwendet.

Bei der Erstellung neuer Projekte werden alle verfügbaren Packages installiert (vorausgesetzt, die Standardeinstellungen wurden nicht geändert). Wenn die aktuellen Installationseinstellungen als Standardeinstellungen für neue Projekte verwendet werden sollen, aktivieren Sie das Kontrollfeld *Vorgabe* am unteren Rand des Dialogfeldes.

Sie können einzelne Komponenten ohne Deinstallation eines gesamten Package aus der Komponentenpalette entfernen, indem Sie im Menü einen der Befehle *Komponente / Palette konfigurieren* oder *Tools / Umgebungsoptionen* wählen und die Registerkarte *Palette* aktivieren. In dieser Registerkarte wird jede installierte Komponente zusam-

men mit dem Namen der zugehörigen Registerkarte aufgelistet. Sie entfernen eine beliebige Komponente aus der Palette, indem Sie die Komponente markieren und auf die Schaltfläche *Verbergen* klicken.

Packages erstellen und bearbeiten

Bei der Erstellung eines Package müssen Sie folgende Elemente festlegen:

- Einen Namen für das Package.
- Eine Liste weiterer Packages, die zu dem neuen Package gelinkt oder von dem neuen Package aufgerufen werden sollen.
- Eine Liste von Unit-Dateien, die bei der Compilierung in das Package aufgenommen oder eingebunden werden sollen. Ein Package ist bildlich gesehen eine Verpackung für diese Quelltext-Units, welche die Funktionalität der compilierten BPL-Datei enthalten. In der **contains**-Klausel hinterlegen Sie die Quelltext-Units für benutzerdefinierte Komponenten, die in das Package compiliert werden sollen.

Die Quelldateien für das Package mit der Namenserweiterung DPK werden vom Package-Editor generiert.

Ein Package erstellen

Mit den nachfolgend beschriebenen Schritten erstellen Sie ein Package. Weitere Informationen zu diesem Thema finden Sie unter »Die Struktur eines Package« auf Seite 9-10.

1. Wählen Sie *Datei* / *Neu*, dann markieren Sie das Symbol *Package* und bestätigen mit *OK*.
2. Das generierte Package wird im Package-Editor angezeigt.
3. Im Package-Editor werden für das neue Package zwei Symbole mit den Namen *Contains* und *Requires* angezeigt.
4. Um eine neue Unit in die **contains**-Klausel aufzunehmen, klicken Sie auf die Schaltfläche *Zu Package hinzufügen*. Geben Sie dann in der Registerkarte *Unit hinzufügen* in das Eingabefeld *Dateiname der Unit* einen Dateinamen mit der Erweiterung PAS ein, oder lokalisieren Sie die gewünschte Datei mit der Schaltfläche *Durchsuchen*. Klicken Sie anschließend auf *OK*. Die gewählte Unit wird nun im Package-Editor unter dem Symbol *Contains* angezeigt. Wenn Sie weitere Units hinzufügen wollen, wiederholen Sie den Vorgang.
5. Um ein Package in die **requires**-Klausel aufzunehmen, klicken Sie auf die Schaltfläche *Zu Package hinzufügen*. Geben Sie dann in der Registerkarte *Benötigt* in das Eingabefeld *Package-Name* einen Dateinamen mit der Erweiterung DCP ein, oder lokalisieren Sie die gewünschte Datei mit der Schaltfläche *Durchsuchen*. Klicken Sie anschließend auf *OK*. Das gewählte Package wird nun im Package-Editor unter

dem Symbol *Requires* angezeigt. Wenn Sie weitere Packages hinzufügen wollen, wiederholen Sie den Vorgang.

6 Im selben Dialogfeld legen Sie fest, welche Art von Package erzeugt werden soll.

- Um ein Package zu erstellen, das ausschließlich zur Entwurfszeit verfügbar ist, markieren Sie das Kontrollfeld *Entwurf*. Sie können zu diesem Zweck auch die Compiler-Direktive {$DESIGNONLY} zur DPK-Datei hinzufügen.
- Für ein reines Laufzeit-Package markieren Sie das Kontrollfeld *Laufzeit* oder fügen die Compiler-Direktive {$RUNONLY} zur DPK-Datei hinzu.
- Soll ein Package sowohl während des Entwurfs als auch zur Laufzeit verfügbar sein, aktivieren Sie keines der beiden Kontrollfelder.

7 Compilieren Sie das neue Package, indem Sie im Package-Editor auf die Schaltfläche *Package compilieren* klicken.

Ein bestehendes Package bearbeiten

Sie haben mehrere Möglichkeiten, ein bestehendes Package zur Bearbeitung zu öffnen.

- Wählen Sie einen der Menübefehle *Datei / Öffnen* oder *Datei / Neu öffnen*, und markieren Sie eine DPK-Datei.
- Wählen Sie *Komponente / Packages installieren*. Markieren Sie in der Liste *Entwurfszeit-Package* ein Package, und klicken Sie auf *Bearbeiten*.
- Wenn der Package-Editor geöffnet ist, markieren Sie eines der Packages unter *Requires*, klicken mit der rechten Maustaste und wählen *Öffnen*.

Um die Beschreibung eines Package zu bearbeiten oder um Anwendungsoptionen festzulegen, klicken Sie im Package-Editor auf den Schalter *Optionen* und wählen dann die Registerkarte *Beschreibung*.

In der linken unteren Ecke des Dialogfelds *Projektoptionen* befindet sich das Kontrollfeld *Vorgabe*. Wenn dieses Feld markiert ist und Sie auf *OK* klicken, werden die von Ihnen gewählten Optionen als Standardeinstellungen für neue Projekte gespeichert. Um die ursprünglichen Einstellungen wiederherzustellen, müssen Sie die Datei DEFPROJ.DOF löschen oder umbenennen.

Quelldateien von Packages manuell bearbeiten

Ähnlich wie Projektdateien generiert Delphi auch die Quelldateien von Packages aus Informationen, die Sie bereitstellen. Package-Quelldateien können manuell bearbeitet werden. Um Verwechslungen mit anderen Dateien zu vermeiden, die ebenfalls Object-Pascal-Quelltext enthalten, müssen Package-Quelldateien mit der Namenserweiterung DPK (Delphi-Package) gespeichert werden.

Mit den folgenden Schritten zeigen Sie die Quelldatei eines Package im Quelltexteditor an:

1 Öffnen Sie das Package im Package-Editor.

2 Klicken Sie im Package-Editor mit der rechten Maustaste, und wählen Sie *Package-Quelle anzeigen*.

- Nach dem Schlüsselwort **package** wird der Name des Package angezeigt.
- In der **requires**-Klausel sind weitere externe Packages aufgeführt, die vom aktuellen Package verwendet werden. Diese Klausel wird nur benötigt, wenn das Package Units enthält, die auf Units in anderen Packages zugreifen.
- In der **contains**-Klausel sind die Unit-Dateien aufgeführt, die compiliert und in das Package eingebunden werden. Eingebunden werden auch Units, die von anderen, enthaltenen Units verwendet werden, die nicht in benötigten Packages enthalten sind. Diese Units sind aber nicht in der **contains**-Klausel aufgeführt (der Compiler gibt eine entsprechende Warnung aus).

Im folgenden Beispiel wird das Package VCLDB50 deklariert:

```
package VCLDB50;
   requires VCL50;
   contains Db, Dbcgrids, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables,
mycomponent in 'C:\components\mycomponent.pas';
end.
```

Die Struktur eines Package

Packages benennen

Die Namen von Packages müssen innerhalb eines Projekts eindeutig sein. Wenn Sie einem Package z.B. den Namen *STATS* geben, erzeugt der Package-Editor eine Quelldatei namens STATS.DPK. Der Compiler generiert eine ausführbare Datei namens STATS.BPL und ein binäres Abbild mit dem Namen STATS.DCP. In den **requires**-Klauseln anderer Packages kann das Package über den Namen *STATS* referenziert werden. Mit diesem Namen wird auch in einer Anwendung auf das Package Bezug genommen.

Die requires-Klausel

Die **requires**-Klausel gibt an, welche externen Packages vom aktuellen Package verwendet werden. Ein in die **requires**-Klausel eingebundenes externes Package wird beim Compilieren automatisch zu jeder Anwendung gelinkt, die sowohl das aktuelle Package als auch eine der Units verwendet, die im externen Package enthalten sind.

Wenn Unit-Dateien, die in Ihrem Package enthalten sind, andere gepackte Units referenzieren, sollten diese Packages in der **requires**-Klausel Ihres Package enthalten sein. Fehlen die Packages in der **requires**-Klausel, lädt der Compiler die entsprechenden Units auf dem normalen Weg, also ohne BPL-Dateien. Ein Package, das keine anderen Packages referenziert, benötigt keine **requires**-Klausel.

Hinweis Die meisten Packages benötigen *VCL50*. Deshalb muß in der **requires**-Klausel eines jeden Package, das von VCL-Units abhängt (einschließlich *SysUtils*), entweder *VCL50* selbst oder ein Package aufgelistet sein, das *VCL50* benötigt.

Zirkuläre Package-Bezüge vermeiden

In der **requires**-Klausel eines Package darf es keine zirkulären Bezüge geben. Das bedeutet im einzelnen:

- Ein Package darf nicht in seiner eigenen **requires**-Klausel aufgeführt sein.
- Packages, die sich in einer Folge referenzieren, dürfen innerhalb dieser Folge keinen erneuten Bezug aufeinander nehmen. Wenn etwa Package *A* Package *B* referenziert, kann Package *B* nicht Package *A* referenzieren. Wenn Package *A* Package *B* und Package *B* Package *C* referenziert, kann Package *C* nicht Package *A* referenzieren.

Doppelte Package-Bezüge vermeiden

Der Compiler ignoriert doppelte Bezüge in der **requires**-Klausel oder im Eingabefeld *Laufzeit-Packages*. Aus Gründen der Übersichtlichkeit sollten Sie aber doppelte Referenzen entfernen.

Die contains-Klausel

In der **contains**-Klausel wird festgelegt, welche Unit-Dateien in das Package eingebunden werden sollen. Wenn Sie ein eigenes Package schreiben, legen Sie den Quelltext in CPP-Dateien ab und nehmen diese in die **contains**-Klausel auf.

Redundanten Quelltext vermeiden

Ein Package kann nicht Bestandteil der **contains**-Klausel eines anderen Package sein.

Alle Units, die entweder direkt in der **contains**-Klausel eines Package oder indirekt in einer seiner Units enthalten sind, werden beim Compilieren in das Package eingebunden.

Eine Unit kann weder direkt noch indirekt in mehr als einem der Packages enthalten sein, die von einer Anwendung (auch von der Delphi-IDE) benutzt werden. Es ist daher nicht möglich, ein selbsterzeugtes Package, das eine der Units in *VCL50* enthält, in der IDE zu installieren. Um eine bereits eingebundene Unit-Datei in einem weiteren Package zu verwenden, muß das erste Package in die **requires**-Klausel des zweiten aufgenommen werden.

Packages compilieren

Ein Package kann entweder in der IDE oder von der Befehlszeile aus compiliert werden. Um ein bestehendes Package in der IDE erneut zu compilieren, gehen Sie folgendermaßen vor:

1. Wählen Sie *Datei / Öffnen*.
2. Wählen Sie in der Dropdown-Liste *Dateityp* die Option *Delphi-Quell-Package (*.DPK)*.
3. Wählen Sie im Dialogfeld eine DPK-Datei aus.

4 Der Package-Editor wird geöffnet. Klicken Sie auf die Schaltfläche *Package compilieren*.

Sie können direkt in den Quelltext eines Package Compiler-Direktiven einfügen. Weitere Informationen zu diesem Thema finden Sie nachfolgend unter »Spezielle Compiler-Direktiven für Packages«.

Wenn Sie von der Befehlszeile aus compilieren, stehen einige Linker-Schalter für Packages zur Verfügung. Weitere Informationen zu diesem Thema finden Sie unter »Der Befehlszeilen-Compiler und -Linker« auf Seite 9-14.

Spezielle Compiler-Direktiven für Packages

Die folgende Tabelle enthält spezielle Compiler-Direktiven für Packages, die in den Quelltext eingefügt werden können.

Tabelle 9.4 Spezielle Compiler-Direktiven für Packages

Anweisung	Zweck
{$IMPLICITBUILD OFF}	Verhindert, daß das Package zu einem späteren Zeitpunkt implizit neu compiliert wird. Diese Direktive kann in DPK-Dateien bei der Compilierung von Packages mit Low-Level-Funktionen verwendet werden, die sich nur selten ändern oder deren Quelltext nicht weitergegeben wird.
{$G-} oder {IMPORTEDDATA OFF}	Deaktiviert die Erstellung von Referenzen auf importierte Daten. Diese Direktive reduziert zwar die Anzahl der Speicherzugriffe, verhindert aber auch, daß die Unit Variablen aus anderen Packages referenziert.
{$WEAKPACKAGEUNIT ON}	Erzeugt »schwach gepackte« Units. Siehe »Schwaches Packen« auf Seite 9-12.
{$DENYPACKAGEUNIT ON}	Verhindert, daß die Unit in ein Package aufgenommen wird.
{$DESIGNONLY ON}	Compiliert das Package für die Installation in der IDE (wird in der DPK-Datei verwendet).
{$RUNONLY ON}	Compiliert das Package nur als Laufzeit-Package (wird in der DPK-Datei verwendet).

Hinweis Wenn Sie die Compiler-Direktive **{$DENYPACKAGEUNIT ON}** im Quelltext verwenden, wird die Unit nicht in das Package aufgenommen. Die Angabe von **{$G-}** oder {**IMPORTEDDATA OFF}** kann bewirken, daß das Package nicht zusammen mit anderen Packages in derselben Anwendung verwendet werden kann. Packages, die mit der Direktive **{$DESIGNONLY ON}** compiliert wurden, enthalten zusätzlichen Quelltext, der von der IDE benötigt wird. Derartige Packages sollten in normalen Anwendungen nicht verwendet werden. Gegebenenfalls können weitere Compiler-Direktiven in den Quelltext von Packages aufgenommen werden. Ausführliche Informationen hierzu finden Sie unter »Compiler-Direktiven« in der Online-Hilfe.

Schwaches Packen

Die Direktive **$WEAKPACKAGEUNIT** beeinflußt die Art und Weise, in der eine DCU-Datei in den DCP- und BPL-Dateien eines Package gespeichert wird. (Detail-

lierte Informationen darüber finden Sie unter »Package-Dateien nach erfolgreicher Compilierung« auf Seite 9-14.) Die Direktive **{$WEAKPACKAGEUNIT ON}** in einer Unit-Datei veranlaßt den Compiler, diese Unit wenn möglich nicht in die BPL-Datei einzubinden. Er erzeugt statt dessen eine nicht an das Package gebundene lokale Kopie der Unit, wenn diese von einer anderen Anwendung oder einem anderen Package benötigt wird. Eine solche Unit bezeichnet man als *schwach gepackt*.

Nehmen wir an, Sie haben ein Package mit dem Namen *PACK* erzeugt, das als einzige Unit *UNIT1* enthält. *UNIT1* verwendet keine weiteren Units, ruft aber Funktionen in RARE.DLL auf. Wenn UNIT1.PAS die Direktive **{$WEAKPACKAGEUNIT ON}** enthält und Sie das Package compilieren, wird *UNIT1* nicht in PACK.DCP eingebunden, und Sie brauchen zusammen mit *PACK* keine Kopien von RARE.DLL weiterzugeben. *UNIT1* wird jedoch nach wie vor in PACK.DCP eingebunden. Andere Packages oder Anwendungen, die *UNIT1* referenzieren, verwenden dagegen die nicht gepackte Kopie aus PACK.DCP.

Nehmen wir nun an, Sie fügen *PACK* eine zweite Unit mit der Bezeichnung *UNIT2* hinzu, die *UNIT1* verwendet. In diesem Fall wird *UNIT1* selbst dann in PACK.DCP eingebunden, wenn UNIT1.PAS beim Compilieren von *PACK* die Direktive **{$WEAKPACKAGEUNIT ON}** enthält. Alle anderen Packages oder Anwendungen, die sich auf *UNIT1* beziehen, verwenden dagegen die nicht in ein Package eingebundene Kopie aus der Datei PACK.DCP.

Hinweis Unit-Dateien, welche die Direktive **{$WEAKPACKAGEUNIT ON}** verwenden, dürfen keine globalen Variablen und Abschnitte zur Initialisierung und Finalisierung enthalten.

Mit der Direktive **$WEAKPACKAGEUNIT** stehen Entwicklern, die ihre BPL-Dateien anderen Delphi-Programmierern zugänglich machen wollen, umfangreiche Möglichkeiten zur Verfügung. So wird beispielsweise die Weitergabe selten verwendeter DLLs überflüssig, und Konflikte zwischen Packages, die von derselben externen Bibliothek abhängig sind, lassen sich leichter vermeiden.

Die Delphi-Unit *PenWin* referenziert beispielsweise die Datei PENWIN.DLL. Die Unit wird von den wenigsten Projekten benutzt, und auf den wenigsten Computern ist PENWIN.DLL installiert. Aus diesem Grund ist die Unit *PenWin* in *VCL50* nur schwach gepackt. Bei der Compilierung eines Projekts, das die Unit *PenWin* und das Package *VCL50* referenziert, wird *PenWin* aus der Datei VCL50.DCP kopiert und direkt in das Projekt eingebunden. Die resultierende ausführbare Datei ist statisch zu PENWIN.DLL gelinkt.

Wäre *PenWin* nicht schwach gepackt, würden sich zwei grundsätzliche Probleme ergeben. Erstens würde *VCL50* selbst statisch zu PENWIN.DLL gelinkt, was zur Folge hätte, daß *VCL50* auf Rechnern ohne installierte PENWIN.DLL nicht mehr geladen werden könnte. Zweitens würde der Compiler den Versuch zurückweisen, ein eigenes Package zu erzeugen, das *PenWin* enthält, da die Unit *PenWin* sowohl in *VCL40* als auch in Ihrem eigenen Package enthalten wäre. Ohne schwaches Packen könnte *PenWin* also nicht so einfach zusammen mit *VCL50* weitergegeben werden.

Der Befehlszeilen-Compiler und -Linker

Um ein Projekt als Package zu erstellen, können Sie die in der folgenden Tabelle aufgelisteten speziellen Optionen für Packages verwenden.

Tabelle 9.5 Package-Optionen für den Befehlszeilen-Linker

Option	Zweck
-$G-	Deaktiviert die Erstellung von Referenzen auf importierte Daten. Diese Direktive reduziert zwar die Anzahl der Speicherzugriffe, verhindert aber auch, daß das Package Variablen aus anderen Packages referenziert.
-LEPfad	Legt das Verzeichnis fest, in dem die BPL-Datei des Package gespeichert wird.
-LNPfad	Legt das Verzeichnis fest, in dem die DCP-Datei des Package gespeichert wird.
-LU*Package*	Verwendet die angegebenen Packages.
-Z	Verhindert, daß das Package zu einem späteren Zeitpunkt implizit neu compiliert wird. Verwenden Sie diese Direktive bei der Compilierung von Packages mit Low-Level-Funktionen, die sich nur selten ändern oder deren Quelltext nicht weitergegeben wird.

Hinweis Die Verwendung der Direktive **-$G-** kann zur Folge haben, daß ein Package nicht mehr zusammen mit anderen Packages in derselben Anwendung verwendet werden kann. Gegebenenfalls können Sie bei der Compilierung von Packages weitere Befehlszeilenoptionen verwenden. Ausführliche Informationen hierzu finden Sie unter »Der Befehlszeilen-Compiler« in der Online-Hilfe.

Package-Dateien nach erfolgreicher Compilierung

Um ein Package zu erstellen, compilieren Sie eine Quelldatei mit der Namenserweiterung DPK. Der Basisname der DPK-Datei wird als Basisname für die vom Compiler generierten Dateien verwendet. Wenn Sie beispielsweise eine Package-Quelldatei mit den Namen TRAYPAK.DPK compilieren, erhalten Sie ein Package namens TRAYPAK.BPL.

Die folgende Tabelle enthält die Dateien, die bei der erfolgreichen Compilierung eines Package erzeugt werden:

Tabelle 9.6 Compilierte Package-Dateien

Erweiterung	Inhalt
DCP	Ein binäres Abbild, das einen Package-Header und die Verkettung aller DCU-Dateien im Package enthält. Für jedes Package wird eine DCP-Datei erzeugt. Der Basisname der DCP-Datei wird vom Basisnamen der DPK-Quelldatei abgeleitet.
DCU	Ein binäres Abbild einer Unit-Datei, die in einem Package enthalten ist. Für jede Unit-Datei wird bei Bedarf eine DCU-Datei angelegt.
BPL	Das Laufzeit-Package. Diese Datei ist eine Windows-DLL mit Delphi-spezifischen Merkmalen. Der Basisname für die BPL-Datei wird aus dem Basisnamen der DPK-Quelldatei abgeleitet.

Packages weitergeben

Anwendungen mit Packages weitergeben

Beim Weitergeben von Anwendungen, die mit Laufzeit-Packages arbeiten, müssen den Benutzern sowohl die EXE-Dateien als auch die zugehörigen Bibliotheksdateien (BPL oder DLL) zur Verfügung gestellt werden. Sollten sich die Bibliotheksdateien in einem anderen Verzeichnis als die Programmdateien befinden, müssen die Benutzerpfade entsprechend geändert werden. Per Konvention werden Bibliotheksdateien im Verzeichnis WINDOWS\SYSTEM abgelegt. Wenn Sie InstallShield Express verwenden, kann das Installationsskript die Systemumgebung überprüfen und feststellen, ob die benötigten Packages vielleicht schon vorhanden sind, anstatt sie einfach neu zu installieren.

Packages anderen Entwicklern zur Verfügung stellen

Wenn Sie anderen Entwicklern Ihre Laufzeit- oder Entwurfszeit-Packages zur Verfügung stellen, müssen sie auch die zugehörigen DCP- und BPL-Dateien weitergeben. Häufig ist es auch sinnvoll, die DCU-Dateien weiterzugeben.

Package-Sammlungen

Ein Package läßt sich am einfachsten in Form einer sogenannten Package-Sammlung (DPC-Datei) an andere Entwickler weitergeben. In einer Package-Sammlung werden eines oder mehrere Packages und alle BPL-Dateien und sonstigen Dateien zusammengefaßt, die weitergegeben werden sollen. Wenn Sie eine Package-Sammlung für die Installation in der IDE auswählen, werden die betreffenden Dateien automatisch aus ihrem PCE-Container extrahiert. Im angezeigten Dialogfeld können Sie dann festlegen, ob alle oder nur bestimmte in der Sammlung enthaltene Packages installiert werden sollen.

Zur Erstellung einer Package-Sammlung gehen Sie folgendermaßen vor:

1. Wählen Sie *Tools / Editor für Package-Sammlung*, um den Editor für Package-Sammlungen zu öffnen.
2. Klicken Sie auf die Schaltfläche *Package hinzufügen*, markieren Sie im Dialogfeld *Package auswählen* ein Package, und klicken Sie auf *Öffnen*. Wenn Sie der Sammlung weitere BPL-Dateien hinzufügen möchten, wiederholen Sie diesen Vorgang. Die hinzugefügten BPL-Dateien werden in einem Baumdiagramm auf der linken Seite des Editors angezeigt. Um ein Package aus der Sammlung zu entfernen, markieren Sie die betreffende BPL-Datei und klicken auf die Schaltfläche *Entfernen*.
3. Markieren Sie das Symbol *Auswahl* ganz oben im Baumdiagramm. Auf der rechten Seite des Editors für Package-Sammlungen sehen Sie nun zwei Felder:

Im Eingabefeld *Autoren-/Hersteller* können Sie erläuternde Informationen zur Package-Sammlung eingeben. Diese Informationen werden angezeigt, wenn die Benutzer die Packages installieren.

Unter *Verzeichnisliste* können Sie die Standardverzeichnisse angeben, in denen die Dateien der Package-Sammlung installiert werden sollen. Um die Liste der Verzeichnisse zu bearbeiten, verwenden Sie die Schaltflächen *Hinzufügen*, *Bearbeiten* und *Löschen*. Wenn Sie beispielsweise alle Quelltextdateien in dasselbe Verzeichnis kopieren wollen, könnten Sie als Verzeichnisnamen SOURCE und als Standardpfad C:\MYPACKAGE\SOURCE eingeben. Bei der Installation wird dann C:\MYPACKAGE\SOURCE als Verzeichnis für die Dateien der Package-Sammlung vorgeschlagen.

4 Eine Package-Sammlung kann neben BPL-Dateien auch Dateien mit der Namenserweiterung DCP, DCU und PAS (Unit-Datei) enthalten. Außerdem können weitere Dateien (etwa zum Zweck der Dokumentation) in die Sammlung aufgenommen und weitergegeben werden. Diese Dateien werden in einer Dateigruppe zusammengefaßt, die dem jeweiligen Package (BPL-Datei) zugeordnet ist. Die Dateien in einer solchen Gruppe können nur zusammen mit der entsprechenden BPL-Datei installiert werden. Um zusätzliche Dateien in eine Package-Sammlung aufzunehmen, markieren Sie im Baumdiagramm eine BPL-Datei, klicken auf die Schaltfläche *Dateigruppe hinzufügen* und geben einen Namen für die Dateigruppe ein. Um weitere Dateigruppen hinzuzufügen, wiederholen Sie diesen Vorgang. Wenn Sie eine Dateigruppe markieren, werden auf der rechten Seite des Editors für Package-Sammlungen neue Felder angezeigt.

   - Markieren Sie in der Verzeichnisliste das Verzeichnis, in dem die Dateien dieser Gruppe installiert werden sollen. Die Dropdown-Liste enthält die Verzeichnisse, die Sie in Schritt 3 unter *Verzeichnisliste* eingegeben haben.
   - Aktivieren Sie das Kontrollfeld *Optionale Gruppe*, wenn die Installation dieser Dateigruppe als Option angeboten werden soll.
   - Geben Sie nun unter *Hinzugefügte Dateien* die Dateien an, die in die Gruppe aufgenommen werden sollen. Zur Bearbeitung der Liste stehen Ihnen die Schaltflächen *Hinzufügen*, *Löschen* und *Auto* zur Verfügung. Mit der Schaltfläche *Auto* können Sie alle Dateien mit einer bestimmten Namenserweiterung auswählen, die in der **contains**-Klausel des Package aufgeführt sind. Der Editor für Package-Sammlungen sucht diese Dateien im globalen Bibliothekssuchpfad von Delphi.

5 Sie können auch Installationsverzeichnisse für die Packages auswählen, die in der **requires**-Klausel eines in der Sammlung enthaltenen Package aufgeführt sind. Wenn Sie eine BPL-Datei im Baumdiagramm markieren, werden auf der rechten Seite des Editors vier neue Felder angezeigt.

   - Im Listenfeld *Verzeichnis der benötigten EXE-Dateien* geben Sie das Verzeichnis an, in dem die BPL-Dateien für Packages installiert werden sollen, die in der **requires**-Klausel enthalten sind. In dieser Dropdown-Liste sind die Verzeichnisse aufgeführt, die Sie in Schritt 3 unter *Verzeichnisliste* angegeben haben. Der Editor für Package-Sammlungen sucht diese Dateien im globalen Bibliothekssuchpfad von Delphi und zeigt sie unter *Benötigte ausführbare Dateien* an.

- Im Listenfeld *Verzeichnis der benötigten Bibliotheken* legen Sie das Verzeichnis fest, in dem die DCP-Dateien für Packages installiert werden sollen, die in der **requires**-Klausel enthalten sind. In dieser Dropdown-Liste sind die Verzeichnisse aufgeführt, die Sie in Schritt 3 unter *Verzeichnisliste* angegeben haben. Der Editor für Package-Sammlungen sucht diese Dateien im globalen Bibliotheks-suchpfad von Delphi und zeigt sie unter *Benötigte Bibliotheksdateien* an.

6 Wählen Sie *Datei / Speichern*, um die Quelldatei der Package-Sammlung zu speichern. Für Quelldateien von Package-Sammlungen vergeben Sie die Namenserweiterung PCE.

7 Klicken Sie auf die Schaltfläche *Package compilieren*, um die Package-Sammlung zu compilieren. Der Editor für Package-Sammlungen erstellt eine DPC-Datei mit dem Basisnamen der PCE-Quelldatei. Wenn Sie die Quelldatei zu diesem Zeitpunkt noch nicht gespeichert haben, werden Sie vom Editor vor dem Compilieren zur Eingabe eines Dateinamens aufgefordert.

Um eine bestehende PCE-Datei zu bearbeiten oder neu zu compilieren, wählen Sie im Editor für Package-Sammlungen den Befehl *Datei / Öffnen*.

Kapitel

10

Anwendungen für den internationalen Markt

In diesem Kapitel werden die Richtlinien besprochen, die Sie beim Schreiben von Anwendungen für den internationalen Markt einhalten sollten. Durch entsprechende Vorausplanung können Sie den Aufwand an Zeit und Quelltext, der erforderlich ist, damit Ihr Programm auch in einer fremdsprachigen Umgebung funktioniert, erheblich verringern.

Internationalisierung und Lokalisierung

Um eine Anwendung zu erstellen, die für den internationalen Vertrieb geeignet ist, müssen Sie zwei grundsätzliche Schritte ausführen:

- Internationalisierung
- Lokalisierung

Internationalisierung

Unter Internationalisierung versteht man den Vorgang, bei dem ein Programm darauf vorbereitet wird, in Umgebungen mit unterschiedlichen Gebietsschemata ausgeführt zu werden. Unter einem Gebietsschema versteht man die Benutzerumgebung unter Berücksichtigung des Kulturkreises und der Sprache des betreffenden Landes. Windows unterstützt eine Vielzahl von Gebietsschemata, die durch eine Kombination aus Sprache und Land definiert sind.

Lokalisierung

Unter Lokalisierung versteht man den Vorgang, bei dem eine Anwendung übersetzt und auf den Einsatz in einem bestimmten Kulturkreis vorbereitet wird. Neben der Übersetzung der Zeichenfolgen, die in der Benutzeroberfläche vorkommen, gehört zur Lokalisierung auch die Übersetzung weiterer landestypischer Elemente. Eine Anwendung aus dem Finanzbereich müßte beispielsweise in verschiedenen Ländern unterschiedliche Steuersätze berücksichtigen.

Internationalisieren von Anwendungen

Es ist nicht weiter schwierig, Anwendungen für den internationalen Markt aufzubereiten. Folgende Schritte sind erforderlich:

1. Sie müssen den Quelltext auf die Verarbeitung von Strings aus internationalen Zeichensätzen vorbereiten.
2. Sie müssen die Benutzeroberfläche so gestalten, daß sie mit allen von der Lokalisierung herrührenden Änderungen harmoniert.
3. Sie müssen die zu lokalisierenden Ressourcen auslagern.

Quelltext anpassen

Sie müssen dafür sorgen, daß Ihre Anwendung alle Arten von Strings verarbeiten kann, die ihr in unterschiedlichen Gebietsschemata begegnen können.

Zeichensätze

Windows arbeitet in den meisten Ländern Europas und in den USA mit dem ANSI Latin-1 (1252)-Zeichensatz. Andere Windows-Versionen verwenden hingegen einen anderen Zeichensatz. Die japanische Version arbeitet beispielsweise mit dem Zeichensatz Shift-Jis (Code-Page 932), in dem die japanischen Schriftzeichen als ein oder zwei Bytes codiert sind.

OEM- und ANSI- Zeichensatz

Manchmal müssen Strings zwischen dem Windows-Zeichensatz (dem ANSI-Zeichensatz) und dem Zeichensatz, der in der Code-Page auf dem Rechner des Anwenders eingestellt ist (dem sogenannten OEM-Zeichensatz) konvertiert werden.

Doppelbyte-Zeichensätze

Bei den ideografischen Zeichensätzen Asiens ist eine einfache 1:1-Abbildung zwischen den Zeichen der Sprache und dem 1 Byte großen Datentyp *Char* nicht möglich. Diese Sprachen enthalten mehr Zeichen, als mit dem Datentyp *Char* dargestellt werden können. Statt dessen werden die Zeichen durch eine Mischung aus ein und zwei Byte großen Codes dargestellt.

Das erste Byte eines 2-Byte-Codes stammt aus einem reservierten Wertebereich, der vom jeweiligen Zeichensatz abhängig ist. Im zweiten Byte dürfen dieselben Werte vorkommen wie in 1-Byte-Codes, aber auch Werte aus dem Bereich, der für das erste Byte eines 2-Byte-Codes reserviert ist. Deshalb gibt es nur eine Möglichkeit, um festzustellen, ob ein bestimmtes Byte in einem String ein selbständiges Zeichen darstellt oder Teil eines 2-Byte-Codes ist: Man muß den String vom Anfang her auswerten und ihn entsprechend zerlegen, wenn man auf das einleitende Byte für einen 2-Byte-Code stößt.

Wenn Sie eine Anwendung für asiatische Gebietsschemata entwickeln, dürfen Sie deshalb nur mit String-Funktionen arbeiten, die Strings in 1- und 2-Byte-Zeichen zerlegen können. Delphi stellt hierfür eine Reihe von Laufzeitbibliotheks-Funktionen zur Verfügung. Sie sind nachfolgend Tabelle aufgeführt:

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrIComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrLIComp	CharToByteLen	

Denken Sie daran, daß die String-Länge in Byte nicht unbedingt mit der String-Länge in Zeichen identisch ist. Achten Sie darauf, beim Aufteilen von Strings keine 2-Byte-Codes auseinanderzureißen. Übergeben Sie in Funktions- oder Prozeduraufrufen keine Zeichen als Parameter, weil die Größe eines Zeichens nicht im voraus bekannt ist. Übergeben Sie statt dessen einen Zeiger auf ein Zeichen oder einen String.

16-Bit-Zeichen

Ein anderer Ansatz zur Verarbeitung von ideografischen Zeichensätzen besteht darin, ein 16-Bit-Codierungsverfahren wie Unicode zu verwenden. Da 16-Bit-Zeichen (*Wide-Zeichen*) nicht aus einem, sondern aus zwei Bytes bestehen, kann ein entsprechender Zeichensatz wesentlich mehr verschiedene Zeichen darstellen.

Das 16-Bit-Codierungsverfahren hat den Vorteil, daß wesentlich mehr der üblichen Annahmen über Strings gelten, als das in MBCS-Systemen der Fall ist. Es besteht ein direkter Zusammenhang zwischen der Anzahl der Bytes und der Anzahl der Zeichen in einem String. Sie brauchen sich keine Gedanken darüber zu machen, ob Sie verse-

hentlich ein Zeichen auseinanderreißen oder die zweite Hälfte eines Zeichens fälschlicherweise als Beginn eines anderen Zeichens interpretieren.

Der größte Nachteil der 16-Bit-Zeichensätze besteht darin, daß Windows 95 keine API-Aufrufe für diese Zeichensätze unterstützt. Aus diesem Grund stellen VCL-Komponenten alle String-Werte als Byte- oder MBCS-Strings dar. Wenn bei jedem Zugriff auf eine String-Eigenschaft eine Übersetzung zwischen einem 16-Bit-Zeichensatz und dem MBCS-Zeichensatz notwendig wäre, würde das große Mengen an zusätzlichem Quelltext erfordern, und die Anwendungen würden deutlich langsamer werden. Es kann sich allerdings lohnen, Strings für bestimmte Algorithmen zur String-Verarbeitung, welche die 1:1-Zuordnung zwischen den Zeichen und dem Typ *WideChar* ausnutzen, entsprechend zu konvertieren.

Bidirektionale Sprachen

In bestimmten Sprachen gilt nicht die Leserichtung von links nach rechts, wie sie in westlichen Sprachen üblich ist. Statt dessen werden Wörter von rechts nach links und Zahlen von links nach rechts gelesen. Man bezeichnet diese Sprachen deshalb als bidirektionale Sprachen. Neben den bekannten bidirektionalen Sprachen Arabisch und Hebräisch sind im Nahen Osten weitere bidirektionale Sprachen gebräuchlich.

TApplication verfügt über zwei Eigenschaften, *BiDiKeyboard* und *NonBiDiKeyboard*, mit deren Hilfe sich das Tastaturlayout festlegen läßt. Außerdem unterstützt die VCL Lokalisierungen für bidirektionale Sprachen durch die Eigenschaften *BiDiMode* und *ParentBiDiMode*. In der folgenden Tabelle sind die VCL-Objekte aufgeführt, die über diese Eigenschaften verfügen.

Tabelle 10.1 VCL-Objekte, die bidirektionale Sprachen unterstützen

Registerkarte der Komponentenpalette	VCL-Objekt
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar
Zusätzlich	TBitBtn
	TCheckListBox
	TDrawGrid

Tabelle 10.1 VCL-Objekte, die bidirektionale Sprachen unterstützen (Fortsetzung)

Registerkarte der Komponentenpalette	VCL-Objekt
	TMaskEdit
	TScrollBox
	TSpeedButton
	TStaticLabel
	TStringGrid
Win32	TDateTimePicker
	THeaderControl
	TListView
	TMonthCalendar
	TPageControl
	TRichEdit
	TStatusBar
	TTabControl
Datensteuerung	TDBCheckBox
	TDBComboBox
	TDBEdit
	TDBGrid
	TDBListBox
	TDBLookupComboBox
	TDBLookupListBox
	TDBMemo
	TDBRadioGroup
	TDBRichEdit
	TDBText
QReport	TQRDBRichText
	TQRDBText
	TQRExpr
	TQRLabel
	TQRMemo
	TQRRichText
	TQRSysData
Andere Klassen	TApplication (verfügt nicht über *ParentBiDiMode)*
	TForm
	THintWindow (verfügt nicht über *ParentBiDiMode)*
	TStatusPanel
	THeaderSection

Hinweis *THintWindow* übernimmt den Wert der Eigenschaft *BiDiMode* des Steuerelements, das den Hinweis aktiviert hat.

Bidirektionale Eigenschaften

Die in Tabelle 10.1, »VCL-Objekte, die bidirektionale Sprachen unterstützen,« auf Seite 10-4 aufgeführten Objekte verfügen über die Eigenschaften *BiDiMode* and *ParentBiDiMode*. Zusammen mit den *TApplication*-Eigenschaften *BiDiKeyboard* und *NonBiDiKeyboard* unterstützen sie Lokalisierungen für bidirektionale Sprachen.

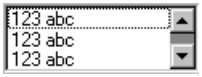
Die Eigenschaft BiDiMode

Die Eigenschaft *BiDiMode* ist ein neuer Aufzählungstyp mit dem Namen *TBiDiMode*, der vier Zustände unterstützt: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign* und *bdRightToLeftReadingOnly*.

bdLeftToRight

Im Status *bdLeftToRight* wird Text von links nach rechts eingegeben. Ausrichtung und Bildlaufleisten ändern sich nicht. Wenn beispielsweise arabischer oder hebräischer Text eingegeben wird, für den die Rechts-/Links-Leserichtung gilt, wird der Cursor in den Einfügemodus versetzt, und die Eingabe erfolgt von rechts nach links. In lateinischen Sprachen (Englisch, Französisch usw.) erfolgt die Texteingabe dagegen von links nach rechts. *bdLeftToRight* ist der voreingestellte Wert.

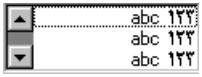
Abbildung 10.1 *TListBox* mit *bdLeftToRight*



bdRightToLeft

Im Status *bdRightToLeft* wird Text von rechts nach links eingegeben. Ausrichtung und Bildlaufleiste ändern sich. Die Eingabe von Text erfolgt für Sprachen mit Rechts-/Links-Leserichtung (Arabisch, Hebräisch usw.) korrekt. Bei lateinischen Sprachen wird der Cursor in den Einfügemodus versetzt, und die Texteingabe erfolgt von links nach rechts.

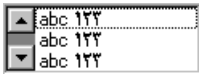
Abbildung 10.2 *TListBox* mit *bdRightToLeft*



bdRightToLeftNoAlign

Im Status *bdRightToLeftNoAlign* wird Text von rechts nach links eingegeben. Die Ausrichtung ändert sich nicht. Die Bildlaufleiste wird versetzt.

Abbildung 10.3 *TListBox* mit *bdRightToLeftNoAlign*



bdRightToLeftReadingOnly

Im Status *bdRightToLeftReadingOnly* wird Text von rechts nach links eingegeben. Ausrichtung und Bildlaufleisten ändern sich nicht.

Abbildung 10.4 *TListBox* mit *bdRightToLeftReadingOnly*

Die Eigenschaft ParentBiDiMode

ParentBiDiMode ist eine Boolesche Eigenschaft, die standardmäßig mit *True* belegt ist. Ein Steuerelement, dessen Eigenschaft *ParentBiDiMode* den Wert *True* hat, übernimmt die *BiDiMode*-Einstellung seines übergeordneten Steuerelements. Wenn es sich bei dem Steuerelement um ein *TForm*-Objekt handelt, übernimmt das Formular die *BiDiMode*-Einstellung von *Application*. Wenn die *BiDiMode*-Einstellung von *Application* geändert wird, während alle *ParentBiDiMode*-Eigenschaften auf *True* gesetzt sind, übernehmen alle Formulare und Steuerelemente im Projekt die neue Einstellung.

Die Methode FlipChildren

Mit der Methode *FlipChildren* können Sie die Positionen der untergeordneten Elemente eines Container-Steuerelements vertauschen. Ein Container-Steuerelement ist ein Element, das andere Steuerelemente enthalten kann (z.B. *TForm*, *TPanel* und *TGroupbox*). *FlipChildren* übernimmt einen Booleschen Parameter namens *AllLevels*. Wenn dieser den Wert *False* hat, werden nur die Positionen von Steuerelementen vertauscht, die dem Container-Element unmittelbar untergeordnet sind. Hat *AllLevels* dagegen der Wert *True*, erstreckt sich der Positionstausch auf alle untergeordneten Ebenen des Container-Steuerelements.

Der Positionstausch erfolgt durch eine Änderung der Eigenschaft *Left* und der Ausrichtung des Steuerelements. Wenn z.B. der linke Rand eines Steuerelements fünf Pixel vom linken Rand des übergeordneten Steuerelements entfernt ist, befindet sich nach dem Positionstausch der rechte Rand des untergeordneten Elements fünf Pixel neben dem rechten Rand des übergeordneten Elements. Wenn Sie *FlipChildren* für ein Eingabefeld mit linksbündiger Ausrichtung aufrufen, wird der Inhalt des Feldes rechtsbündig ausgerichtet.

Während des Entwurfs können Sie mit *Bearbeiten / Untergeordnete Elemente vertauschen* einen Positionstausch durchführen. Wählen Sie entweder *Alle* oder *Markierte*, je nachdem, ob die Positionen aller Steuerelemente oder nur die der untergeordneten Elemente des ausgewählten Elements vertauscht werden sollen. Sie können den Positionstausch auch durchführen, indem Sie das betreffende Steuerelement im Formular markieren, mit der rechten Maustaste klicken und im lokalen Menü den Befehl *Untergeordnete Elemente vertauschen* wählen.

Hinweis Eingabefelder werden nicht als Container-Steuerelemente betrachtet. Wenn Sie ein Eingabefeld markieren und den Befehl *Untergeordnete Elemente vertauschen / Markierte* wählen, geschieht nichts.

Weitere Methoden

Die folgende Tabelle enthält weitere Methoden, die bei der Entwicklung von bidirektionalen Anwendungen hilfreich sind.

Methode	Beschreibung
OkToChangeFieldAlignment	Wird bei Datenbank-Steuerelementen verwendet. Überprüft, ob die Ausrichtung eines Elements geändert werden kann.
DBUseRightToLeftAlignment	Kapselt die Überprüfung der Ausrichtung für Datenbank-Steuerelemente.
ChangeBiDiModeAlignment	Ändert den übergebenen Ausrichtungsparameter. Die Einstellung von *BiDiMode* wird nicht überprüft. Nur die Ausrichtung wird vertauscht (linksbündig in rechtsbündig und umgekehrt). Zentrierte Steuerelemente werden nicht verändert.
IsRightToLeft	Liefert *True*, wenn eine Rechts-/Links-Option ausgewählt ist. Bei *False* befindet sich das Steuerelement im Links-/Rechts-Modus.
UseRightToLeftReading	Liefert *True*, wenn für das Steuerelement die Leserichtung von rechts nach links eingestellt ist.
UseRightToLeftAlignment	Liefert *True*, wenn für das Steuerelement die Rechts-/Links-Ausrichtung eingestellt ist. Diese Methode kann überschrieben und angepaßt werden.
UseRightToLeftScrollBar	Liefert *True*, wenn das Steuerelement eine linke Bildlaufleiste enthält.
DrawTextBiDiModeFlags	Liefert die korrekten Textausgabe-Flags für die *BiDiMode*-Einstellung des Steuerelements.
DrawTextBiDiModeFlagsReadingOnly	Liefert die korrekten Textausgabe-Flags für die *BiDiMode*-Einstellung des Steuerelements, wobei nur die Leserichtung beachtet wird.
AddBiDiModeExStyle	Fügt dem Steuerelement, das gerade erstellt wird, die entsprechenden *ExStyle*-Flags hinzu.

Funktionen für bestimmte Gebietsschemas

Manchmal möchte man Anwendungen mit zusätzlichen Funktionen versehen, die nur bei bestimmten Gebietsschemata zur Verfügung stehen. Besonders in Umgebungen mit asiatischen Sprachen kann es wünschenswert sein, daß die Anwendung den sogenannten Input Method Editor (IME) steuert. Der IME konvertiert die Tastatureingaben des Anwenders in Strings.

VCL-Komponenten unterstützen die Programmierung des IME. Die meisten fensterorientierten Steuerelemente, die unmittelbar mit Texteingaben zu tun haben, verfügen über die Eigenschaft *ImeName*, mit deren Hilfe Sie angeben können, welcher spezielle IME verwendet werden soll, wenn das Steuerelement den Fokus hat. Diese Komponenten unterstützen auch die Eigenschaft *ImeMode*, die festlegt, wie der IME die Tastatureingaben verarbeiten soll. *TWinControl* enthält einige als **protected** deklarierte Methoden, mit deren Hilfe Sie den IME über von Ihnen definierte Klassen steuern können. Zudem liefert die globale Variable *Screen* Informationen über die IMEs, die auf dem Rechner des Anwenders zur Verfügung stehen.

Die globale Variable *Screen* liefert auch Informationen über die aktive Tastenbelegung auf dem Rechner des Anwenders. Auf diese Weise erhalten Sie gebietsschemaspezifische Informationen über die Umgebung, in der Ihre Anwendung ausgeführt wird.

Die Benutzeroberfläche gestalten

Wenn Sie eine Anwendung für den internationalen Markt entwickeln, müssen Sie die Benutzeroberfläche so gestalten, daß sie allen möglichen Änderungen, die bei der Übersetzung vorgenommen werden, Rechnung trägt.

Text

Sämtlicher Text, der in der Benutzeroberfläche erscheint, muß übersetzt werden. Der übersetzte Text kann länger als der Text in der Originalsprache sein, wobei die englische Version oft am kürzesten ist. Gestalten Sie die Elemente der Benutzeroberfläche, in denen Text angezeigt wird, so, daß sie auch etwas längere Text-Strings aufnehmen können. Dies betrifft beispielsweise Dialogfelder, Menüs, Statusleisten und anderes mehr. Vermeiden Sie Abkürzungen, weil Sprachen mit ideografischen Zeichensätzen keine Abkürzungen kennen.

Kurze Strings neigen bei der Übersetzung eher als lange Strings dazu, an Umfang zuzulegen. Tabelle 10.2 zeigt am Beispiel des Englischen, wieviel zusätzlichen Platz man für die übersetzten Strings einkalkulieren sollte:

Tabelle 10.2 Geschätzte String-Länge

Länge der englischen Strings (in Zeichen)	Zu erwartender Zuwachs
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
über 50	10%

Grafiken

Im Idealfall sollten Sie Grafiken verwenden, die nicht lokalisiert werden müssen. Das heißt natürlich in erster Linie, daß Ihre Grafiken keinen Text enthalten sollten, weil dieser immer übersetzt werden muß. Wenn Sie in Ihren Grafiken nicht auf Text verzichten können, sollten Sie den Text nicht in das Bild selbst einfügen, sondern ein *TLabel*-Objekt mit transparentem Hintergrund in der Grafik plazieren.

Beim Anfertigen von Grafiken sind noch weitere Gesichtspunkte zu beachten. Versuchen Sie, auf Bilder zu verzichten, die nur in einem bestimmten Kulturkreis verstanden werden. Briefkästen beispielsweise sehen in den verschiedenen Ländern sehr unterschiedlich aus. Religiöse Symbole sind in Ihrer Anwendung fehl am Platz, wenn sie für Länder gedacht ist, in denen eine andere Religion vorherrscht. Selbst Farben haben in unterschiedlichen Kulturkreisen eine unterschiedliche symbolische Bedeutung.

Formate und Sortierreihenfolge

Die Formate für Datum, Uhrzeit, Zahlen und Währungsangaben sollten für das anvisierte Gebietsschema lokalisiert werden. Wenn Sie ausschließlich Windows-Formate verwenden, brauchen Sie nichts zu übersetzen, weil diese Formate der Windows-Systemregistrierung auf dem Rechner des Anwenders entnommen werden. Falls Sie jedoch eigene Format-Strings benutzen, sollten Sie diese als Ressourcenkonstanten deklarieren, damit sie einfach lokalisiert werden können.

Die Reihenfolge, in der Strings sortiert werden, variiert ebenfalls von Land zu Land. Viele europäische Sprachen enthalten diakritische Zeichen (wie die deutschen Umlaute), die je nach Gebietsschema unterschiedlich einsortiert werden. Dazu kommt, daß in manchen Ländern Kombinationen aus zwei Zeichen für die Sortierreihenfolge wie ein einziges Zeichen behandelt werden. Im Spanischen beispielsweise wird die Zeichenkombination *ch* wie ein einziger Buchstabe behandelt und zwischen *c* und *d* einsortiert. Manchmal wird auch ein einzelnes Zeichen so behandelt, als bestünde es aus zwei Zeichen, wie der deutsche Buchstabe *ß*.

Tastenzuordnungen

Seien Sie vorsichtig, wenn Sie Tastenkombinationen für Zugriffstasten zuweisen. Nicht alle Zeichen auf der deutschen Tastatur lassen sich auf fremdsprachigen Tastaturen ohne weiteres eingeben. Verwenden Sie möglichst Ziffern- und Funktionstasten als Zugriffstasten, weil diese auf praktisch allen Tastaturen zur Verfügung stehen.

Ressourcen auslagern

Der offensichtlichste Teil bei der Lokalisierung einer Anwendung besteht darin, die Strings zu übersetzen, die in der Benutzeroberfläche erscheinen. Damit eine Anwendung übersetzt werden kann, ohne daß überall Eingriffe in den Quelltext erforderlich sind, sollten die Strings der Benutzeroberfläche in einem einzigen Modul zusammengefaßt werden. Delphi erstellt automatisch eine DFM-Datei, welche die Ressourcen von Menüs, Dialogfeldern und Bitmap-Grafiken enthält.

Außer diesen offensichtlichen Bestandteilen der Benutzeroberfläche müssen Sie alle Strings auslagern, die Sie dem Anwender präsentieren, beispielsweise Fehlermeldungen. String-Ressourcen werden nicht in die DFM-Datei eingebunden. Sie können sie aber in einer RC-Datei zusammenfassen.

Ressourcen-DLLs erstellen

Das Auslagern von Ressourcen vereinfacht den Übersetzungsprozeß. Eine fortgeschrittene Stufe der Auslagerung besteht im Erstellen von Ressourcenmodulen. Ein Ressourcenmodul ist eine DLL, die alle Ressourcen – und nur diese – eines Programms enthält. Eine Anwendung, die Ressourcenmodule verwendet, unterstützt eine Vielzahl von Übersetzungen, indem einfach das Ressourcenmodul ausgetauscht wird.

Mit dem Ressourcen-DLL-Experten können Sie für Ihre Anwendung ein Ressourcenmodul erstellen. Er steht Ihnen zur Verfügung, wenn Sie ein Projekt geöffnet, compiliert und gespeichert haben. Der Experte erstellt eine RC-Datei, in der die String-Tabellen der verwendeten RC-Dateien und die **resourcestring**-Strings des Projekts enthalten sind. Außerdem generiert der Experte ein Projekt für eine Ressourcen-DLL, das alle relevanten Formulare und die erstellte RES-Datei enthält. Die RES-Datei wird aus der neuen RC-Datei compiliert.

Für jede Übersetzung, die Sie unterstützen wollen, sollte Ihre Anwendung über ein Ressourcenmodul verfügen. Damit der Dateinamenserweiterung das entsprechende Gebietsschema zu entnehmen ist, sollten die beiden ersten Zeichen auf die Sprache und das dritte Zeichen auf das Land hinweisen. Mit dem folgenden Quelltext können Sie den Gebietsschemacode der Zielsprache ermitteln:

```
unit locales;
interface
uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
   StdCtrls;
type
   TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
   private
    { private-Deklarationen }
   public
    { public-Deklarationen }
   end;
var
   Form1: TForm1;
implementation
{$R *.DFM}
function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
   BufSize: Integer;
begin
   BufSize := GetLocaleInfo(ID, Flag, nil, 0);
   SetLength(Result, BufSize);
   GetLocaleinfo(ID, Flag, PChar(Result), BufSize);
   SetLength(Result, BufSize - 1);
end;
{ Wird für jedes unterstützte Gebietsschema aufgerufen. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
   LCID: Integer;
begin
   LCID := StrToInt('$' + Copy(Name, 5, 4));
   Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
   Result := Bool(1);
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
   EnumSystemLocales(@LocalesCallback, LCID_SUPPORTED);
```

```
end;
end.
```

Ressourcen-DLLs verwenden

Die ausführbare Datei, die DLLs und die Packages, aus denen Ihre Anwendung besteht, enthalten alle benötigten Ressourcen. Um diese Ressourcen durch lokalisierte Versionen zu ersetzen, brauchen Sie Ihrer Anwendung lediglich lokalisierte Ressourcen-DLLs beizulegen, die dieselben Namen haben wie Ihre EXE-, DLL- oder DPL-Dateien.

Beim Laden ermittelt Ihre Anwendung das Gebietsschema des Rechners, auf dem sie ausgeführt wird. Wenn sie Ressourcen-DLLs findet, die dieselben Namen haben wie ihre EXE-, DLL- oder DPL-Dateien, überprüft sie die Dateinamenserweiterung dieser DLLs. Falls die Dateinamenserweiterung eines Ressourcenmoduls mit dem Gebietsschemacode des Rechners übereinstimmt, verwendet Ihre Anwendung anstelle der Ressourcen in den EXE-, DLL- oder DPL-Dateien die Ressourcen aus dem Modul. Falls für diese Kombination aus Sprache und Land kein Ressourcenmodul vorhanden ist, sucht die Anwendung nach einem Ressourcenmodul, das wenigstens die geforderte Sprache unterstützt. Fehlt auch ein solches Modul, verwendet die Anwendung die Ressourcen aus der ausführbaren Datei, der DLL oder dem Package.

Wenn Ihre Anwendung ein anderes Ressourcenmodul verwenden soll als das, welches dem Gebietsschema des Rechners entspricht, können Sie dies durch einen Eintrag in der Windows-Registrierung erreichen. Fügen Sie unter dem Schlüssel HKEY_CURRENT_USER\Software\Borland\Locales den Pfad und den Dateinamen Ihrer Anwendung als String hinzu, und tragen Sie in der Spalte *Wert* die Dateinamenserweiterung Ihrer Ressourcen-DLL ein. Die Anwendung sucht beim Start nach Ressourcen-DLLs mit dieser Erweiterung, bevor sie das Gebietsschema des Systems verwendet. Indem Sie diesen Registrierungseintrag setzen, können Sie lokalisierte Versionen Ihrer Anwendung testen, ohne das Gebietsschema des Systems zu ändern.

Die folgende Funktion kann beispielsweise in einem Installations- oder Setup-Programm verwendet werden. Sie setzt in der Registrierung den Schlüsselwert, der das Gebietsschema festlegt, das beim Laden von Delphi-Anwendungen verwendet werden soll:

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software\Borland\Locales', True) then
      Reg.WriteString(LocalOverride, FileName);
  finally
    Reg.Free;
end;
```

In der Anwendung können Sie mit der globalen Funktion *FindResourceHInstance* das Handle des aktuellen Ressourcenmoduls ermitteln. Ein Beispiel:

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

Auf diese Weise können Sie eine Anwendung ausliefern, die sich automatisch an das Gebietsschema des Rechners anpaßt, auf dem sie ausgeführt wird. Sie benötigen dazu lediglich die entsprechenden Ressourcen-DLLs.

Ressourcen-DLLs dynamisch wechseln

Eine Ressourcen-DLL kann nicht nur beim Start einer Anwendung geladen werden, sondern es ist auch möglich, die verwendete Ressourcen-Datei zur Laufzeit zu wechseln. Wenn Sie Ihre Anwendung mit dieser Funktionalität versehen wollen, nehmen Sie in die **uses**-Anweisung die Unit *ReInit* auf. (Diese Unit befindet sich im Verzeichnis Demos im Richedit-Beispiel.) Um die verwendete Sprache zu wechseln, übergeben Sie in einem Aufruf von *LoadResourceModule* die LCID für die neue Sprache und rufen anschließend *ReinitializeForms* auf.

Mit den folgenden Anweisungen können Sie die Sprache der Benutzeroberfläche in Französisch ändern:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

Dieses Vorgehen hat den Vorteil, daß die aktuelle Instanz der Anwendung mit allen zugehörigen Formularen verwendet werden kann. Dabei ist weder eine Aktualisierung von Registrierungseinstellungen mit anschließendem Neustart der Anwendung noch die erneute Anforderung von benötigten Ressourcen (z.B. das Anmelden bei einem Datenbank-Server) erforderlich.

Bei einem Wechsel der Ressourcen-DLL werden die Eigenschaftswerte in den laufenden Instanzen der Formulare durch die entsprechenden Eigenschaftswerte der neuen DLL ersetzt.

Hinweis Änderungen, die zur Laufzeit an den Formulareigenschaften vorgenommen wurden, gehen verloren. Nach dem Laden der neuen DLL werden keine Standardwerte zurückgesetzt. Sie dürfen deshalb im Quelltext nie davon ausgehen, daß die Formularobjekte automatisch mit ihrem Anfangsstatus neu initialisiert werden.

Anwendungen lokalisieren

Nachdem Sie Ihre Anwendung für die Lokalisierung vorbereitet (internationalisiert) haben, können Sie lokalisierte Versionen für die verschiedenen ausländischen Märkte erstellen, in denen Sie die Anwendung vertreiben wollen.

Ressourcen lokalisieren

Wenn möglich, sollten Sie die Ressourcen in eine gesonderte Ressourcen-DLL auslagern, die DFM-Dateien und eine RES-Datei enthält. Sie können Ihre Formulare in der IDE öffnen und die relevanten Eigenschaften übersetzen.

Hinweis In einem Ressourcen-DLL-Projekt können Komponenten weder gelöscht noch hinzugefügt werden. Da es aber möglich ist, die Werte von Eigenschaften in einer Weise zu ändern, daß daraus Laufzeitfehler resultieren, sollten Sie nur diejenigen Eigenschaften bearbeiten, die einer Übersetzung bedürfen. Zur Vermeidung entsprechender Fehler läßt sich der Objektinspektor so konfigurieren, daß nur lokalisierbare Eigenschaften angezeigt werden; dazu klicken Sie ihn mit der rechten Maustaste an und verwenden das Menü *Ansicht*, um alle unerwünschten Eigenschaften herauszufiltern.

Übersetzen Sie die relevanten Strings in der RC-Datei. Sie können dazu den Stringtabellen-Editor verwenden, indem Sie die RC-Datei von der Projektverwaltung aus öffnen.

Wenn in Ihrer Delphi-Version die Integrated Translation Engine enthalten ist, lassen sich damit Ihre Lokalisierungsvorhaben verwalten. Informationen dazu finden Sie in der Online-Hilfe.

Kapitel

11

Anwendungen weitergeben

Wenn eine Delphi-Anwendung fertiggestellt und lauffähig ist, kann sie weitergegeben werden. Unter Weitergabe versteht man die Bereitstellung der Anwendung für andere Benutzer. Damit eine Anwendung auf anderen Computern ausgeführt werden kann, müssen gewisse Vorkehrungen getroffen werden. Die Schritte, die für die Weitergabe einer Anwendung erforderlich sind, hängen vom Typ der Anwendung ab. In den folgenden Abschnitten werden diese Schritte ausführlich erläutert.

- Allgemeine Anwendungen weitergeben
- Datenbankanwendungen weitergeben
- Web-Anwendungen weitergeben
- Unterschiedliche Host-Umgebungen berücksichtigen
- Software-Lizenzvereinbarungen

Allgemeine Anwendungen weitergeben

Zusätzlich zur ausführbaren Datei werden für eine Anwendung oft weitere, unterstützende Dateien benötigt. Dazu gehören z.B. DLLs, Package-Dateien und Hilfsprogramme. Außerdem müssen vielleicht in der Windows-Registrierung anwendungsspezifische Einträge vorgenommen werden, z.B. zur Angabe der Position benötigter Dateien oder zur Festlegung von Programmeinstellungen. Das Kopieren der Anwendungsdateien auf einen Computer und das Eintragen der erforderlichen Einstellungen in der Registrierung kann durch ein Installationsprogramm (z.B. InstallShield Express) automatisiert werden. Bei der Weitergabe der meisten Anwendungsarten sind die folgenden Punkte von grundlegender Bedeutung:

- Installationsprogramme verwenden
- Anwendungsdateien identifizieren

Bei Delphi-Anwendungen, die auf Datenbanken zugreifen oder im Web ausgeführt werden, sind zusätzliche Installationsschritte erforderlich. Weitere Informationen zur Installation von Datenbankanwendungen finden Sie unter »Datenbankanwendungen weitergeben« auf Seite 11-4. Einzelheiten zur Installation von Web-Anwendungen enthält der Abschnitt »Web-Anwendungen weitergeben« auf Seite 11-8. Die Installation von ActiveX-Steuerelementen wird unter »ActiveX-Steuerelemente im Web weitergeben« auf Seite 48-21 erläutert. Informationen über die Weitergabe von CORBA-Anwendungen finden Sie im Abschnitt »CORBA-Anwendungen weitergeben« auf Seite 28-18.

Installationsprogramme verwenden

Einfache Delphi-Anwendungen, die nur aus einer ausführbaren Datei bestehen, lassen sich sehr einfach auf einem Zielcomputer installieren, da in diesem Fall nur die ausführbare Datei kopiert werden muß. Für komplexe Anwendungen, die mehrere Dateien umfassen, ist eine aufwendigere Installation erforderlich, die mit Hilfe eines speziellen Installationsprogramms durchgeführt wird.

Setup-Toolkits automatisieren die Erstellung von Installationsprogrammen. Zusätzlicher Quelltext ist in den meisten Fällen nicht erforderlich. Mit Setup-Toolkits erstellte Installationsprogramme übernehmen verschiedene Aufgaben, die bei der Installation von Delphi-Anwendungen durchgeführt werden müssen (z.B. die ausführbaren und unterstützenden Dateien auf den Host-Computer kopieren, die Einträge in der Windows-Registrierung vornehmen und für Datenbankanwendungen die Borland Database Engine installieren).

InstallShield Express ist ein Setup-Toolkit, das zusammen mit Delphi ausgeliefert wird. Dieses Toolkit ist für die Verwendung mit Delphi und der Borland Database Engine zertifiziert. Da InstallShield Express nicht automatisch zusammen mit Delphi installiert wird, müssen Sie dieses Programm manuell installieren, wenn Sie es zur Erstellung von Installationsprogrammen einsetzen wollen. Führen Sie dazu das Installationsprogramm auf der Delphi-CD aus. Ausführliche Informationen zur Erstellung von Installationsprogrammen mit InstallShield Express finden Sie in der Online-Hilfe von InstallShield Express.

Wenn Sie zur Erstellung eines Installationsprogramms für Delphi-Anwendungen andere Setup-Toolkits einsetzen, sollten Sie darauf achten, daß diese für die Weitergabe der Borland Database Engine zertifiziert sind.

Anwendungsdateien identifizieren

Häufig müssen mit der Anwendung neben der ausführbaren Datei zusätzliche Dateien weitergegeben werden.

- Anwendungsdateien nach Namenserweiterung
- Package-Dateien
- ActiveX-Steuerelemente

Anwendungsdateien nach Namenserweiterung

In der folgenden Tabelle sind die Dateiarten aufgeführt, die eventuell zusammen mit einer Anwendung weitergegeben werden müssen.

Tabelle 11.1 Anwendungsdateien

Dateiart	Namenserweiterung
Programmdateien	.EXE und .DLL
Package-Dateien	.BPL und .DCP
Hilfedateien	.HLP, .CNT und .TOC (falls verwendet)
ActiveX-Dateien	.OCX (werden manchmal von einer DLL unterstützt)
Lokale Tabellendateien	.DBF, .MDX, .DBT, .NDX, .DB, .PX, .Y*, .X*, .MB, .VAL, .QBE

Package-Dateien

Wenn die Anwendung Laufzeit-Packages verwendet, müssen die entsprechenden Package-Dateien zusammen mit der Anwendung weitergegeben werden. InstallShield Express behandelt Package-Dateien bei der Installation wie DLLs, d.h., die Dateien werden kopiert, und in der Windows-Registrierung werden die erforderlichen Einträge vorgenommen. Es empfiehlt sich, die von Borland bereitgestellten Laufzeit-Packages im Verzeichnis WINDOWS\SYSTEM zu installieren. Da dieses Verzeichnis allgemein zugänglich ist, können verschiedene Anwendungen auf eine einzige Instanz der Dateien zugreifen. Packages, die Sie selbst erstellt haben, sollten in demselben Verzeichnis wie die Anwendung installiert werden. Nur die BPL-Dateien müssen weitergegeben werden.

Wenn Sie Packages an andere Programmierer weitergeben, stellen Sie sowohl die BPL- als auch die DCP-Dateien zur Verfügung.

ActiveX-Steuerelemente

Einige der Komponenten von Delphi sind ActiveX-Steuerelemente. Obwohl die Klassen dieser Komponenten in die ausführbare Datei (oder in ein Laufzeit-Package) eingebunden sind, muß die OCX-Datei für die Komponente zusammen mit der Anwendung weitergegeben werden. Zu diesen Komponenten gehören:

- Chart FX, Copyright SoftwareFX Inc.
- VisualSpeller, Copyright Visual Components, Inc.
- Formula One (Tabellenkalkulation), Copyright Visual Components, Inc.
- First Impression (VtChart), Copyright Visual Components, Inc.
- Graph Custom Control, Copyright Bits Per Second Ltd.

ActiveX-Steuerelemente, die Sie selbst erstellt haben, müssen vor der Verwendung auf dem Zielcomputer registriert werden. Installationsprogramme wie InstallShield Express automatisieren diesen Registrierungsvorgang. Zur manuellen Registrierung eines ActiveX-Steuerelements verwenden Sie die Demoanwendung TREGSVR oder

das Microsoft-Dienstprogramm REGSRV32.EXE (nicht in allen Windows-Versionen enthalten).

DLLs, die ein ActiveX-Steuerelement unterstützen, müssen ebenfalls zusammen mit der Anwendung weitergegeben werden.

Hilfsanwendungen

Hilfsanwendungen sind separate Programme, ohne die eine Delphi-Anwendung nicht oder nur zum Teil funktionsfähig ist. Es kann sich dabei um Windows- oder Borland-Programme handeln, oder um Anwendungen von Fremdherstellern. Das InterBase-Dienstprogramm Server Manager ist ein Beispiel für eine Hilfsanwendung. Es übernimmt die Verwaltung von InterBase-Datenbanken, -Benutzern und -Sicherheitsfunktionen.

Wenn eine Anwendung von einer Hilfsanwendung abhängig ist, muß diese zusammen mit der Anwendung weitergegeben werden. Bei der Weitergabe von Hilfsanwendungen sind möglicherweise Lizenzrechte zu beachten. Entsprechende Informationen finden Sie in der Dokumentation der betreffenden Anwendung.

Position von DLL-Dateien

DLL-Dateien, die nur von einer einzigen Anwendung verwendet werden, können in demselben Verzeichnis wie die Anwendung installiert werden. Wird eine DLL jedoch von mehreren Anwendungen genutzt, muß sie sich an einer Position befinden, auf die alle Anwendungen zugreifen können. Häufig werden derartige DLLs entweder im Verzeichnis WINDOWS oder in WINDOWS\SYSTEM bereitgestellt. Besser ist es aber, wenn ein separates Verzeichnis für gemeinsam genutzte DLLs angelegt wird, ähnlich wie dies bei der Installation der BDE (Borland Database Engine) geschieht.

Datenbankanwendungen weitergeben

Die Installation von Anwendungen, die auf Datenbanken zugreifen, erfordert neben dem Kopieren der ausführbaren Datei auf den Host-Computer zusätzliche Arbeitsschritte. Der Datenbankzugriff wird meist durch den Einsatz einer separaten Datenbank-Engine optimiert, deren Dateien nicht in die ausführbare Datei der Anwendung eingebunden werden können. Wenn die Datendateien nicht bereits existieren, müssen sie für die Anwendung verfügbar gemacht werden. Mehrschichtige Datenbankanwendungen erfordern eine noch komplexere Installation, da hier die Dateien, aus denen sich die Anwendung zusammensetzt, meist auf mehreren Computern residieren. Bei Anwendungen mit Datenbankzugriff gibt es folgende Möglichkeiten:

- Die Datenbank-Engine bereitstellen
- MIDAS (Multi-tiered Distributed Application Services)

Die Datenbank-Engine bereitstellen

Der Datenbankzugriff einer Anwendung läßt sich mit Hilfe verschiedener Datenbank-Engines implementieren. So kann etwa die Anwendung die Borland Database Engine oder die Datenbank-Engine eines Fremdherstellers nutzen. Für den nativen Zugriff auf SQL-Datenbanksysteme dient SQL Links (nicht in allen Delphi-Versionen verfügbar). In den folgenden Abschnitten wird die Installation der Datenbankzugriffselemente einer Anwendung beschrieben:

- Borland Database Engine
- Datenbank-Engines von Fremdherstellern
- SQL Links

Borland Database Engine

Für den Datenbankzugriff der Standard-Datenkomponenten von Delphi muß der Zugriff auf die Borland Database Engine gewährleistet sein. Rechte und Einschränkungen hinsichtlich der Weitergabe der BDE werden in der Datei BDEDEPLOY.TXT erläutert.

Borland empfiehlt für die Installation der BDE die Verwendung von InstallShield Express (oder eines anderen zertifizierten Installationsprogramms). InstallShield Express erzeugt die erforderlichen Einträge in der Registrierung und definiert alle Aliasnamen, die für die Anwendung benötigt werden. Die Verwendung eines zertifizierten Installationsprogramms zur Weitergabe von BDE-Dateien ist aus folgenden Gründen wichtig:

- Eine inkorrekte Installation der BDE kann die Funktion anderer Anwendungen, welche die BDE nutzen, beeinträchtigen. Davon können nicht nur Borland-Anwendungen, sondern auch Programme anderer Hersteller betroffen sein, die von der BDE Gebrauch machen.
- Unter Windows 95 und Windows NT werden die BDE-Konfigurationsdaten nicht mehr in INI-Dateien (wie dies in den 16-Bit-Versionen der Fall war), sondern in der Windows-Registrierung gespeichert. Das Festlegen der korrekten Einträge bei der Installation und das Durchführen von Löschungen bei der Deinstallation gestaltet sich sehr komplex.

Es ist möglich, nur diejenigen Komponenten der BDE zu installieren, die zur Ausführung einer Anwendung erforderlich sind. Verwendet eine Anwendung z.B. nur Paradox-Tabellen, braucht nur die BDE-Komponente installiert zu werden, die für den Zugriff auf Paradox-Tabellen zuständig ist. Auf diese Weise läßt sich der von einer Anwendung belegte Festplattenspeicher reduzieren. Eine Teilinstallation der BDE kann mit zertifizierten Installationsprogrammen wie InstallShield Express durchgeführt werden. Entfernen Sie bei einer Teilinstallation keine BDE-Systemdateien, die zwar von der weitergegebenen Anwendung nicht benötigt werden, aber für die Ausführung anderer Programme unbedingt erforderlich sind.

Datenbank-Engines von Fremdherstellern

Der Datenbankzugriff von Delphi-Anwendungen kann auch mit Datenbank-Engines von Fremdherstellern implementiert werden. Informationen über Installation, Konfiguration und Weitergaberechte erhalten Sie in der zugehörigen Dokumentation oder bei Ihrem Software-Händler.

SQL Links

SQL Links stellt die Treiber bereit, die eine Anwendung (über die Borland Database Engine) mit der Client-Software für eine SQL-Datenbank verbinden. Rechte und Einschränkungen hinsichtlich der Weitergabe von SQL Links werden in der Datei DEPLOY.TXT erläutert. SQL Links muß wie die BDE unter Verwendung von InstallShield Express (oder eines anderen zertifizierten Installationsprogramms) weitergegeben werden.

Hinweis SQL Links stellt nur die Verbindung zwischen der BDE und der Client-Software her, nicht jedoch zur SQL-Datenbank selbst. Es muß deshalb sichergestellt sein, daß auch die Client-Software für das verwendete SQL-Datenbanksystem installiert ist. Informationen über die Installation und Konfiguration der Client-Software erhalten Sie in der Dokumentation des verwendeten SQL-Datenbanksystems oder bei Ihrem Software-Händler.

In Tabelle 11.2 sind die Namen der Treiber- und Konfigurationsdateien aufgeführt, mit denen SQL Links eine Verbindung zu den verschiedenen SQL-Datenbanksystemen herstellt. Diese Dateien sind Bestandteil von SQL Links. Ihre Weitergabe kann gemäß den Lizenzvereinbarungen von Delphi erfolgen.

Tabelle 11.2 Client-Software-Dateien für SQL-Datenbanken

Hersteller	Dateien zur Weitergabe
Oracle 7	SQLORA32.DLL und SQL_ORA.CNF
Oracle8	SQLORA8.DLL und SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL und SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL und SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL und SQL_MSS.CNF
Informix 7	SQLINF32.DLL und SQL_INF.CNF
Informix 9	SQLINF9.DLL und SQL_INF9.CNF
DB/2	SQLDB232.DLL und SQL_DB2.CNF
InterBase	SQLINT32.DLL und SQL_INT.CNF

Verwenden Sie zur Installation von SQL Links entweder InstallShield Express oder ein anderes zertifiziertes Installationsprogramm. Ausführliche Informationen zur Installation und Konfiguration von SQL Links finden Sie in der Datei SQLLNK32.HLP (die standardmäßig im BDE-Hauptverzeichnis installiert wird).

MIDAS (Multi-tiered Distributed Application Services)

MIDAS (Multi-tiered Distributed Application Services) setzt sich aus Business Object Broker, OLEnterprise, Remote DataBroker und ConstraintBroker Manager (SQL-Explorer) zusammen. Mit Hilfe von MIDAS können Sie in Ihre Delphi-Anwendungen die Möglichkeiten mehrschichtiger Datenbankanwendungen integrieren.

Die Installation der ausführbaren und unterstützenden Dateien einer mehrschichtigen Anwendung erfolgt auf die gleiche Weise wie bei allgemeinen Anwendungen. Möglicherweise müssen bestimmte MIDAS-Dateien auf dem Client-Computer und andere auf dem Server-Computer installiert werden. Einzelheiten zur Installation allgemeiner Anwendungen finden Sie im Abschnitt »Allgemeine Anwendungen weitergeben« auf Seite 11-1». In der Textdatei LICENSE.TXT auf der MIDAS-CD und in der Delphi-Datei DEPLOY.TXT finden Sie Informationen über Lizenz- und Weitergaberechte.

Auf dem Client-Computer muß die Datei MIDAS.DLL installiert und in Windows registriert werden. Auf dem Server-Computer müssen für Remote DataBroker die Dateien MIDAS.DLL und STDVCL40.DLL und für ConstraintBroker die Datei DBEXPLOR.EXE installiert und registriert werden. Installationsprogramme wie InstallShield Express automatisieren die Registrierung dieser DLLs. Zu manuellen Registrierung der DLLs verwenden Sie die Demoanwendung TREGSVR oder das Microsoft-Dienstprogramm REGSRV32.EXE (nicht in allen Windows-Versionen enthalten).

Auf der MIDAS-CD sind Installationsprogramme für die Client- und Server-Komponenten von OLEnterprise und Business ObjectBroker enthalten. Verwenden Sie zur Installation von OLEnterprise nur den Setup Launcher auf der MIDAS-CD.

Die folgenden Dateien müssen auf dem Server-Computer installiert werden:

UNINSTALL.EXE	OBJFACT.ICO	W32PTHD.DLL	NBASE.IDL
LICENSE.TXT	ODEBKN40.DLL	RPMARN40.DLL	OBJX.EXE
README.TXT	ODECTN40.DLL	RPMAWN40.DLL	OLECFG.EXE
OLENTER.HLP	RPMEGN40.DLL	RPMCBN40.DLL	OLEWAN40.CAB
OLENTER.CNT	ODEDIN40.DLL	RPMCPN40.DLL	OLENTEXP.EXE
FILELIST.TXT	ODEEGN40.DLL	BROKER.EXE	OLENTEXP.HLP
SETLOG.TXT	ODELTN40.DLL	RPMFEN40.DLL	OLENTEXP.CNT
SETLOG.EXE	LIBAVEMI.DLL	RPMUTN40.DLL	BRKCP.EXE
OBJPING.EXE	OLEAAN40.DLL	RPMFE.CAT	BROKER.ICO
OBJFACT.EXE	OLERAN40.DLL	EXPERR.CAT	

Die folgenden Dateien müssen auf dem Client-Computer installiert werden:

NBASE.IDL	ODEN40.DLL	RPMFEN40.DLL	OLENTEXP.EXE
ODECTN40.DLL	RPMARN40.DLL	RPMUTN40.DLL	SETLOG.EXE

ODEDIN40.DLL	RPMAWN40.DLL	OLERAN40.DLL	OLECFG.EXE
ODEEGN40.DLL	RPMCBN40.DLL	OLEAAN40.DLL	W32PTHD.DLL
ODELTN40.DLL	RPMCPN40.DLL	OLEWAN40.CAB	
ODEMSG.DLL	RPMEGN40.DLL	OBJX.EXE	

Web-Anwendungen weitergeben

Bestimmte Delphi-Anwendungen sind für die Ausführung im WWW vorgesehen. Dazu gehören z.B. serverseitige Erweiterungs-DLLs (ISAPI), CGI-Anwendungen und ActiveForms.

Web-Anwendungen werden wie allgemeine Anwendungen installiert, die Weitergabe erfolgt aber nicht an andere Computer, sondern an den Web-Server. Informationen zur Installation allgemeiner Anwendungen finden Sie im Abschnitt »Allgemeine Anwendungen weitergeben« auf Seite 11-1.

Bei der Weitergabe von Web-Anwendungen sind folgende Punkte gesondert zu berücksichtigen:

- Bei einer Datenbankanwendung wird die Borland Database Engine (oder eine andere Datenbank-Engine) zusammen mit den Anwendungsdateien auf dem Web-Server installiert.
- Für den Zugriff auf die Verzeichnisse mit den Anwendungsdateien, der BDE oder den Datenbankdateien müssen die entsprechenden Berechtigungen bestehen.
- Für das Verzeichnis, in dem sich die Anwendung befindet, muß Lese- und Schreibberechtigung bestehen.
- In der Anwendung dürfen die Pfade für den Zugriff auf die Datenbank oder auf andere Dateien nicht hart codiert sein.
- Die Position eines ActiveX-Steuerelements wird durch den Parameter CODEBASE des HTML-Tags <OBJECT> festgelegt.

Unterschiedliche Host-Umgebungen berücksichtigen

Aufgrund der Natur der Windows-Umgebung müssen einige Faktoren berücksichtigt werden, die in Abhängigkeit von den Benutzereinstellungen bzw. der Konfiguration des Computers variieren können. Folgende Faktoren können sich auf eine Anwendung auswirken, die zur Ausführung auf einem anderen Computer weitergegeben wurde:

- Bildschirmauflösung und Farbtiefe
- Schriften
- Windows-Version
- Hilfsanwendungen

- Position der DLLs

Bildschirmauflösung und Farbtiefe

Die Größe des Windows-Desktop und die Anzahl der auf einem Computer verfügbaren Farben ist konfigurierbar und von der installierten Hardware abhängig. Es ist möglich, daß diese Attribute auf einem Weitergabecomputer anders eingestellt sind als auf dem Entwicklungscomputer.

Es gibt verschiedene Möglichkeiten, das Erscheinungsbild einer Anwendung (Fenster-, Objekt- und Schriftgröße) auf Computern zu beeinflussen, die für eine andere Bildschirmauflösung konfiguriert sind:

- Gehen Sie bei der Entwicklung der Anwendung von der niedrigsten Auflösung aus, die auf dem Zielcomputer möglich ist (normalerweise 640 x 480). Lassen Sie die Größe der Objekte unverändert, und passen Sie sie nicht proportional an die Bildschirmanzeige des Host-Computers an. Je höher in diesem Fall die Bildschirmauflösung auf dem Zielcomputer ist, desto kleiner werden die Objekte dargestellt.
- Verwenden Sie während der Entwicklung der Anwendung eine beliebige Auflösung, und sorgen Sie dafür, daß zur Laufzeit alle Formulare und Objekte dynamisch an die Auflösung des Zielcomputers angepaßt werden.
- Verwenden Sie während der Entwicklung der Anwendung eine beliebige Auflösung, und sorgen Sie dafür, daß zur Laufzeit nur die Formulare der Anwendung dynamisch angepaßt werden. Abhängig von der Position der visuellen Steuerelemente müssen die Formulare eventuell bildlauffähig sein, damit die Benutzer Zugriff auf alle Steuerelemente haben.

Anwendungen ohne dynamische Größenanpassung

Wenn die Größe der Formulare und visuellen Steuerelemente einer Anwendung zur Laufzeit nicht dynamisch angepaßt werden soll, entwickeln Sie die Anwendungselemente für eine Zielumgebung mit der niedrigstmöglichen Auflösung. Wird eine Anwendung auf einem Computer ausgeführt, dessen Auflösung niedriger ist als die des Entwicklungscomputers, liegen die Formulare der Anwendung möglicherweise außerhalb des Bildschirmbereichs.

Dazu ein Beispiel: Wenn auf dem Entwicklungscomputer eine Bildschirmauflösung von 1024 x 768 eingestellt ist und Sie ein Formular mit einer Breite von 700 Pixel entwerfen, ist dieses Formular auf dem Windows-Desktop nur teilweise sichtbar, wenn die Anwendung auf einem Rechner mit der geringeren Auflösung von 640 x 480 ausgeführt wird.

Anwendungen mit dynamischer Größenanpassung der Formulare und Steuerelemente

Wenn Sie die Formulare und die visuellen Steuerelemente einer Anwendung dynamisch anpassen, müssen Sie alle Aspekte berücksichtigen, die sich auf diese Größen-

anpassung auswirken können. Nur so ist bei allen Bildschirmauflösungen ein optimales Erscheinungsbild der Anwendung gewährleistet. Nachstehend sind die wichtigsten Faktoren aufgeführt, die bei einer dynamischen Größenanpassung visueller Elemente einer Anwendung zu beachten sind:

- Die Größenänderung von Formularen und visuellen Steuerelementen wird in dem Verhältnis durchgeführt, das sich aus dem Vergleich der Bildschirmauflösung des Entwicklungscomputers mit derjenigen des Zielcomputers (auf dem die Anwendung installiert wird) ergibt. Verwenden Sie eine Konstante, um eine Dimension (Höhe oder Breite) der Bildschirmauflösung auf dem Entwicklungscomputer darzustellen. Die Angabe erfolgt in Pixel. Ermitteln Sie zur Laufzeit die entsprechende Dimension des Zielcomputers mit Hilfe der Eigenschaft *TScreen.Height* bzw. *TScreen.Width* des Bildschirmobjekts. Das Verhältnis zwischen den Auflösungen der beiden Computer erhalten Sie, indem Sie den Wert des Entwicklungscomputers durch den Wert des Zielcomputers dividieren.
- Ändern Sie die Größe der visuellen Elemente der Anwendung (Formulare und Steuerelemente), indem Sie ihre Größe und Position in den Formularen erhöhen oder verringern. Die Änderung erfolgt proportional zur Differenz zwischen der Bildschirmauflösung des Entwicklungs- und des Zielcomputers. Sie können die Größen- und Positionsänderung visueller Elemente in Formularen automatisch vornehmen, indem Sie die Eigenschaft *TCustomForm.Scaled* des Formulars auf *True* setzen und seine Methode *TWincontrol.ScaleBy* aufrufen. Die Methode *ScaleBy* ändert aber nicht die Höhe und die Breite des Formulars. Um die Größe des Formulars zu ändern, müssen Sie die aktuellen Werte der Eigenschaften *Height* und *Width* manuell mit dem Verhältnis der Bildschirmauflösungen multiplizieren.
- Statt die Steuerelemente in einem Formular mit der Methode *TWincontrol.ScaleBy* automatisch zu skalieren, können Sie ihre Größe auch manuell anpassen, indem Sie jedes visuelle Steuerelement in einer Schleife referenzieren und seine Größe und Position festlegen. Die Werte der Eigenschaften *Height* und *Width* für visuelle Steuerelemente werden mit dem Verhältnis der Auflösungen multipliziert. Die Position eines visuellen Steuerelements legen Sie fest, indem Sie seine Eigenschaften *Top* und *Left* mit diesem Verhältnis multiplizieren.
- Wenn eine Anwendung auf einem Computer entwickelt wird, dessen Bildschirmauflösung höher ist als die des Zielcomputers, werden die Schriftgrößen bei der proportionalen Anpassung der visuellen Steuerelemente automatisch angepaßt. Ist die zur Entwurfszeit verwendete Schriftgröße zu klein, kann die Schrift durch die Größenanpassung zur Laufzeit unleserlich werden. Angenommen, die Standard-Schriftgröße für ein Formular ist 8, der Entwicklungscomputer arbeitet mit einer Bildschirmauflösung von 1024x768 und der Zielcomputer mit einer Auflösung von 640x480. Die Dimensionen der visuellen Steuerelemente werden in diesem Fall um den Faktor 0,625 (640 / 1024 = 0,625) reduziert, und die ursprüngliche Schriftgröße von 8 verringert sich auf 5 (8 * 0,625 = 5). Da Windows zur Anzeige von Text in der Anwendung die reduzierte Schriftgröße verwendet, erscheint der Text verstümmelt und unleserlich.
- Die Größe bestimmter visueller Steuerelemente wie *TLabel* und *TEdit* wird dynamisch angepaßt, wenn sich die Schriftgröße für das Steuerelement ändert. Dies kann sich auf weitergegebene Anwendungen auswirken, wenn die Größe von Formularen und Steuerelementen dynamisch geändert wird. Die Anpassung des

Steuerelements aufgrund von Schriftgrößenänderungen addiert sich zur proportionalen Vergrößerung bzw. Verkleinerung, die aus der Bildschirmauflösung resultiert. Sie können diesen Effekt eliminieren, indem Sie die Eigenschaft *AutoSize* der betreffenden Steuerelemente auf *False* setzen.

- Vermeiden Sie die explizite Angabe von Pixel-Koordinaten, wenn Sie direkt auf der Zeichenfläche zeichnen. Ändern Sie die Koordinaten statt dessen unter Verwendung eines Proportionalfaktors, der das Verhältnis zwischen den Auflösungen auf dem Entwicklungs- und dem Zielcomputer wiedergibt. Wenn die Anwendung z.B. auf der Zeichenfläche ein Rechteck mit einer Höhe von 10 und einer Breite von 20 Pixel erstellt, verwenden Sie diese Maßangaben nicht direkt, sondern multiplizieren sie mit dem Auflösungsverhältnis. Dadurch ist sichergestellt, daß das Rechteck bei verschiedenen Bildschirmauflösungen in derselben Größe dargestellt wird.

Unterschiedliche Farbtiefen

Damit die weitergegebene Anwendung das gewünschte Aussehen auch auf Computern behält, die nicht für dieselbe Farbtiefe wie der Entwicklungscomputer konfiguriert sind, sollten Sie Grafiken mit der geringstmöglichen Farbanzahl verwenden. Dies gilt speziell für Symbole auf Steuerelementen, für die immer 16 Farben verwendet werden sollten. Zur Anzeige von Grafiken können Sie entweder mehrere Kopien mit verschiedenen Auflösungen und Farbtiefen bereitstellen oder den Benutzer in der Anwendung auf die Minimalanforderungen hinsichtlich Auflösung und Farben hinweisen.

Schriften

Windows wird mit einem Standardsatz von TrueType- und Raster-Schriftarten ausgeliefert. Berücksichtigen Sie bei der Entwicklung von Anwendungen, die weitergegeben werden sollen, daß auf vielen Computern nur dieser Standardsatz zur Verfügung steht.

Für die Textkomponenten einer Anwendung sollten nur Schriften verwendet werden, die auf allen Weitergabecomputern verfügbar sind.

Sollte die Verwendung einer Schrift, die nicht im Standardsatz enthalten ist, unvermeidbar sein, muß diese Schrift zusammen mit der Anwendung weitergegeben werden. Die Schrift muß entweder vom Installationsprogramm oder von der Anwendung selbst auf dem Zielcomputer installiert werden. Bei der Weitergabe von Schriften, die von einem Fremdhersteller stammen, sind möglicherweise Lizenzrechte zu beachten.

Windows verfügt über eine Funktion, die sicherstellt, daß bei Nichtvorhandensein einer Schrift die ähnlichste auf dem Computer verfügbare Schrift verwendet wird. Dadurch werden zwar Fehler aufgrund nicht vorhandener Schriften vermieden, das Erscheinungsbild der Anwendung kann aber darunter leiden. Deshalb ist es besser, die Anwendung bereits bei der Entwicklung auf alle Eventualitäten vorzubereiten.

Mit den Windows-API-Funktionen *AddFontResource* und *DeleteFontResource* können Sie eine Schrift, die nicht im Standardsatz von Windows enthalten ist, für eine An-

wendung verfügbar machen. Die FOT-Datei für die betreffende Schrift muß zusammen mit der Anwendung weitergegeben werden.

Windows-Version

Wenn Sie in einer Anwendung Windows-API-Funktionen verwenden oder auf Bereiche des Windows-Betriebssystems zugreifen, besteht die Möglichkeit, daß eine Funktion oder Operation unter der Windows-Version des Zielcomputers nicht verfügbar ist. So stehen z.B. Dienste nur im Betriebssystem Windows NT zur Verfügung. Wird eine Anwendung, die als Dienst fungiert oder mit einem Dienst interagiert, unter Windows 95 installiert, tritt ein Fehler auf.

Um unterschiedliche Versionen von Windows zu berücksichtigen, können Sie folgende Vorkehrungen treffen:

- Geben Sie in den Systemanforderungen der Anwendung die Windows-Versionen an, unter denen sie lauffähig ist. Für die Installation und Ausführung der Anwendung in der korrekten Betriebssystemumgebung ist der Benutzer verantwortlich.
- Fragen Sie während der Installation der Anwendung die verwendete Windows-Version ab. Wird eine nicht kompatible Version festgestellt, brechen Sie die Installation ab oder weisen den Benutzer auf das Problem hin.
- Fragen Sie zur Laufzeit der Anwendung die verwendete Windows-Version ab. Dieser Test muß unmittelbar vor der Ausführung einer Operation stattfinden, die nicht in allen Windows-Versionen durchgeführt werden kann. Wird eine nicht kompatible Version festgestellt, brechen Sie den Vorgang ab und weisen den Benutzer auf das Problem hin. Sie können aber auch Quelltext schreiben, der in Abhängigkeit von der jeweils verwendeten Windows-Version ausgeführt wird. In Windows 95 werden einige Operationen anders ausgeführt als in Windows NT. Mit der Windows-API-Funktion *GetVersionEx* kann die verwendete Windows-Version ermittelt werden.

Software-Lizenzvereinbarungen

Für die Weitergabe einiger Dateien, die zu Delphi-Anwendungen gehören, gelten bestimmte Einschränkungen. Es gibt auch Dateien, die nicht weitergegeben werden dürfen. Die rechtlichen Bestimmungen für die Weitergabe dieser Dateien finden Sie in folgenden Dokumenten:

- DEPLOY.TXT
- README.TXT
- Lizenzvereinbarungen
- Dokumentation zu Produkten von Fremdherstellern

DEPLOY.TXT

In der Datei DEPLOY.TXT sind die rechtlichen Bestimmungen hinsichtlich der Weitergabe verschiedener Komponenten, Dienstprogramme und anderer Produkte beschrieben, die Bestandteil einer Delphi-Anwendung sein können. Die Textdatei DEPLOY.TXT wird bei der Installation in das Hauptverzeichnis von Delphi kopiert. In dieser Datei werden folgende Themen behandelt:

- EXE-, DLL- und BPL-Dateien
- Komponenten und Entwurfszeit-Packages
- Borland Database Engine (BDE)
- ActiveX-Steuerelemente
- Beispielgrafiken
- MIDAS (Multi-tiered Distributed Application Services)
- SQL Links

README.TXT

Die Datei README.TXT enthält neueste Informationen über Delphi, die auch die Weitergaberechte für Komponenten, Dienstprogramme und andere Produktbereiche betreffen können. README.TXT ist eine Windows-Hilfedatei, die bei der Installation in das Hauptverzeichnis von Delphi kopiert wird.

Lizenzvereinbarungen

Die Lizenzvereinbarungen für Delphi, die in gedruckter Form vorliegen, gehen auf weitere rechtliche Belange ein, die Delphi betreffen.

Dokumentation zu Produkten von Fremdherstellern

Die Weitergaberechte für Komponenten, Dienstprogramme, Hilfsanwendungen, Datenbank-Engines und sonstige Produkte von Fremdherstellern sind in den Lizenzvereinbarungen des betreffenden Anbieters festgelegt. Lesen Sie vor der Weitergabe derartiger Produkte zusammen mit Delphi-Anwendungen in der Dokumentation des Produkts nach, oder wenden Sie sich an den Software-Händler, bei dem Sie das Produkt erworben haben.

Teil

II

Datenbankanwendungen entwickeln

Die Kapitel in diesem Teil des Handbuchs beschreiben die grundlegenden Konzepte der Entwicklung von Datenbankanwendungen mit Delphi.

Hinweis Für Datenbankanwendungen benötigen Sie eine der Delphi-Editionen Professional, Client/Server oder Enterprise. Um fortgeschrittene Client/Server-Datenbanken zu implementieren, benötigen Sie Delphi in der Edition Client/Server oder Enterprise.

Kapitel

12

Datenbankanwendungen entwerfen

Über Datenbankanwendungen können Benutzer auf die in einer Datenbank gespeicherten Informationen zugreifen. Datenbanken strukturieren Informationen und ermöglichen unterschiedlichen Anwendungen, darauf zuzugreifen.

Delphi unterstützt Anwendungen für relationale Datenbanken. In relationalen Datenbanken sind Informationen in Tabellen organisiert, die Zeilen (Datensätze) und Spalten (Felder) enthalten. Diese Tabellen können durch einfache Operationen bearbeitet werden.

Für den Entwurf einer Datenbankanwendung ist das Verständnis der Datenstruktur von großer Bedeutung, da die Benutzeroberfläche auf Grundlage dieser Struktur entwickelt wird. Die Oberfläche dient dem Benutzer zur Anzeige, Eingabe und Änderung von Daten.

Dieses Kapitel enthält allgemeine Überlegungen zum Entwurf einer Datenbankanwendung und beschäftigt sich detailliert mit der Entwicklung der Benutzeroberfläche.

Datenbanken

Die Komponenten der Registerkarten *Datenzugriff*, *ADO* bzw. *InterBase* der Komponentenpalette ermöglichen Anwendungen den Schreib- und Lesezugriff auf Datenbanken. Die Komponenten der Registerkarte *Datenzugriff* greifen über die Borland Database Engine (BDE) auf Datenbankinformationen zu und stellen sie den datensensitiven Steuerelementen der Benutzeroberfläche zur Verfügung. Die Komponenten der Registerkarte *ADO* verwenden ActiveX-Datenobjekte (Active Data Objects = ADO), um über OLEDB auf Datenbankinformationen zuzugreifen. Mit den Komponenten auf der Registerkarte *InterBase* kann direkt auf eine InterBase-Datenbank zugegriffen werden.

Die BDE enthält je nach verwendeter Delphi-Version Treiber für unterschiedliche Datenbanktypen, die jedoch alle ihre Daten in Tabellenform speichern. Einige Datenbanktypen unterstützen darüber hinaus folgende Funktionen:

- Datenbanksicherheit.
- Transaktionen.
- Das Data Dictionary.
- Referentielle Integrität, Stored Procedures und Trigger.

Datenbanktypen

Sie können mit verschiedensten Datenbanktypen arbeiten, wenn entsprechende Treiber für BDE oder ADO installiert wurden.

Diese Treiber können Ihre Anwendungen mit Lokalen Datenbanken wie Paradox, Access und dBASE oder mit Remote-Datenbank-Servern wie Microsoft SQL Server, Oracle und Informix verbinden. Ähnlich die InterBase-Express-Komponenten; sie können auf eine lokale oder auf eine entfernte InterBase-Version zugreifen.

Hinweis Verschiedene Versionen von Delphi werden mit Komponenten ausgeliefert, die diese Treiber verwenden (BDE oder ADO), oder mit den Komponenten von InterBase Express.

Die Auswahl des passenden Datenbanktyps hängt neben anderen Faktoren auch davon ab, ob die Daten bereits in einer bestehenden Datenbank gespeichert sind. Vor Erstellung der Tabellen für die Anwendung sollten Sie folgende Fragen klären:

- Wie umfangreich sind die in den Tabellen zu speichernden Daten?
- Wie viele Benutzer greifen gemeinsam auf die Tabellen zu?
- Welche Anforderungen werden an die Leistung (Geschwindigkeit) der Datenbank gestellt?

Lokale Datenbanken

Lokale Datenbanken befinden sich auf einem lokalen Laufwerk oder in einem lokalen Netzwerk. Der Zugriff auf die Daten erfolgt über eigene APIs. Lokale Datenbanken sind oft auf einzelnen Systemen eingerichtet und steuern den gemeinsamen Zugriff mehrerer Benutzer über dateibasierte Sperrmechanismen. Sie werden deshalb auch als dateibasierte Datenbanken bezeichnet.

Da sich lokale Datenbanken häufig auf demselben System wie die Datenbankanwendung befinden, ist die Zugriffsgeschwindigkeit bei ihnen in der Regel höher als bei Remote-Datenbankservern.

Lokale Datenbanken haben aufgrund ihrer Architektur im Vergleich zu Remote-Datenbankservern eine geringere Speicherkapazität. Bei der Entscheidung für oder gegen eine lokale Datenbank sollte daher auch der erwartete Datenumfang eine Rolle spielen.

Anwendungen für lokale Datenbanken bezeichnet man auch als einschichtige Anwendungen, weil sich die Anwendung und die Datenbank ein Dateisystem teilen.

Paradox, dBASE, FoxPro und Access sind Beispiele für lokale Datenbanken.

Remote-Datenbankserver

Remote-Datenbankserver befinden sich normalerweise auf einem entfernten Computer. Der Datenzugriff der Client-Computer wird bei solchen Servern über SQL (Structured Query Language = strukturierte Abfragesprache) realisiert. Sie werden deshalb auch als SQL-Server oder RDBMS (Remote Database Management System) bezeichnet. Die meisten Remote-Datenbankserver unterstützen zusätzlich zu den gemeinsamen SQL-Basisbefehlen eigene SQL-Dialekte.

Remote-Datenbankserver ermöglichen mehreren Benutzern den gleichzeitigen Zugriff auf eine Datenbank. Im Gegensatz zum dateibasierten Sperrsystem lokaler Datenbanken bieten sie eine hochentwickelte Mehrbenutzerunterstützung auf der Grundlage von Transaktionen.

Remote-Datenbankserver enthalten mehr Daten als lokale Datenbanken. Diese Daten befinden sich entweder auf einem einzelnen Remote-Computer oder auf mehreren Servern.

Anwendungen für Remote-Datenbankserver werden als zweischichtige Anwendungen oder auch als mehrschichtige Anwendungen bezeichnet, weil die Anwendung und die Datenbank auf unabhängigen Systemen (oder Schichten) operieren.

InterBase, Oracle, Sybase, Informix, Microsoft SQL Server und DB2 sind Beispiele für SQL-Server.

Datenbanksicherheit

Zum Schutz der gespeicherten Daten stellen die unterschiedlichen Datenbanktypen entsprechende Sicherheitsmechanismen zur Verfügung. In einigen Datenbanken wie Paradox oder dBASE wird der Datenschutz auf Tabellen- und Feldebene realisiert. Bei diesen Modellen benötigen Benutzer ein gültiges Kennwort, um auf geschützte Tabellen zuzugreifen, und nur die für sie freigegebenen Felder (Spalten) werden angezeigt.

Für den Zugriff auf die meisten SQL-Server ist ein Kennwort und ein Benutzername erforderlich. Durch diese Angaben werden die Zugriffsberechtigungen eines Benutzers in einer Datenbank festgelegt. Informationen zur Übergabe von Kennwörtern an SQL-Server mit der BDE finden Sie unter »Server-Login steuern« auf Seite 17-6. Das Übertragen dieser Informationen beim Einsatz von ActiveX-Datenobjekten (ADO) wird unter »Die Anmeldung für eine Verbindung steuern« auf Seite 23-8 beschrieben. Die Übergabe der Informationen mit den InterBase-Direktzugriffskomponenten erläutert der Abschnitt zum Ereignis *OnLogin* der Klasse *TIBDatabase*.

Beachten Sie, daß nicht alle Delphi-Editionen die Direktzugriffskomponenten für ADO und InterBase enthalten.

Bei der Entwicklung von Datenbankanwendungen muß die Art der Bestätigung für den jeweiligen Datenbankserver berücksichtigt werden. Benutzerzugriffe ohne Kennwort sind nur bei Datenbanken möglich, die kein Kennwort erfordern. Außerdem besteht die Möglichkeit, das Kennwort und den Benutzernamen programmseitig an den Server zu übergeben. In diesem Fall muß aber die Sicherheitslücke beachtet werden, die durch Lesen des Kennworts in der Anwendung entstehen könnte.

Sollen Benutzer nur über ein Kennwort Zugriff erhalten, muß der Eingabezeitpunkt dieses Kennworts festgelegt werden. Für lokale Datenbanken, die später zu einem größeren SQL-Server erweitert werden sollen, sollte bereits ein Kennwort vergeben werden, obwohl es bei diesem Datenbanktyp noch nicht erforderlich ist.

Aufgrund der Anmeldung bei mehreren geschützten Systemen oder Datenbanken kann die Eingabe mehrfacher Kennwörter in einer Anwendung erforderlich sein. In solchen Fällen kann dem Benutzer durch die Vergabe eines Hauptkennwortes die mehrfache Eingabe abgenommen werden. Die Eingabe dieses Hauptkennwortes ruft eine Tabelle mit den angeforderten Kennwörtern ab, die dann von der Anwendung automatisch weitergeleitet werden.

Mehrschichtige Anwendungen erfordern ein völlig anderes Sicherheitsmodell. Mit HTTPs, CORBA oder MTS können Sie die Zugriffe auf die mittleren Schichten steuern, die ihrerseits alle Details der Anmeldung bei den Datenbankservern regeln.

Transaktionen

Eine Transaktion besteht aus einer Gruppe von Aktionen, die in einer oder mehreren Datenbanktabellen erfolgreich ausgeführt werden müssen, bevor sie endgültig gespeichert werden. Wenn eine der Aktionen fehlschlägt, werden auch alle anderen Aktionen komplett rückgängig gemacht.

Transaktionen bieten während der Ausführung eines oder mehrerer Datenbankbefehle Schutz vor Hardwarefehlern. Sie bilden außerdem die Grundlage für die gleichzeitige Steuerung mehrfacher Benutzerzugriffe auf den SQL-Server. Wenn jeder Benutzer mit der Datenbank ausschließlich über Transaktionen interagiert, können einzelne Transaktionseinheiten nicht durch fremde Benutzerbefehle unterbrochen werden. Statt dessen verwaltet der SQL-Server die eingehenden Transaktionen immer als Ganzes, unabhängig davon, ob sie erfolgreich sind oder fehlschlagen.

Lokale Datenbanken unterstützen meist keine Transaktionen. Die BDE-Treiber unterstützen diese Funktion jedoch in begrenztem Rahmen für einige Datenbanken. Bei SQL-Servern und ODBC-kompatiblen Datenbanken erfolgt die Unterstützung durch die Komponente, die die Datenbankverbindung repräsentiert. Mehrschichtige Anwendungen ermöglichen sogar Transaktionen, die neben Datenbankoperationen weitere Aktionen enthalten oder sich auf mehrere Datenbanken erstrecken.

Details über Transaktionen in BDE-basierten Datenbankanwendungen finden Sie unter »Transaktionen« auf Seite 13-5. Details über Transaktionen in ADO-Datenbankanwendungen finden Sie unter »Mit Verbindungstransaktionen arbeiten« auf Seite 23-12. Details über Transaktionen in mehrschichtigen Anwendungen finden Sie unter »Transaktionen in mehrschichtigen Anwendungen verwalten« auf Seite 14-28.

Die Implementierung von Transaktionen in Anwendungen mit InterBase-Direktzugriffskomponenten wird in der Online-Hilfe unter *TIBTransaction* beschrieben.

Das Daten-Dictionary

Das Daten-Dictionary steht allen Anwendungen zur Verfügung, die über die BDE auf die Daten zugreifen. Es bietet einen benutzerdefinierbaren Speicherbereich zur Erstellung erweiterter Feldattributmengen, die den Inhalt und das Erscheinungsbild von Daten beschreiben.

Wenn Sie beispielsweise häufig Finanzanwendungen entwickeln, bietet sich die Erstellung von speziellen Feldattributmengen zur Beschreibung unterschiedlicher Währungsformate an. Bei der Erstellung von Datenmengen zur Entwurfszeit können die Währungsfelder mit einer erweiterten Feldattributmenge im Daten-Dictionary verbunden werden und brauchen nicht mit dem Objektinspektor jeder Datenmenge einzeln zugewiesen zu werden. Das Daten-Dictionary sorgt so für ein einheitliches Erscheinungsbild der Daten innerhalb der Anwendungen.

In einer Client/Server-Umgebung kann sich das Daten-Dictionary zur gemeinsamen Nutzung auf einem Remote-Server befinden.

Wie Sie zur Entwurfszeit mit dem Feldeditor erweiterte Feldattributmengen erstellen und sie mit Feldern verbinden, erfahren Sie unter »Attributsätze für Feldkomponenten erstellen« auf Seite 19-16». Weitere Informationen über die Erstellung von Daten-Dictionaries und erweiterten Feldattributen mit SQL- und Datenbank-Explorer finden Sie in der Online-Hilfe.

Eine Programmierschnittstelle zum Daten-Dictionary ist in der Unit *Drintf* im Verzeichnis LIB verfügbar. Diese Schnittstelle stellt folgende Methoden zur Verfügung:

Tabelle 12.1 Schnittstelle zum Daten-Dictionary

Routine	Verwendung
DictionaryActive	Zeigt an, ob das Daten-Dictionary aktiviert ist.
DictionaryDeactivate	Deaktiviert das Daten-Dictionary.
IsNullID	Zeigt an, ob eine bestimmte ID eine Null-ID ist.
FindDatabaseID	Liefert die ID für den Alias einer Datenbank.
FindTableID	Liefert die ID für eine Tabelle in einer bestimmten Datenbank.
FindFieldID	Liefert die ID für ein Feld in einer bestimmten Tabelle.
FindAttrID	Liefert die ID für den Namen einer Attributmenge.
GetAttrName	Liefert den Namen für die ID einer Attributmenge.
GetAttrNames	Führt für jede Attributmenge im Dictionary einen Callback durch.
GetAttrID	Liefert die ID einer Attributmenge für ein bestimmtes Feld.
NewAttr	Erzeugt eine neue Attributmenge für eine Feldkomponente.
UpdateAttr	Aktualisiert eine Attributmenge zur Anpassung der Feldeigenschaften.
CreateField	Erzeugt eine Feldkomponente auf Grundlage von gespeicherten Attributen.
UpdateField	Ändert die Eigenschaften eines Feldes zur Anpassung an eine bestimmte Attributmenge.

Tabelle 12.1 Schnittstelle zum Daten-Dictionary (Fortsetzung)

Routine	Verwendung
AssociateAttr	Verbindet eine Attributmenge mit einer bestimmten Feld-ID.
UnassociateAttr	Beendet die Verbindung zwischen einer Attributmenge und einer Feld-ID.
GetControlClass	Liefert die Steuerelementklasse für eine bestimmte Attribut-ID.
QualifyTableName	Liefert einen (durch den Benutzernamen) vollständig qualifizierten Tabellennamen.
QualifyTableNameBy Name	Liefert einen (durch den Benutzernamen) vollständig qualifizierten Tabellennamen.
HasConstraints	Zeigt an, ob es für die Datenmenge im Dictionary Beschränkungen gibt.
UpdateConstraints	Aktualisiert die importierten Beschränkungen einer Datenmenge.
UpdateDataset	Aktualisiert eine Datenmenge anhand der aktuellen Einstellungen und Beschränkungen im Dictionary.

Referentielle Integrität, Stored Procedures und Trigger

Alle relationalen Datenbanken verfügen im allgemeinen über bestimmte Funktionen, die Anwendungen das Speichern und Bearbeiten von Daten ermöglichen. Zusätzlich bieten Datenbanken oft spezifische Funktionen zur Sicherstellung konsistenter Beziehungen zwischen den einzelnen Tabellen einer Datenbank. Die folgenden Abschnitte beschreiben einige dieser Funktionen:

- **Referentielle Integrität:** Durch referentielle Integrität wird die Auflösung von Haupt/Detailbeziehungen zwischen Tabellen verhindert. Ein Benutzer kann kein Feld in der Haupttabelle löschen, wenn dieser Vorgang verwaiste Datensätze in der Detailtabelle zurückläßt. Durch die Regeln der referentiellen Integrität wird in diesem Fall entweder das Löschen verhindert, oder die verwaisten Detaildatensätze werden automatisch gelöscht.
- **Stored Procedures:** Stored Procedures sind Gruppen von SQL-Anweisungen, die unter einem bestimmten Namen auf einem SQL-Server gespeichert sind. Sie führen normalerweise gemeinsame datenbankbezogene Operationen auf dem Server aus und liefern Gruppen von Datensätzen (Datenmengen).
- **Trigger:** Trigger sind Gruppen von SQL-Anweisungen, die als Reaktion auf einen speziellen Befehl automatisch ausgeführt werden.

Datenbankarchitektur

Datenbankanwendungen setzen sich aus Elementen der Benutzeroberfläche, aus Komponenten zur Verwaltung einer oder mehrerer Datenbanken und aus Komponenten zur Darstellung der Tabellendaten (Datenmengen) zusammen. Die Gesamtheit dieser Teile bildet die Architektur einer Datenbankanwendung.

Die Isolierung von Datenzugriffskomponenten in Datenmodulen ermöglicht die Entwicklung einer einheitliche Benutzeroberfläche mit Hilfe von Formularen. Die in der Objektablage gespeicherten Verknüpfungen zu bestehenden Formularen und Daten-

modulen dienen Entwicklern als Grundlage für neue Projekte. Darüber hinaus ermöglichen Formulare und Module die Entwicklung gemeinsamer Standards für Datenbankzugriffe und Anwendungsschnittstellen.

Wesentliche Aspekte der Architektur einer Datenbankanwendung hängen vom verwendeten Datenbanktyp, der voraussichtlichen Anzahl der Benutzer und der Art der verarbeiteten Daten ab. Weitere Informationen über die verschiedenen Datenbanktypen finden Sie unter »Datenbanktypen« auf Seite 12-2.

Für Anwendungen, in denen nicht mehrere Benutzer gleichzeitig auf die Daten zugreifen, sollte eine lokale Datenbank in einer *einschichtigen Anwendung* eingesetzt werden. Die lokale Speicherung der Daten ergibt eine höhere Zugriffsgeschwindigkeit und erspart Ihnen die Kosten eines eigenen Datenbankservers und teurer Standortlizenzen. Die Speicherkapazität der Tabellen und die Anzahl der unterstützten Benutzer ist bei lokalen Datenbanken allerdings begrenzt.

Im Vergleich dazu unterstützen *zweischichtige Anwendungen* gleichzeitige Zugriffe mehrerer Benutzer und ermöglichen die Verwendung von Remote-Datenbanken mit erheblich größerer Speicherkapazität.

Hinweis Zweischichtige Anwendungen benötigen SQL Links, InterBase oder ADO (ActiveX Data Objects).

Bei komplizierten Beziehungen zwischen mehreren Tabellen einer Datenbank oder bei einer zunehmenden Anzahl von Clients bietet sich die Verwendung einer *mehrschichtigen Anwendung* an. Bei mehrschichtigen Anwendungen wird die Logik der Datenbankinteraktionen in den mittleren Schichten zentralisiert. Durch diese zentralisierte Steuerung der Datenbeziehungen können unterschiedliche Client-Anwendungen auf dieselben Daten zugreifen, ohne die Konsistenz der Daten zu gefährden. Außerdem können kleinere Client-Anwendungen eingesetzt werden, da ein Teil der Berechnungen in die mittleren Schichten ausgelagert ist. Kleinere Client-Anwendungen sind einfacher zu installieren, zu konfigurieren und zu verwalten, da sie keine Verbindungssoftware zur Datenbank enthalten. Außerdem verbessern mehrschichtige Anwendungen durch die Verteilung von Berechnungsoperationen auf mehrere Systeme die Systemleistung.

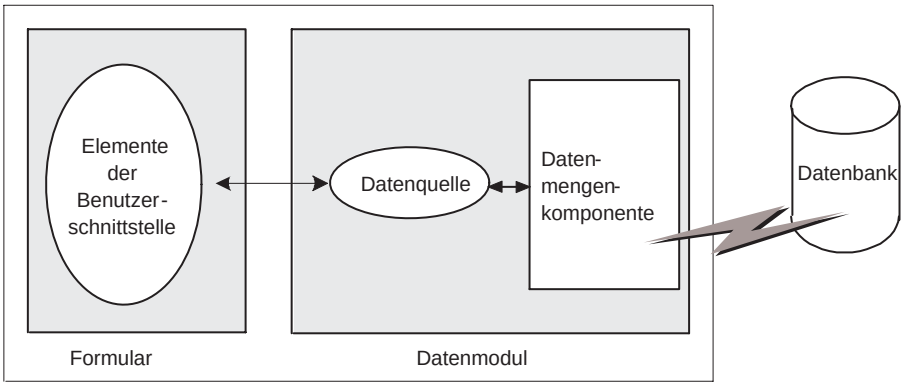
Skalierbarkeit planen

Je vielschichtiger eine Anwendung ist, um so komplexer und teurer kann der Entwicklungsprozeß werden. Deshalb werden einige Anwendungen zunächst auf Grundlage einer einschichtigen Architektur entwickelt. Durch die Zunahme des Datenumfangs, der Benutzeranzahl oder der Anzahl verschiedener Anwendungen, die auf die Daten zugreifen, kann sich allerdings später die Notwendigkeit der Erweiterung auf eine mehrschichtige Architektur ergeben. Durch diese Planung auf Skalierbarkeit können ein- oder zweischichtige Anwendungen bereits so programmiert werden, daß ihr Quelltext bei zukünftigen Erweiterungen wiederverwendet werden kann. Die rechtzeitige Planung der Skalierbarkeit trägt also zu einer Senkung der Entwicklungskosten bei.

Die datensensitiven VCL-Komponenten vereinfachen die Programmierung skalierbarer Anwendungen, indem sie das Verhalten der Datenbank und der in ihr gespei-

cherten Daten abstrahieren. Bei allen Anwendungsarten (ein-, zwei- und mehrschichtig) kann die Benutzeroberfläche von der Datenzugriffsschicht isoliert werden. Abbildung 12.1. verdeutlicht diesen Prozeß.

Abbildung 12.1 Verbindung zwischen Benutzeroberfläche und Datenmengen in allen Datenbankanwendungen



Das Formular stellt die Benutzerschnittstelle dar. Es enthält die Datensteuerungskomponenten und die anderen Elemente der Benutzeroberfläche. Die Datensteuerungskomponenten sind über eine Datenquelle mit Datenmengen aus Tabellen verbunden. Durch die Isolierung von Datenquelle und Datenmenge in einem Datenmodul müssen bei einer Anwendungserweiterung nur die Datenmengen, nicht aber das Formular geändert werden.

Hinweis Einige Elemente der Benutzeroberfläche erfordern bei der Planung der Skalierbarkeit besondere Aufmerksamkeit. So wird beispielsweise die Sicherheit in Datenbanken auf unterschiedliche Weise realisiert. Informationen über einheitliche Authentifizierungsverfahren bei Datenbankänderungen finden Sie unter »Datenbanksicherheit« auf Seite 12-3.

Wenn Sie mit den Datenzugriffskomponenten von Delphi arbeiten (die entweder die BDE, ADO oder InterBase Express verwenden), ist es einfach, einschichtige in zweischichtige Anwendungen zu überführen. Dazu müssen lediglich einige Eigenschaften der Datenmenge geändert werden, damit die Datenmenge mit einem SQL-Server statt mit einer lokalen Datenbank verknüpft werden kann.

Eine Datenbankanwendung, die unstrukturierte Dateien verwendet, kann einfach zum Client einer mehrschichtigen Anwendung skaliert werden, weil beide Architekturen dieselben Client-Datenmengenkomponenten verwenden. Eine Anwendung kann durch entsprechende Programmierung sogar auf beide Arten eingesetzt werden. Beachten Sie dazu den Abschnitt »Das Aktenkoffer-Modell« auf Seite 13-19.

Die mögliche Erweiterung auf eine dreischichtige Architektur sollte bereits bei der Entwicklung einer ein- oder zweischichtigen Anwendung berücksichtigt werden, indem die Benutzeroberfläche von den anderen Programmteilen getrennt wird. Auch die Logik, die später auf die mittlere Schicht verlagert wird, sollte bereits jetzt isoliert werden. Außerdem können Elemente der Benutzeroberfläche mit Client-Datenmen-

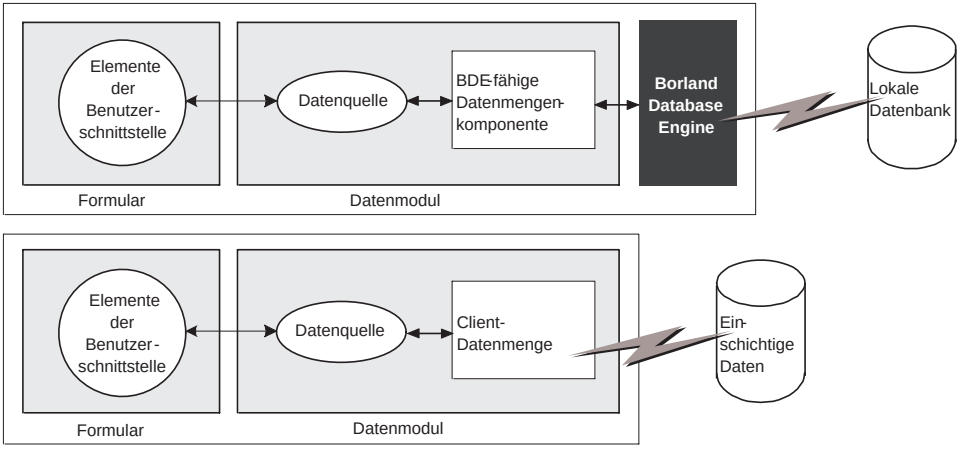
gen in mehrschichtigen Anwendungen verbunden werden, die wiederum mit lokalen Versionen der InterBase-, BDE- oder ADO-Datenmengen in einem eigenen, eventuell später in die mittlere Schicht verlagerten Datenmodul verknüpft werden. Auch ohne diesen Trick mit einer zusätzlichen Datenmengenschicht wirft die spätere Erweiterung ein- und zweischichtiger auf dreischichtige Anwendungen kaum Probleme auf. Weitere Informationen finden Sie unter »Skalierung auf eine dreischichtige Anwendung« auf Seite 13-20.

Einschichtige Datenbankanwendungen

Bei einer einschichtigen Datenbankanwendung teilen sich die Anwendung und die Datenbank ein gemeinsames Dateisystem. Dabei werden lokale Datenbanken oder unstrukturierte Dateien verwendet.

Eine Anwendung umfaßt die Benutzeroberfläche und den Datenzugriffsmechanismus (entweder die BDE oder ein System zum Laden und Speichern von unstrukturierten Daten). In Abhängigkeit vom Speicherort der Daten (lokale Datenbank oder unstrukturierte Datei) wird eine entsprechende Datenmengenkomponente zur Darstellung der Datenbanktabellen verwendet. Die folgende Abbildung zeigt diese beiden Möglichkeiten.

Abbildung 12.2 Architektur einschichtiger Datenbankanwendungen

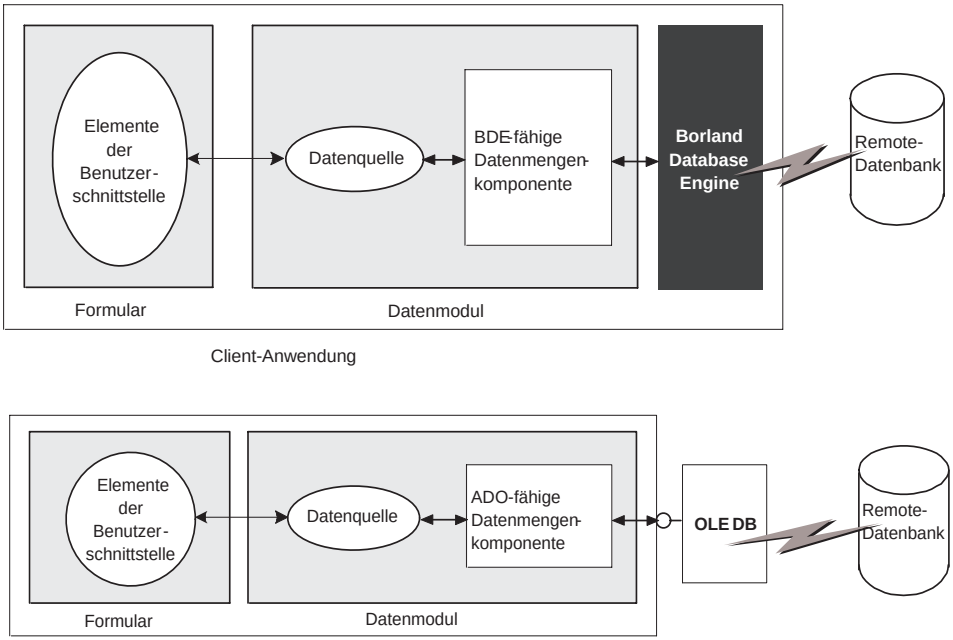


Informationen über einschichtige Datenbankanwendungen finden Sie in Kapitel 13, »Ein- und zweischichtige Anwendungen erstellen«.

Zweischichtige Datenbankanwendungen

Bei zweischichtigen Datenbankanwendungen stellt eine Client-Anwendung die Benutzeroberfläche für den Datenzugriff zur Verfügung und interagiert direkt mit einem Remote-Datenbankserver. Abbildung 12.3 zeigt diese Beziehung.

Abbildung 12.3 Architekturen zweischichtiger Datenbankanwendungen



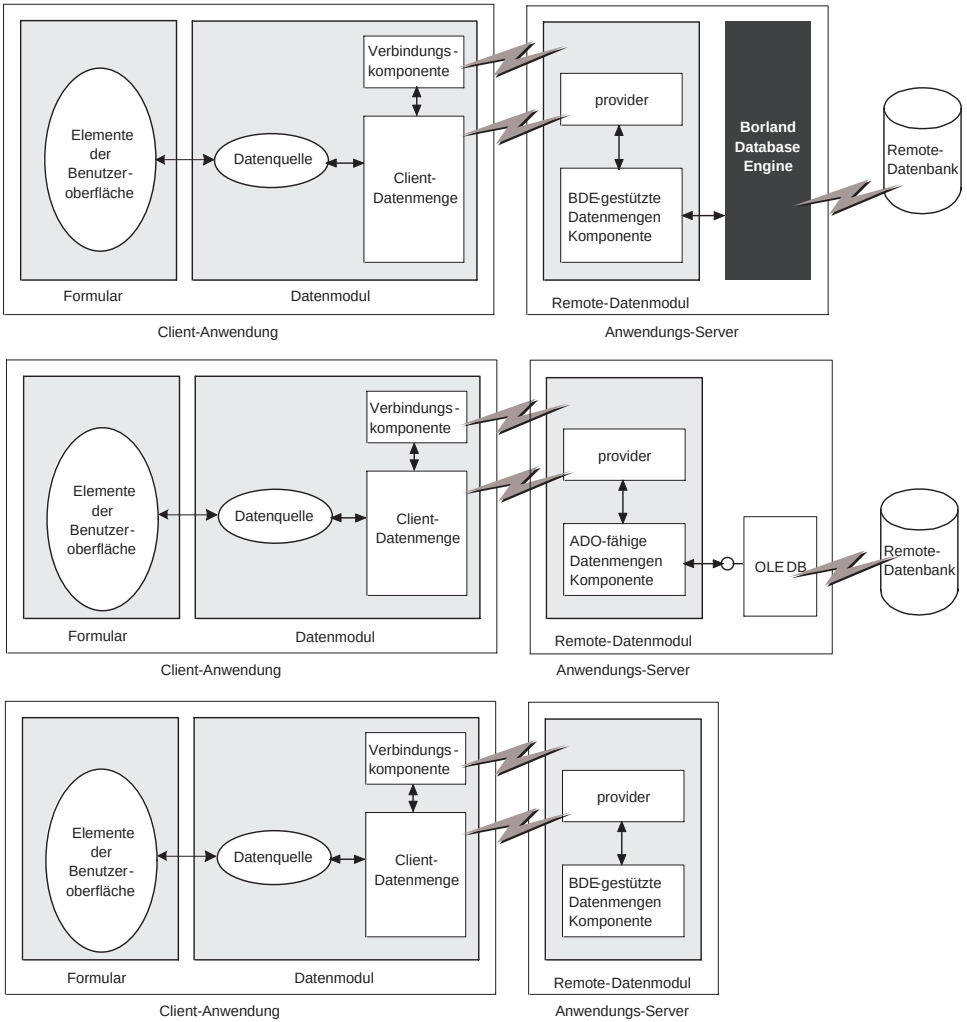
In diesem Modell fungieren alle Anwendungen als Datenbank-Clients. Ein Client fordert Daten von einem Datenbankserver an und sendet Daten an ihn. Der Server bearbeitet gleichzeitig Anfragen mehrerer Clients, koordiniert den Zugriff und aktualisiert die Daten.

Mehr Informationen über die Erstellung zweischichtiger Datenbankanwendungen finden Sie unter »BDE-Anwendungen« auf Seite 13-2 sowie »ADO-basierte Anwendungen« auf Seite 13-11.

Mehrschichtige Datenbankanwendungen

Mehrschichtige Datenbankanwendungen sind in Komponenten aufgeteilt, die sich auf verschiedenen Computern befinden. Eine Client-Anwendung stellt die Benutzeroberfläche für den Datenzugriff zur Verfügung und übergibt alle Datenanfragen und Aktualisierungen an einen Anwendungsserver, der auch als Remote-Datenbroker bezeichnet wird. Der Anwendungsserver kommuniziert seinerseits direkt mit einem Remote-Datenbankserver oder einer anderen benutzerdefinierten Datenmenge. Bei diesem Modell befinden sich die Client-Anwendung, der Anwendungsserver und der Remote-Datenbankserver auf getrennten Computern. Die folgende Abbildung zeigt diese Beziehungen für verschiedene Typen von mehrschichtigen Anwendungen.

Abbildung 12.4 Architekturen mehrschichtiger Datenbanken



Delphi ermöglicht die Erstellung von Client-Anwendungen und Anwendungsservern. Die Client-Anwendung verwendet zur Anzeige und Bearbeitung der Daten datensensitive Standardsteuerelemente, die über eine Datenquelle mit einer oder mehreren Client-Datenmengen verbunden sind. Die Kommunikation zwischen einer Client-Datenmenge und einem Anwendungsserver erfolgt über eine *IAppServer*-Schnittstelle, die vom Remote-Datenmodul des Anwendungsservers implementiert wird. Zur Herstellung dieser Verbindung kann die Client-Anwendung eine Vielzahl von Protokollen (TCP/IP, HTTP, DCOM, MTS oder CORBA) verwenden. Das geeignete Protokoll hängt vom Typ der in der Client-Anwendung verwendeten Verbindungskomponente und vom Typ des in der Server-Anwendung verwendeten Remote-Datenmoduls ab.

Der Anwendungsserver enthält Provider-Komponenten zur Vermittlung der Kommunikation zwischen den Client-Datenmengen der Client-Anwendung und den Datenmengen des Anwendungsservers. Die Übergabe der Daten zwischen Client-Anwendungen und Provider-Komponenten erfolgt über die *IAppServer*-Schnittstelle.

Im mehrschichtigen Modell kommunizieren üblicherweise mehrere Client-Anwendungen mit einem Anwendungsserver, der als Gateway zu den Datenbanken fungiert. Durch diesen Anwendungsserver können außerdem unternehmensweite Datenbankoperationen allen Clients an einem zentralen Standort zur Verfügung gestellt werden. Weitere Informationen über die Erstellung und die Verwendung mehrschichtiger Datenbankanwendungen finden Sie in Kapitel 14, »Mehrschichtige Anwendungen erstellen«.

Benutzeroberflächen entwickeln

Die Registerkarte *Datensteuerung* der Komponentenpalette stellt eine Reihe datensensitiver Steuerelemente zur Verfügung. Diese Steuerelemente dienen zur Darstellung von Daten aus Datensatzfeldern und erlauben Benutzern die Bearbeitung dieser Daten und die Speicherung der Änderungen in der Datenbank. Datensensitive Steuerelemente realisieren den Zugriff über die Benutzeroberfläche einer Datenbankanwendung. Weitere Information über datensensitive Steuerelemente finden Sie in Kapitel 26, »Datensensitive Steuerelemente«.

Datensensitive Steuerelemente erhalten Daten von und senden Daten an eine Datenquellenkomponente (*TDataSource* oder *TIBDataSource*). Eine Datenquellenkomponente dient als Bindeglied zwischen der Benutzeroberfläche und einer Datenmengenkomponente. Mehrere datensensitive Steuerelemente in einem Formular können sich eine Datenquelle teilen, was eine synchrone Anzeige in jedem Steuerelement zur Folge hat. Wenn der Benutzer die Datensätze durchläuft, werden in jedem Steuerelement die Feldwerte des jeweils aktuellen Datensatzes angezeigt. Die Datenquellenkomponenten einer Anwendung befinden sich üblicherweise in einem von den datensensitiven Steuerelementen im Formular getrennten Datenmodul.

Die Auswahl der datensensitiven Steuerelemente für eine Benutzeroberfläche hängt vom jeweiligen Datentyp (unformatierter Text, formatierter Text, Grafiken, Multimediaelemente usw.) und von der Art ab, wie diese Informationen organisiert sind. Eine zusätzliche Rolle bei der Wahl der Steuerelemente spielt die Organisation des Benutzerzugriffs (Berechtigungen und Navigation).

In den folgenden Abschnitten werden die Komponenten für verschiedene Typen von Benutzeroberflächen beschrieben.

Einzelne Datensätze anzeigen

In vielen Anwendungen wird immer nur ein Datensatz zur selben Zeit angezeigt. Eine Anwendung, in der Aufträge erfaßt werden, soll beispielsweise nur die Daten eines einzelnen Auftrags anzeigen. Die benötigten Informationen befinden sich höchstwahrscheinlich in einem einzigen Datensatz einer Auftragsdatenmenge.

Anwendungen, die immer nur einen einzelnen Datensatz anzeigen, sind normalerweise nicht allzu kompliziert aufgebaut, da sich alle Datenbankinformationen immer auf denselben Gegenstand (z. B. denselben Auftrag) beziehen. Die datensensitiven Steuerelemente der entsprechenden Benutzeroberfläche stellen jeweils ein einzelnes Feld aus einem Datensatz dar. Die Registerkarte *Datensteuerung* der Komponentenpalette enthält eine umfangreiche Auswahl an Steuerelementen zur Darstellung unterschiedlicher Feldarten. Weitere Informationen über spezielle datensensitive Steuerelemente finden Sie unter »Steuerelemente zur Darstellung eines einzelnen Feldes« auf Seite 26-10 .

Mehrere Datensätze anzeigen

Bestimmte Formulare dienen der Anzeige mehrerer Datensätze, beispielsweise bei einer Anwendung, in der alle Aufträge eines bestimmten Kunden in Rechnung gestellt werden.

Für die Anzeige mehrerer Datensätze werden Datengitter verwendet. In Datengittern können mehrere Felder und Datensätze gleichzeitig angezeigt werden. Diese Steuerelemente werden in den Abschnitten »Daten mit TDBGrid anzeigen und bearbeiten« auf Seite 26-20 und »Gitter mit anderen datensensitiven Steuerelementen erstellen« auf Seite 26-33.

Benutzeroberflächen können sowohl Felder eines einzelnen Datensatzes als auch Gitter mit mehreren Datensätzen enthalten. Es gibt zwei Möglichkeiten, eine solche Benutzeroberfläche zu realisieren:

- **Haupt/Detail-Formulare:** Durch die gemeinsame Verwendung von Gittern und Steuerelementen zur Anzeige einzelner Datensätze in einem Formular können Daten aus einer Haupt- und einer Detailtabelle abgerufen werden. So können beispielsweise für einen bestimmten Kunden in einem Detailgitter die unterschiedlichen Aufträge angezeigt werden. Informationen zum Verknüpfen der einem Haupt/Detail-Formular zugrundeliegenden Tabellen finden Sie im Abschnitt »Haupt/Detail-Formulare erstellen« auf Seite 20-28 oder »Verschachtelte Tabellen« auf Seite 20-29.
- **Gemischte Formulare:** In einem Formular, in dem mehrere Datensätze angezeigt werden, zeigen Steuerelemente für einzelne Felder Informationen aus dem jeweils aktuellen Datensatz in detaillierter Form an. Dieser Ansatz ist besonders dann sinnvoll, wenn die Datensätze lange Memos oder Grafiken enthalten. Wenn der Benutzer im Gitter durch die Datensätze blättert, wird für den jeweils aktuellen Datensatz das zugehörige Memo oder die Grafik angezeigt. Die Einrichtung dieses Modells ist einfach. Wenn alle beteiligten Steuerelemente auf dieselbe Datenquelle zugreifen, wird die Anzeige automatisch synchronisiert.

Tip Die beiden Modelle sollten möglichst nicht gemeinsam in einem Formular eingesetzt werden. Die vielen Beziehungen zwischen den Daten sind für den Benutzer möglicherweise verwirrend, auch wenn das Ergebnis im Einzelfall sehr effektiv sein kann.

Daten analysieren

Manche Datenbankanwendungen dienen nicht der Anzeige, sondern der Analyse und Zusammenfassung von Datenbankinformationen. Der Benutzer kann dann aus diesen aufbereiteten Daten bestimmte Schlüsse ziehen.

Die Komponente *TDBChart* in der Registerkarte *Datensteuerung* der Komponentenpalette ermöglicht die Darstellung von Datenbankinformationen in grafischer Form. Benutzer können so die Bedeutung bestimmter Informationen schneller erfassen.

In der Komponentenpalette der Delphi-Versionen ist zusätzlich die Registerkarte *Datenanalyse* mit sechs Komponenten zur Durchführung von Datenanalysen und Kreuztabellenoperationen enthalten. Weitere Informationen über die Verwendung der Datenanalysekomponenten finden Sie in Kapitel 27, »Entscheidungskomponenten«.

Für Zusammenfassungen auf der Grundlage beliebig gruppierter Kriterien können Sie aktive Zusammenfassungen in Verbindung mit einer Client-Datenmenge verwenden. Weitere Informationen über aktive Zusammenfassungen finden Sie unter »Gewartete Aggregate verwenden« auf Seite 24-10.

Anzeige der Daten festlegen

Die Daten, die in der Anwendung angezeigt werden sollen, entsprechen oft nicht genau den Daten in der Tabelle. Manchmal wollen Sie vielleicht nur eine Untermenge der Felder oder Datensätze einer Tabelle anzeigen oder Informationen mehrerer Tabellen in einer verbundenen Ansicht kombinieren.

Welche Daten der Anwendung zur Verfügung stehen, bestimmen Sie durch die Wahl der Datenmengenkomponenten. Datenmengen abstrahieren die Eigenschaften und Methoden einer bestimmten Tabelle. Es ist deshalb unerheblich, ob die Daten aus einer oder aus mehreren Tabellen stammen. Weitere Informationen über die gemeinsamen Eigenschaften und Methoden von Datenmengen finden Sie in Kapitel 18, »Datenmengen«.

Eine Anwendung kann mehrere Datenmengen enthalten. Jede Datenmenge stellt eine logische Tabelle dar. Die Verwendung von Datenmengen grenzt die Anwendungslogik gegen die physikalischen Datenbanktabellen ab. Daher muß gegebenenfalls der Typ der Datenmengenkomponente oder die Methode zum Abrufen der Daten geändert werden, die restliche Benutzeroberfläche kann jedoch ohne Änderungen weiterverwendet werden.

Erfolgt der Zugriff auf die Daten über die BDE, können Sie die folgenden Typen von Datenmengen verwenden:

- **Tabellenkomponenten (TTable):** Tabellenkomponenten entsprechen exakt den zugrundeliegenden Tabellen in der Datenbank. Mit persistenten Feldkomponenten können Sie festlegen, welche Felder angezeigt werden, und Lookup-Felder und berechnete Felder hinzufügen. Durch die Verwendung von Wertebereichen und Filtern kann die Anzahl der anzuzeigenden Datensätze begrenzt werden. Eine genauere Beschreibung von Tabellen finden Sie in Kapitel 20, »Tabellen«. Per-

sistente Felder werden im Abschnitt »Persistente Feldkomponenten« auf Seite 19-4, Wertebereiche und Filter in »Mit Teilmengen der Daten arbeiten« auf Seite 20-12 beschrieben.

- **Abfragekomponenten (TQuery):** Abfragen sind die am häufigsten verwendeten Mechanismen zur Definition des Inhalts BDE-basierter Datenmengen. Mit Abfragen können Daten aus mehreren Tabellen über Verknüpfungen kombiniert und die Anzahl der anzuzeigenden Felder und Datensätze über SQL-Kriterien begrenzt werden. Weitere Informationen über Abfragen finden Sie in Kapitel 21, »Abfragen«.
- **Stored Procedures (TStoredProc):** Stored Procedures sind Gruppen von SQL-Anweisungen, die unter einem bestimmten Namen auf einem SQL-Server gespeichert sind. Eine Stored-Procedure-Komponente kann zur Definition eine Stored Procedure verwendet werden, durch die der Datenbankserver eine gewünschte Datenmenge liefern soll. Weitere Informationen über Stored Procedures finden Sie in Kapitel 22, »Stored Procedures«.
- **Verschachtelte Datenmengen (TNestedTable):** Verschachtelte Datenmengen stellen die Datensätze einer verschachtelten Detailmenge von Oracle8 dar. Mit Delphi können keine Oracle8-Tabellen mit verschachtelten Datenmengenfeldern erstellt werden. Verschachtelte Datenmengen dienen jedoch zur Anzeige und Bearbeitung von Daten bestehender Datenmengenfelder. Verschachtelte Datenmengen erhalten ihre Daten von einer Feldkomponente in einer Datenmenge mit Oracle8-Daten. Weitere Informationen über die Verwendung von verschachtelten Datenmengen finden Sie unter »Verschachtelte Tabellen« auf Seite 20-29 und »Datenmengenfelder« auf Seite 19-30.

Erfolgt der Zugriff auf die Daten über ADO, können die folgenden Typen von Datenmengen eingesetzt werden:

- **ADO-Datenmengen (TADODataset):** ADO-Datenmengen stellen den flexibelsten Mechanismus für den Zugriff auf Daten über ADO dar. ADO-Datenmengen können eine einzelne Datenbanktabelle oder das Ergebnis einer SQL-Abfrage darstellen. Sie können mit Hilfe persistenter Feldkomponenten festlegen, welche Felder verfügbar sind (einschließlich Lookup- und berechneter Felder). Außerdem können Sie die anzuzeigenden Datensätze mit Bereichen und Filtern beschränken. Zum Generieren der Daten können Sie eine SQL-Anweisung angeben. ADO-Datenmengen werden ausführlich unter »Gemeinsame Merkmale aller ADO-Datenmengenkomponenten« auf Seite 23-14 und »TADODataSet verwenden« auf Seite 23-21 beschrieben.
- **ADO-Tabellenkomponenten (TADOTable):** ADO-Tabellen entsprechen direkt den zugrundeliegenden Tabellen in der Datenbank. Sie können mit Hilfe persistenter Feldkomponenten festlegen, welche Felder verfügbar sind (einschließlich Lookup- und berechneter Felder). Außerdem können Sie die anzuzeigenden Datensätze mit Bereichen und Filtern beschränken. ADO-Tabellen werden im Abschnitt »TADOTable verwenden« auf Seite 23-22 ausführlich beschrieben.
- **ADO-Abfragekomponenten (TADOQuery):** ADO-Abfragen repräsentieren die Ergebnismenge der Ausführung eines SQL-Befehls oder einer DDL-Anweisung (Data Definition Language). Weitere Informationen zu Abfragen finden Sie unter »TADOQuery verwenden« auf Seite 23-24.

- **ADO Stored Procedures (TADOStoredProc):** Ist auf dem Datenbankserver eine Stored Procedure definiert, welche die gewünschte Datenmenge zurückgibt, können Sie eine ADO-Stored-Procedure-Komponente verwenden. Weitere Informationen zu ADO Stored Procedures finden Sie unter »TADOStoredProc verwenden« auf Seite 23-25.

Wenn Sie InterBase als Datenbankserver einsetzen, können Sie jeden der folgenden Typen von Datenmengen verwenden:

- **IB-Datenmengen (TIBDataSet):** IB-Datenmengen repräsentieren die Ergebnismenge einer SQL-Anweisung (üblicherweise eine SELECT-Anweisung). Sie können SQL-Anweisungen zum Auswählen und Aktualisieren der in der Datenmenge zwischengespeicherten Daten angeben.
- **IB-Tabellenkomponenten (TIBTable):** IB-Tabellen rufen Daten direkt aus einer Interbase-Tabelle oder -Ansicht ab. Sie können mit Hilfe persistenter Feldkomponenten festlegen, welche Felder verfügbar sind (einschließlich Lookup- und berechneter Felder). Außerdem können Sie die anzuzeigenden Datensätze mit Filtern beschränken.
- **IB-Abfragekomponenten (TIBQuery):** IB-Abfragen stellen die Ergebnismenge der Ausführung eines SQL-Befehls dar. Sie bilden die am einfachsten skalierbare Komponente beim Übergang vom lokalen InterBase-Server auf einen InterBase-Remote-Server.
- **IB Stored Procedures (TIBStoredProc)**: *IBStoredProc* führt eine InterBase Stored Procedure des Typs *Execute* aus. Diese Datenmengen geben keine Ergebnismenge zurück. Sie müssen *TIBDataSet* oder *TIBQuery* für Stored Procedures verwenden, die eine Ergebnismenge zurückgeben.

Delphi stellt die folgenden Möglichkeiten zur Verfügung, wenn Sie weder BDE, ADO noch InterBase einsetzen:

- **Client-Datenmengen (TClientDataSet):** Client-Datenmengen verwalten die Datensätze der logischen Datenmengen im Arbeitspeicher und können daher nur eine begrenzte Anzahl Datensätze enthalten. Client-Datenmengen werden folgendermaßen mit Daten gefüllt: vom Anwendungsserver oder mit unstrukturierten Daten, die auf einem Datenträger gespeichert sind. Sie benötigen keine Datenbank-Engine wie BDE oder ADO, sondern basieren auf einer einzelnen DLL (MIDAS.DLL). Weitere Informationen über Client-Datenmengen finden Sie in Kapitel 24, »Client-Datenmengen«.
- **Benutzerdefinierte Datenmengen:** Sie können von der Klasse *TDataSet* benutzerdefinierte Nachkommen ableiten, um Daten darzustellen, auf die Sie in Ihrer Anwendung zugreifen. Benutzerdefinierte Datenmengen ermöglichen eine beliebige Art der Datenverwaltung, während die Benutzeroberfläche weiterhin mit VCL-Komponenten erstellt wird. Weitere Informationen über benutzerdefinierte Komponenten finden Sie in Kapitel 31, »Die Komponentenentwicklung im Überblick«.

Berichte erstellen

Mit den Berichtskomponenten der Registerkarte *QReport* der Komponentenpalette können Daten aus Datenmengen gedruckt werden. Die Komponenten ermöglichen die Erstellung optisch ansprechender Berichte zur Darstellung und Zusammenfassung von Tabellendaten. Sie haben die Möglichkeit, in Kopf- und Fußzeilen Zusammenfassungen zur Analyse von gruppierten Daten hinzuzufügen.

Mit dem Symbol *QuickReport* im Dialogfeld *Objektgalerie* fügen Sie Ihrer Anwendung einen Bericht hinzu. Wählen Sie im Hauptmenü *Datei* / *Neu*, und aktivieren Sie die Registerkarte *Geschäftlich*. Doppelklicken Sie auf das Symbol *QuickReport-Experte*, um den Experten zu starten.

Hinweis In der mit Delphi gelieferten QuickReport-Demo finden Sie ein Beispiel zum Einsatz der Komponenten auf der Registerkarte *QuickReport*.

Kapitel

13

Ein- und zweischichtige Anwendungen erstellen

Bei ein- und zweischichtigen Anwendungen werden Datenbankinformationen in derselben Anwendung bearbeitet, die auch die Benutzeroberfläche implementiert. Die Datenbearbeitungslogik ist also nicht in einer getrennten Schicht isoliert. Daher sind solche Anwendungstypen hauptsächlich für Datenbanken geeignet, auf die nur eine Anwendung zugreift. Ein- und zweischichtige Anwendungen können aber auch eingesetzt werden, wenn sich mehrere Anwendungen die Datenbankinformationen teilen. In diesem Fall muß die Datenbank einfach strukturiert sein und darf keine Datensemantik aufweisen, die alle Anwendungen duplizieren müssen.

Sie sollten zunächst mit der Entwicklung einer ein- oder zweischichtigen Anwendung beginnen, selbst wenn die spätere Erweiterung auf ein mehrschichtiges Modell vorgesehen ist. Durch diesen Ansatz wird der Anwendungsserver nicht durch die Entwicklung der Datenbearbeitungslogik blockiert und bleibt verfügbar, während die Benutzeroberfläche entworfen wird. Außerdem kann so vor umfangreichen Investitionen in ein großes Projekt mit mehreren Systemen ein einfacher und billiger Prototyp entwickelt werden. Durch die Isolierung der Datenbearbeitungslogik kann bereits im Vorfeld eine eventuelle Erweiterung auf eine mehrschichtige Anwendung berücksichtigt werden. Die isolierte Logik wird dann später einfach auf die mittlere Schicht verlagert.

Delphi unterstützt zwei Typen von einschichtigen Anwendungen: Anwendungen, die eine lokale Datenbank verwenden (beispielsweise Paradox, dBASE, Access oder Local Interbase) und Datenbankanwendungen für unstrukturierte Daten. Zweischichtige Anwendungen verwenden für den Zugriff auf eine Remote-Datenbank einen Treiber.

Die Erwägungen beim Schreiben einschichtiger Anwendungen für den Zugriff auf eine lokale Datenbank bzw. zweischichtiger Anwendungen sind grundsätzlich identisch und primär vom Mechanismus der Verbindung mit der Datenbank abhängig. Delphi stellt für diese Art von Anwendungen drei integrierte Mechanismen bereit:

- BDE-basierte Anwendungen
- ADO-basierte Anwendungen
- InterbaseExpress-Anwendungen

Datenbankanwendungen für unstrukturierte Daten basieren auf der in MIDAS.DLL verfügbaren Unterstützung für Client-Datenmengen.

BDE-Anwendungen

Ein- und zweischichtige BDE-Anwendungen werden im Prinzip auf die gleiche Weise geschrieben. In beiden Architekturen sind nämlich die Datenzugriffskomponenten (und die BDE) für das Lesen und Aktualisieren von Daten und für die Navigation zuständig.

Bei der Entwicklung von BDE-Anwendungen wird die BDE in die Anwendung integriert, was zu einer Vergrößerung der Anwendung und zu einer erhöhten Komplexität führt. Mehrere BDE-Anwendungen können sich jedoch die BDE teilen und deren vielfältige Vorteile nutzen. BDE-Anwendungen können beispielsweise auf die leistungsstarke Bibliothek der BDE-API-Aufrufe zugreifen und unterstützen folgende Funktionen, die in Datenbankanwendungen mit unstrukturierten Dateien nicht zur Verfügung stehen:

- Verbinden mit Datenbanken
- Verwenden von Transaktionen
- Zwischenspeichern von Aktualisierungen
- Erstellen und Umstrukturieren von Datenbanktabellen

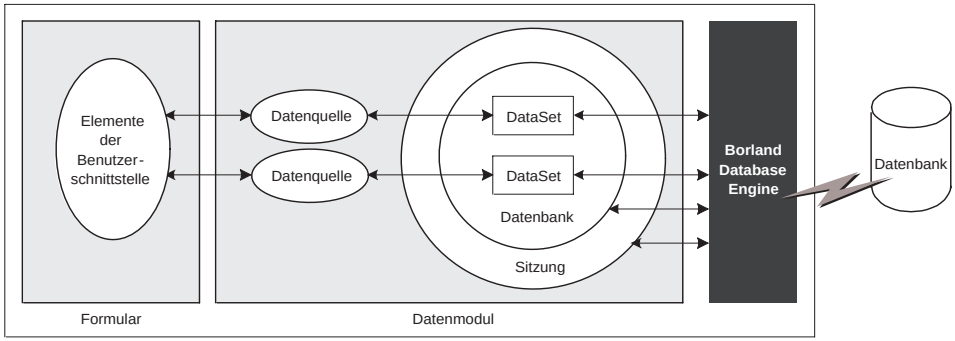
BDE-Architektur

Eine ein- oder zweischichtige BDE-Anwendung umfaßt folgende Elemente:

- Eine Benutzeroberfläche mit datensensitiven Steuerelementen.
- Eine oder mehrere Datenmengen, die Informationen aus den Datenbanktabellen darstellen.
- Eine Datenquellenkomponente für jede Datenmenge (zur Verbindung der datensensitiven Steuerelemente mit dieser Datenmenge).
- Eine oder mehrere optionale Datenbank-Komponenten zur Steuerung der Transaktionen in ein- und zweischichtigen Anwendungen und zur Verwaltung von Datenbankverbindungen in zweischichtigen Anwendungen.
- Eine oder mehrere optionale Sitzungskomponenten zur Isolation von Datenzugriffsoperationen wie Datenbankverbindungen und zur Verwaltung von Datenbankgruppen.

Die folgende Abbildung zeigt die Beziehungen zwischen diesen Elementen:

Abbildung 13.1 Komponenten einer BDE-Anwendung



Grundlagen von Datenbanken und Datenmengen

Datenbanken enthalten Informationen, die in Tabellen gespeichert sind. Sie können außerdem Tabellen über Datenbankinhalte, Objekte (z. B. Tabellenindizes) und SQL-Objekte (z. B. Stored Procedures) enthalten. Weitere Informationen über Datenbanken finden Sie in Kapitel 17, »Datenbankverbindungen«.

Die Registerkarte Datenzugriff der Komponentenpalette enthält verschiedene Datenmengenkomponenten zur Darstellung der tatsächlichen Datenbanktabellen und der auf ihnen basierenden logischen Tabellen. Weitere Informationen über diese Datenmengen finden Sie unter »Anzeige der Daten festlegen« auf Seite 12-14. Jede Anwendung, in der Datenbankinformationen bearbeitet werden, muß eine Datenmengenkomponente enthalten.

Jede BDE-Datenmengenkomponente der Registerkarte *Datenzugriff* verfügt über die als **published** deklarierte Eigenschaft *DatabaseName*. Diese Eigenschaft legt die Datenbank fest, deren Daten die Grundlage der jeweiligen Datenmenge bilden. Erst nach der Zuweisung einer Datenbank an die Eigenschaft *DatabaseName* kann die Datenmenge an bestimmte, in dieser Datenbank enthaltene Informationen gebunden werden. Der jeweilige Wert hängt von den nachfolgend beschriebenen Voraussetzungen ab.

- Wenn die Datenbank einen BDE-Alias besitzt, kann dieser als Wert für *DatabaseName* verwendet werden. Ein BDE-Alias steht stellvertretend für eine Datenbank und ihre Konfigurationsinformationen. Diese Konfigurationsinformationen differieren je nach Datenbanktyp (Oracle, Sybase, InterBase, Paradox, dBASE usw.). BDE-Aliase werden mit dem Dienstprogramm BDE-Konfiguration oder mit dem SQL-Explorer erstellt und verwaltet.
- Bei Verwendung einer Paradox- oder dBASE-Datenbank kann der Eigenschaft *DatabaseName* auch das Verzeichnis der Datenbanktabellen zugewiesen werden.
- Wenn Sie eine Datenbankanwendung entwickeln, verwenden Sie Datenbank-Komponenten. Diese Komponenten (*TDatabase*) repräsentieren eine Datenbank in einer Anwendung. Wenn Sie keine Datenbank-Komponente verwenden, wird automatisch eine temporäre, auf dem Wert der Eigenschaft *DatabaseName* basierende Komponente erstellt. Verwenden Sie hingegen explizit eine Datenbank-Kompo-

nente, erhält die Eigenschaft *DatabaseName* den Wert der *DatabaseName*-Eigenschaft der Datenbank-Komponente. Weitere Informationen über die Verwendung von Datenbank-Komponenten finden Sie unter »Persistente und temporäre Datenbank-Komponenten« auf Seite 17-1.

Sitzungen verwenden

Sitzungen dienen zur Isolierung von Datenzugriffsoperationen (z. B. Datenbankverbindungen) und zur Verwaltung von Datenbankgruppen. Bei der Verwendung einer Sitzung erfolgt der gesamte Zugriff auf die BDE im Kontext dieser Sitzung. Außerdem können mit Hilfe von Sitzungen Konfigurationsinformationen festgelegt und auf alle beteiligten Datenbanken übertragen werden. Mit dieser Funktion kann das mit der BDE-Konfiguration definierte Standardverhalten überschrieben werden.

Die folgenden Abschnitte beschreiben die Einsatzmöglichkeiten von Sitzungen.

- Verwaltung von BDE-Aliasen. Neue Aliase können erstellt und bestehende geändert oder gelöscht werden. Änderungen beziehen sich standardmäßig nur auf die jeweilige Sitzung, können jedoch der permanenten BDE-Konfigurationsdatei hinzugefügt werden. Weitere Informationen über die Verwaltung von BDE-Aliasen finden Sie unter »BDE-Aliase« auf Seite 16-10.
- Steuerung der Beendigung von Datenbankverbindungen in zweischichtigen Anwendungen. Datenbankverbindungen können auch dann aufrecht erhalten werden, wenn keine Datenmenge in der Datenbank aktiv ist. Solche Verbindungen binden zwar Ressourcen und verhindern deren Freigabe, erhöhen aber andererseits die Geschwindigkeit und reduzieren den Netzwerkverkehr. Um eine Datenbankverbindung ohne aktive Datenmenge offen zu halten, muß die Eigenschaft *KeepConnections* auf *True* (Standard) gesetzt werden.
- Zugriffsverwaltung für kennwortgeschützte Paradox- und dBASE-Dateien in einschichtigen Anwendungen. Die Sitzungskomponente wird von Datenmengen beim Zugriff auf kennwortgeschützte Paradox- und dBASE-Tabellen zur Bereitstellung eines Kennwortes verwendet. Standardmäßig wird immer, wenn ein Kennwort erforderlich ist, ein entsprechendes Abfrage-Dialogfeld angezeigt. Dieses Standardverhalten kann mit der Sitzungskomponente überschrieben werden, um das Kennwort programmseitig bereitzustellen. Soll eine einschichtige Anwendung später auf eine zwei- oder mehrschichtige Architektur erweitert werden, kann eine gemeinsame Benutzeroberfläche zur Abfrage von Benutzernamen und Kennwörtern erstellt werden. Bei der Umstellung auf einen Remote-Datenbankserver, der einen Benutzernamen und ein Kennwort anfordert, braucht diese Oberfläche nicht geändert zu werden. Weitere Informationen über die Verwendung von Sitzungen zur Verwaltung von Paradox- und dBASE-Kennwörtern finden Sie unter »Kennwortgeschützte Paradox- und dBase-Tabellen« auf Seite 16-15.
- Festlegung des Pfades für spezielle Paradox-Verzeichnisse. In einem Netzwerk verwenden Paradox-Datenbanken ein Netzwerkverzeichnis, das temporäre Dateien mit Sperrinformationen für Tabellen- und Datensätze enthält, und ein privates Verzeichnis für temporäre Dateien wie z. B. Abfrageergebnisse. Weitere Informationen über diese Verzeichnispfade finden Sie unter »Paradox-Verzeichnisse angeben« auf Seite 16-14.

Zur Isolierung von mehreren gleichzeitigen Zugriffen einer Anwendung auf dieselbe Datenbank müssen mehrere Sitzungen verwendet werden. Wenn Sie dies unterlassen, beeinträchtigen Sie möglicherweise die Logik der Transaktionen mit der betroffenen Datenbank (dies gilt auch für automatisch erstellte Transaktionen). Parallele Zugriffe finden bei gleichzeitigen Abfragen oder bei der Verwendung mehrerer Threads statt. Weitere Informationen über die Verwendung mehrerer Sitzungen finden Sie unter »Mehrere Sitzungen verwalten« auf Seite 16-17.

Wenn Sie keine mehrfachen Sitzungen benötigen, können Sie die Standardsitzung verwenden.

Verbindung mit Datenbanken

Die BDE (Borland Database Engine) enthält Treiber für unterschiedliche Datenbanken. In der Standard-Version von Delphi stehen nur Treiber für lokale Datenbanken zur Verfügung: Paradox, dBASE, FoxPro und Access. Die Professional-Version wird mit einem ODBC-Adapter ausgeliefert, der die BDE zur Verwendung von ODBC-Treibern befähigt. Durch einen ODBC-Treiber kann eine Anwendung jede ODBC-kompatible Datenbank verwenden. Einige Delphi-Versionen enthalten zusätzliche Treiber für Remote-Datenbankserver. Treiber für SQL Links dienen zur Kommunikation mit Remote-Datenbankservern wie InterBase, Oracle, Sybase, Informix, Microsoft SQL Server und DB2.

Hinweis Zweischichtige BDE-Anwendungen unterscheiden sich von einschichtigen BDE-Anwendungen nur durch die Verwendung von Remote-Datenbankservern anstelle lokaler Datenbanken.

Transaktionen

Eine Transaktion besteht aus einer Gruppe von Aktionen. Diese Aktionen müssen in einer oder mehreren Datenbanktabellen erfolgreich abgeschlossen werden, bevor das Ergebnis der Transaktion in die Datenbank übernommen wird. Schlägt nur eine der zugehörigen Aktionen fehl, werden alle Aktionen dieser Gruppe rückgängig gemacht (*Rollback*). Transaktionen verhindern also einen inkonsistenten Status der Datenbank, wenn bei der Ausführung einer Aktion ein Problem auftritt.

In Finanzanwendungen dienen Transaktionen beispielsweise zur Absicherung von Überweisungen von einem Konto auf ein anderes. Wenn eine Überweisung von einem Konto abgebucht wird und ein Fehler auftritt, bevor der Betrag dem anderen Konto gutgeschrieben werden kann, wird die Transaktion rückgängig gemacht, und die korrekten Kontostände werden wiederhergestellt.

Standardmäßig bietet die BDE eine implizite Transaktionssteuerung. Bei Anwendungen mit Transaktionssteuerung wird für jeden Datensatz einer Datenmenge, der in die zugrundeliegende Tabelle geschrieben werden soll, eine eigene Transaktion verwendet. Implizite Transaktionen reduzieren Konflikte bei Datensatzaktualisierungen auf ein Minimum und garantieren eine konsistente Datenbank. Ihre Verwendung kann jedoch zu erhöhtem Netzwerkverkehr und verringerter Anwendungsleistung führen, da jeder Datensatz in einer eigenen Transaktion in die Datenbank geschrieben wird. Außerdem sind logische Operationen, die mehr als einen Datensatz umfassen,

nicht durch die implizite Transaktionssteuerung geschützt (z. B. die zuvor beschriebene Überweisung von einem Konto auf ein anderes).

Durch eine explizite Transaktionssteuerung können Sie den richtigen Zeitpunkt zum Starten, Eintragen oder Rückgängigmachen einer Transaktion selbst wählen. Diese Form der Steuerung sollte bei der Entwicklung von Anwendungen in einer Mehrbenutzerumgebung, insbesondere beim Zugriff auf Remote-SQL-Server, eingesetzt werden.

Hinweis Die Anzahl der benötigten Transaktionen kann durch die Zwischenspeicherung von Aktualisierungen verringert werden. Weitere Informationen über zwischengespeicherte Aktualisierungen finden Sie in Kapitel 25, »Zwischengespeicherte Aktualisierungen«.

Transaktionen explizit steuern

In einer BDE-Datenbankanwendung gibt es zwei Möglichkeiten der expliziten Transaktionssteuerung, die sich gegenseitig ausschließen:

- Die Verwendung der Methoden und Eigenschaften der Datenbank-Komponenten (z. B. *StartTransaction*, *Commit*, *Rollback*, *InTransaction* und *TransIsolation*). Der Hauptvorteil dieser Methode besteht in der Erstellung einer klaren und übertragbaren Anwendung, die von keiner bestimmten Datenbank und keinem bestimmten Server abhängig ist.
- Die Verwendung von Passthrough-SQL in einer Abfragekomponente zur direkten Übergabe von SQL-Anweisungen an Remote-SQL- oder ODBC-Server. Weitere Informationen über Abfragekomponenten finden Sie in Kapitel 21, »Abfragen«. Der Hauptvorteil von Passthrough-SQL besteht darin, daß die Fähigkeiten der einzelnen Datenbankserver zur Transaktionsverwaltung genutzt werden (z. B. Zwischenspeicherung eines Schemas). Einzelheiten zur Transaktionsverwaltung finden Sie in der Dokumentation des jeweiligen Servers.

Einschichtige Anwendungen können kein Passthrough-SQL verwenden. Mit einer Datenbank-Komponente können jedoch eingeschränkte explizite Transaktionen für lokale Datenbanken erstellt werden. Weitere Informationen zur Verwendung lokaler Transaktionen finden Sie unter »Lokale Transaktionen« auf Seite 13-9.

Wenn Sie zweischichtige Anwendungen schreiben (was SQL Links erfordert), können Sie die Transaktionsverwaltung entweder über Datenbank-Komponenten oder über Passthrough-SQL realisieren. Weitere Informationen über die Verwendung von Passthrough-SQL finden Sie unter »Passthrough-SQL« auf Seite 13-9.

Datenbank-Komponenten für Transaktionen verwenden

Nach dem Start einer Transaktion werden alle nachfolgenden Anweisungen (Schreib- und Lesezugriffe auf die Datenbank) im Rahmen dieser Transaktion ausgeführt. Da jede einzelne Anweisung als Teil einer Gruppe betrachtet wird, werden Änderungen entweder vollständig in die Datenbank übernommen, oder jede in der Gruppe vorgenommene Änderung wird rückgängig gemacht.

Im Idealfall sollte eine Transaktion nicht länger dauern als nötig. Da während der Durchführung einer Transaktion Benutzer auf die Datenbank zugreifen und parallele

Transaktionen gestartet und beendet werden, steigt mit zunehmender Dauer der Transaktion die Wahrscheinlichkeit von Konflikten beim Eintragen von Änderungen in die Datenbank.

Bei Verwendung einer Datenbank-Komponente erfordert eine Transaktion die folgenden Schritte:

1 Starten Sie die Transaktion durch einen Aufruf der Methode *StartTransaction* der Datenbank:

```
DatabaseInterBase.StartTransaction;
```

2 Nach dem Start der Transaktion werden alle nachfolgenden Datenbankaktionen bis zur ausdrücklichen Beendigung der Transaktion als Teil der Transaktion betrachtet. Über die Eigenschaft *InTransaction* der Datenbank-Komponente können Sie feststellen, ob eine Transaktion ausgeführt wird. Während der Ausführung ist die Ansicht der Daten in den Datenbanktabellen durch die Isolationsstufe festgelegt. Weitere Informationen über Isolationsstufen finden Sie unter »Die Eigenschaft TransIsolation« auf Seite 13-7.

3 Wenn alle Aktionen einer Transaktion erfolgreich durchgeführt wurden, können die Änderungen mit der Methode *Commit* der Datenbank-Komponente dauerhaft eingetragen werden:

```
DatabaseInterBase.Commit;
```

Der Aufruf der Methode *Commit* wird normalerweise in eine **try...except**-Anweisung eingebunden. Dadurch kann im Falle einer unvollständigen Transaktion der Fehler im **except**-Block bearbeitet und die Operation wiederholt oder verworfen werden.

4 Bei der Durchführung von Änderungen im Rahmen einer Transaktion oder beim Eintragen der Transaktion selbst können Fehler auftreten. In diesem Fall müssen alle durchgeführten Änderungen der Transaktion verworfen werden. Diese Operation wird mit der Methode *Rollback*> der Datenbank-Komponente realisiert:

```
DatabaseInterBase.Rollback;
```

Die Methode *Rollback* wird normalerweise in folgenden Quelltext eingebunden:

- In die Exception-Behandlung, falls eine Korrektur des Datenbankfehlers nicht möglich ist.
- In die Ereignisbehandlungsroutine einer Schaltfläche oder eines Menüs namens *Abbrechen.*

Die Eigenschaft TransIsolation

Die Eigenschaft *TransIsolation* legt die Isolationstufe für die Transaktionen einer Datenbank fest. Die Isolationsstufe einer Transaktion regelt beim gleichzeitigen Zugriff anderer Transaktionen auf dieselbe Tabelle die gegenseitige Interaktion. Dies betrifft insbesondere die Sichtbarkeit von Tabellenänderungen, die andere Transaktionen vorgenommen haben.

Die Standardwert für *TransIsolation* ist *tiReadCommitted*. In der folgenden Tabelle sind die möglichen Werte für *TransIsolation* und ihre Bedeutung zusammengefaßt:

Tabelle 13.1 Mögliche Werte für die Eigenschaft TransIsolation

Isolationsstufe	Bedeutung
tiDirtyRead	Erlaubt das Lesen von nicht eingetragenen Änderungen paralleler Transaktionen. Nicht eingetragene Änderungen können jederzeit widerrufen werden (Rollback). Auf dieser Stufe ist die Transaktion am wenigsten von den Änderungen anderer Transaktionen isoliert.
tiReadCommitted	Erlaubt nur das Lesen der durch andere Transaktionen eingetragenen (permanenten) Änderungen. Dies ist die Standardisolationsstufe.
tiRepeatableRead	Erlaubt genau eine abgeschlossene Leseoperation in der Datenbank. Nachfolgende Änderungen durch andere, gleichzeitige Transaktionen sind für die Transaktion nicht sichtbar. Diese Stufe garantiert, daß sich ein gelesener Datensatz für die Transaktion nicht mehr ändert. Auf dieser Ebene ist die Transaktion am stärksten von Änderungen durch andere Transaktionen isoliert.

Die beschriebenen Isolationsstufen werden durch Datenbankserver unterschiedlich oder nur teilweise unterstützt. Die BDE verwendet automatisch die nächsthöhere Isolationsstufe, falls ein Server die angeforderte Isolationsstufe nicht unterstützt. Die derzeit von den gängigen Servern verwendeten Isolationsstufen zeigt die folgende Tabelle. Genauere Beschreibungen zur Implementierung der jeweiligen Isolationsstufe finden Sie in der Dokumentation des entsprechenden Servers.

Tabelle 13.2 Isolationsstufen

Server	Angeforderte Stufe	Tatsächliche Stufe
Oracle	tiDirtyRead tiReadCommitted tiRepeatableRead	tiReadCommitted tiReadCommitted tiRepeatableRead (READONLY)
Sybase, MS-SQL	tiDirtyRead tiReadCommitted tiRepeatableRead	tiReadCommitted tiReadCommitted Not supported
DB2	tiDirtyRead tiReadCommitted tiRepeatableRead	tiDirtyRead tiReadCommitted tiRepeatableRead
Informix	tiDirtyRead tiReadCommitted tiRepeatableRead	tiDirtyRead tiReadCommitted tiRepeatableRead
InterBase	tiDirtyRead tiReadCommitted tiRepeatableRead	tiReadCommitted tiReadCommitted tiRepeatableRead
Paradox, dBase, Access, FoxPro	tiDirtyRead tiReadCommitted tiRepeatableRead	tiDirtyRead Not supported Not supported

Hinweis Bei der Verwendung von Transaktionen in lokalen Paradox-, dBASE-, Access- oder FoxPro-Tabellen muß die Eigenschaft *TransIsolation* nicht auf den Standardwert

tiReadCommitted, sondern auf *tiDirtyRead* gesetzt werden, da ansonsten ein BDE-Fehler auftritt.

Kommuniziert eine Anwendung über ODBC mit einem Server, muß der zugehörige ODBC-Treiber auch die Isolationsstufe unterstützen. Weitere Informationen finden Sie in der Dokumentation des betreffenden ODBC-Treibers.

Passthrough-SQL

Mit Passthrough-SQL können Sie über die Komponenten *TQuery, TStoredProc* oder *TUpdateSQL* eine SQL-Anweisung zur Transaktionssteuerung direkt an einen Remote-Datenbankserver senden. Die SQL-Anweisung wird nicht von der BDE ausgeführt. Die Verwendung von Passthrough-SQL ermöglicht die direkte Nutzung der Transaktionssteuerung des Servers. Dieser Aspekt ist besonders vorteilhaft, wenn es sich dabei nicht um Standardfunktionen handelt.

So setzen Sie Passthrough-SQL zur Transaktionssteuerung ein:

- Installieren Sie die richtigen Treiber für SQL Links. Bei der Delphi-Standardinstallation werden die Treiber für SQL Links automatisch installiert.
- Konfigurieren Sie das Netzwerkprotokoll. Weitere Informationen dazu erhalten Sie beim zuständigen Netzwerk-Administrator.
- Greifen Sie auf eine Datenbank auf einem Remote-Server zu.
- Wenn Sie den SQL-Explorer verwenden, setzen Sie den Wert für SQLPASSTHRU MODE auf NOT SHARED. SQLPASSTHRU MODE legt fest, ob die BDE und die Passthrough-SQL-Anweisungen dieselben Datenbankverbindungen verwenden. In den meisten Fällen hat SQLPASSTHRU MODE den Wert SHARED AUTOCOMMIT. Wenn Sie Anweisungen zur Transaktionssteuerung verwenden, können die Datenbankverbindungen nicht gemeinsam genutzt werden. Weitere Informationen über SQLPASSTHRU MODE finden Sie in der Hilfedatei der BDE-Konfiguration.

Hinweis Wenn SQLPASSTHRU MODE auf NOT SHARED eingestellt ist, müssen Sie für Datenmengen, die SQL-Transaktionsanweisungen an den Server übergeben, und für solche, die dies nicht tun, getrennte Datenbank-Komponenten verwenden.

Lokale Transaktionen

Die BDE unterstützt lokale Transaktionen für lokale Paradox-, dBASE-, Access- und FoxPro-Tabellen. Aus der Sicht des Programmierers besteht zwischen einer lokalen Transaktion und einer Transaktion mit einem Remote-Datenbankserver kein Unterschied.

Nach dem Start einer Transaktion für eine lokale Tabelle werden die in der Tabelle vorgenommenen Aktualisierungen protokolliert. Jeder Protokolldatensatz enthält die vorherigen Daten eines Datensatzes. Für die Dauer der Transaktion bleiben alle zu aktualisierenden Datensätze bis zum Eintragen oder bis zum Rollback der Transaktion gesperrt. Bei einem Rollback dienen die ursprünglichen Daten zur Wiederherstellung des vorherigen Status der aktualisierten Datensätze.

Gegenüber Transaktionen mit SQL-Servern und ODBC-Treibern weisen lokale Transaktionen folgende Einschränkungen auf:

- Die automatische Wiederherstellung nach einem Programmabsturz ist nicht verfügbar.
- Anweisungen zur Datendefinition werden nicht unterstützt.
- Mit temporären Tabellen können keine Transaktionen ausgeführt werden.
- In einer Paradox-Datenbank können lokale Transaktionen nur in Tabellen mit gültigen Indizes ausgeführt werden, da ohne Index kein Rollback von Aktualisierungen möglich ist.
- Es kann nur eine begrenzte Anzahl von Datensätzen gesperrt und geändert werden. Bei Paradox-Tabellen sind dies 255 Datensätze, bei dBase-Tabellen 100.
- Mit dem ASCII-Treiber der BDE können keine Transaktionen ausgeführt werden.
- Für die Eigenschaft *TransIsolation* muß der Wert *tiDirtyRead* gesetzt werden.
- Das Schließen des Cursors einer Tabelle während einer Transaktion bewirkt für die einzelnen Aktionen dieser Transaktion ein Rollback. Dies gilt jedoch nicht, wenn
  - mehrere Tabellen geöffnet sind;
  - an der betreffenden Tabelle keine Änderungen vorgenommen wurden.

Aktualisierungen zwischenspeichern

Die Borland Database Engine unterstützt die Zwischenspeicherung von Aktualisierungen. Bei dieser Funktion ruft die Anwendung Daten aus einer Datenbank ab, führt an einer lokalen, zwischengespeicherten Kopie der Daten Änderungen durch und speichert diese Änderungen als Ganzes in der Datenmenge. Zwischengespeicherte Aktualisierungen werden in einer einzigen Transaktion zur Datenbank übertragen.

Die Zwischenspeicherung von Aktualisierungen verkürzt die Transaktionszeiten und verringert den Netzwerkverkehr. Da die Daten jedoch lokal in der Anwendung zwischengespeichert werden, unterliegen sie keiner Transaktionssteuerung und sind für andere Anwendungen nicht sichtbar. Während der Bearbeitung der lokalen Datenkopien können andere Anwendungen in der zugrundeliegenden Datenbanktabelle Änderungen vornehmen. Das Verfahren ist also ungeeignet, wenn eine Anwendung mit Daten arbeitet, die von anderen Benutzern ständig geändert werden.

Die Eigenschaft *CachedUpdates* veranlaßt BDE-Datenmengen zur Zwischenspeicherung von Aktualisierungen. Nach der Bestätigung der Änderungen können diese zur Datenmengenkomponente, zur Datenbank-Komponente oder zu einem speziellen Aktualisierungsobjekt übertragen werden. Ist die Übertragung der Änderungen mit zusätzlichen Berechnungen verbunden (beispielsweise bei einer Abfrage über mehrere Tabellen), muß das Ereignis *OnUpdateRecord* verwendet werden, um die Änderungen in jede beteiligte Tabelle einzutragen.

Weitere Informationen über zwischengespeicherte Aktualisierungen finden Sie in Kapitel 25, »Zwischengespeicherte Aktualisierungen«.

Hinweis Wenn Sie mit zwischengespeicherten Aktualisierungen arbeiten, sollten Sie wegen der besseren Steuerungsmöglichkeiten den Wechsel zu einem mehrschichtigen Modell ins Auge fassen. Informationen darüber finden Sie in Kapitel 14, »Mehrschichtige Anwendungen erstellen«.

Datenbanktabellen erstellen und umstrukturieren

In BDE-Anwendungen dient die Komponente *TTable* zur Erstellung neuer Datenbanktabellen und zum Hinzufügen von Indizes zu bestehenden Tabellen.

Tabellen können entweder zur Entwurfszeit im Formular-Designer oder zur Laufzeit erstellt werden. Beim Erstellen einer Tabelle richten Sie mit der Eigenschaft *FieldDefs* die benötigten Felder ein, fügen mit der Eigenschaft *IndexDefs* beliebige Indizes hinzu und rufen die Methode *CreateTable* auf. Alternativ dazu können Sie auch den Befehl *Tabelle erstellen* im lokalen Menü der Tabelle verwenden. Detaillierte Anweisungen zur Erstellung von Tabellen finden Sie im Abschnitt »Tabellen erstellen« auf Seite 20-19.

Hinweis Beim Erstellen von Oracle8-Tabellen können keine Objektfelder (ADT-Felder, Array-Felder, Referenzfelder und Datenmengenfelder) eingesetzt werden.

Die Umstrukturierung von Tabellen zur Laufzeit erfolgt im Unterschied zum Hinzufügen von Indizes mit der BDE-API-Funktion *DbiDoRestructur*e. Indizes können einer bestehenden Tabelle mit der *TTable-Methode AddIndex* hinzugefügt werden.

Hinweis Zur Entwurfszeit können Sie zum Erstellen und Umstrukturieren von Paradox- und dBASE-Tabellen den Datenbank-Desktop verwenden. Um Tabellen auf Remote-Datenbankservern zu erstellen und umzustrukturieren, verwenden Sie den SQL-Explorer und SQL.

ADO-basierte Anwendungen

Delphi-Anwendungen, die für den Datenzugriff ADO-Komponenten verwenden, sind entweder ein- oder zweischichtig. Die Kategorie einer Anwendung ist vom verwendeten Datenbanktyp anhängig. Wird mit ADO beispielsweise auf eine Microsoft-SQL-Server-Datenbank zugegriffen, handelt es sich immer um eine zweischichtige Anwendung, da SQL Server ein SQL-Datenbanksystem ist. SQL-Datenbanksysteme befinden sich normalerweise auf einem dedizierten SQL-Server. Auf der anderen Seite sind Anwendungen, die mit ADO auf lokale Datenbanken wie dBASE oder FoxPro zugreifen, immer einschichtige Anwendungen.

Es gibt vier Hauptbereiche, die für den Datenbankzugriff mit ADO und den ADO-Komponenten von Delphi berücksichtigt werden müssen. Diese Bereiche gelten sowohl für ein- als auch für zweischichtige Anwendungen:

- ADO-basierte Architektur
- Verbindung zu ADO-Datenbanken herstellen

- Daten abrufen
- ADO-Datenbanktabellen erstellen und umstrukturieren

ADO-basierte Architektur

Eine ADO-basierte Anwendung enthält die folgenden funktionalen Bereiche:

- Benutzeroberfläche mit visuellen, datensensitiven Steuerelementen. Die visuellen Datensteuerelemente werden nicht benötigt, wenn alle Datenzugriffe manuell programmiert werden.
- Mindestens eine Datenmengenkomponente, die Informationen aus Tabellen oder Abfragen darstellt.
- Eine Datenquellenkomponente für jede Datenmengenkomponente, die als Nahtstelle zwischen der Datenmengenkomponente und den visuellen datensensitiven Steuerelementen dient.
- Eine Verbindungskomponente zum Herstellen der Verbindung mit dem ADO-Datenspeicher. Die Verbindungskomponente dient als Nahtstelle zwischen den Datenmengenkomponenten der Anwendung und der Datenbank, auf die über den Datenspeicher zugegriffen wird.

Die ADO-Schicht einer ADO-basierten Delphi-Anwendung besteht aus Microsoft ADO 2.1, einem OLE-DB-Provider oder ODBC-Treiber für den Datenspeicherzugriff, Client-Software für das verwendete Datenbanksystem (bei SQL-Datenbanken), einem Datenbank-Backend-System, auf das die Anwendung zugreifen kann (für SQL-Datenbanksysteme), und einer Datenbank. Die ADO-Anwendung muß auf jede dieser Komponenten zugreifen können, damit der volle Funktionsumfang verfügbar ist.

Grundlagen zu ADO-Datenbanken und -Datenmengen

Die Registerkarte *ADO* (ActiveX-Datenobjekte) der Komponentenpalette enthält alle Komponenten, die zum Herstellen von Datenbankverbindungen und für den Zugriff auf die enthaltenen Tabellen benötigt werden.

Alle Metadatenobjekte einer ADO-basierten Anwendung befinden sich in der Datenbank, auf die über den ADO-Datenspeicher zugegriffen wird. Bevor eine Anwendung auf diese Objekte oder die in ihnen enthaltenen Daten zugreifen kann, muß eine Verbindung zum Datenspeicher hergestellt werden. Weitere Informationen finden Sie unter »Verbindungen zu ADO-Datenspeichern einrichten« auf Seite 23-3.

Die Daten einer Datenbank werden in mindestens einer Tabelle gespeichert. Sie müssen mindestens eine ADO-Datenmengenkomponente (*TADODataSet, TADOQuery* usw.) in Ihre Anwendung aufnehmen, um mit den in Datenbanktabellen gespeicherten Daten arbeiten zu können. Unter »Daten abrufen« auf Seite 13-13 und »ADO-Datenmengen verwenden« auf Seite 23-13 finden Sie weitere Informationen zur Verwendung von ADO-Datenmengenkomponenten für den Zugriff auf die Daten in den Tabellen.

Verbindung zu ADO-Datenbanken herstellen

Eine ADO-basierte Delphi-Anwendung verwendet Microsoft ActiveX Data Objects (ADO) 2.1 für die Interaktion mit einem OLE-DB-Provider, um die Verbindung zu einem Datenspeicher herzustellen und auf die Daten zuzugreifen. Ein Datenspeicher kann beispielsweise eine Datenbank repräsentieren. Eine ADO-basierte Anwendung setzt voraus, daß ADO 2.1 auf dem Client-Computer installiert ist. ADO und OLE DB werden von Microsoft bereitgestellt und mit Windows installiert.

Der Provider kann eine Reihe von Zugriffstypen ermöglichen, von nativen OLE-DB-Treibern bis hin zu ODBC-Treibern. Diese Treiber müssen auch auf dem Client-Computer installiert werden. OLE-DB-Treiber für unterschiedliche Datenbanksysteme werden vom Datenbankhersteller oder Fremdherstellern bereitgestellt.

Wenn die Anwendung eine SQL-Datenbank wie Microsoft SQL Server oder Oracle verwendet, muß die Client-Software für das Datenbanksystem ebenfalls auf dem Client-Computer installiert werden. Die Client-Software wird vom Datenbankhersteller geliefert und von der CD (oder Diskette) mit dem Datenbanksystem installiert.

Um die Anwendung mit dem Datenspeicher zu verbinden, wird die ADO-Verbindungskomponente für die Verwendung eines der verfügbaren Provider konfiguriert. Sobald die Verbindungskomponente mit dem Datenspeicher verbunden ist, können Datenmengenkomponenten für den Zugriff auf die Tabellen in der Datenbank den Verbindungskomponenten zugeordnet werden. Unter »Verbindungen zu ADO-Datenspeichern einrichten« auf Seite 23-3 finden Sie Informationen zum Einrichten der Verbindung mit Datenspeichern.

Die Verbindungskomponente ermöglicht nicht nur den Zugriff der Anwendung auf die Datenbank, sondern kapselt auch die ADO-Funktionalität zur Transaktionsverarbeitung.

Daten abrufen

Sobald eine Anwendung eine gültige Verbindung zu einer Datenbank hergestellt hat, kann mit Hilfe von Datenmengenkomponenten auf die Daten in der Datenbank zugegriffen werden. Die Registerkarte *ADO* der Komponentenpalette enthält die für den Zugriff auf Daten in ADO-Datenspeichern erforderlichen ADO-Datenmengenkomponenten.

Zu diesen Komponenten gehören *TADODataSet, TADOTable, TADOQuery* und *TADOStoredProc*. Diese Komponenten können Daten aus ADO-Datenspeichern abrufen, bearbeiten und für den Benutzer einer Anwendung anzeigen, um die interaktive Nutzung der Daten zu ermöglichen. Das Abrufen und Bearbeiten von Daten mit ADO-Datenmengenkomponenten wird unter »ADO-Datenmengen verwenden« auf Seite 23-13 beschrieben.

Verwenden Sie die mitgelieferten datensensitiven Steuerelemente, um den visuellen Zugriff auf die Daten mit einer ADO-Datenmengenkomponente zu ermöglichen. Es gibt keine ADO-spezifischen datensensitiven Steuerelemente. Ausführliche Informationen zum Einsatz der datensensitiven Steuerelemente finden Sie unter »Datensensitive Steuerelemente im Überblick« auf Seite 26-1.

Die Datenquellen-Standardkomponente dient als Nahtstelle zwischen den ADO-Datenmengenkomponenten und den datensensitiven Steuerelementen. Es gibt keine dedizierte Datenquellenkomponente für ADO. Ausführliche Informationen zum Einsatz von Datenquellenkomponenten finden Sie unter »Datenquellen verwenden« auf Seite 26-6.

Bei Bedarf können persistente Feldobjekte verwendet werden, um die Felder in ADO-Datenmengenkomponenten darzustellen. Verwenden Sie wie bei datensensitiven Steuerelementen und Datenquellenkomponenten einfach die inhärenten Delphi-Feldklassen (*TField* und *Nachkommen*). Informationen zur Verwendung dynamischer und persistenter Feldobjekte finden Sie unter »Was sind Feldkomponenten?« auf Seite 19-2.

ADO-Datenbanktabellen erstellen und umstrukturieren

Das Erstellen und Löschen von Metadaten in einer ADO-Datenbank durch eine Delphi-Anwendung erfolgt mit Hilfe von SQL. Auch das Umstrukturieren von Tabellen wird mit SQL-Anweisungen durchgeführt. Das Ändern anderer Metadatenobjekte ist nicht immer direkt möglich. Sie müssen vielmehr das Metadatenobjekt löschen und dann durch ein neues mit anderen Attributen ersetzen.

Es gibt viele Datenbanktypen, auf die über ADO zugegriffen werden kann. Nicht alle Treiber für eine bestimmte Datenbank unterstützen dieselbe SQL-Syntax. Dieses Handbuch kann nicht alle Varianten der SQL-Syntax beschreiben. Eine ausführliche und aktuelle Beschreibung der SQL-Implementierung eines bestimmten Datenbanksystems finden Sie in der mit dem Datenbanksystem gelieferten Dokumentation.

Grundsätzlich werden Tabellen mit der Anweisung CREATE TABLE in der Datenbank erstellt, während CREATE INDEX der Erstellung von Indizes für diese Tabellen dient. Sofern diese unterstützt werden, können Sie mit anderen CREATE-Anweisungen wie CREATE DOMAIN, CREATE VIEW und CREATE SCHEMA verschiedene Metadatenobjekte hinzufügen.

Zu jeder der CREATE-Anweisungen existiert eine entsprechende DROP-Anweisung, mit der das Metadatenobjekt gelöscht wird. Die Anweisungen lauten beispielsweise DROP TABLE, DROP VIEW, DROP DOMAIN und DROP SCHEMA.

Mit der Anweisung ALTER TABLE können Sie die Struktur einer Tabelle ändern. ALTER TABLE besitzt die Klauseln ADD und DROP zum Erstellen und Löschen von Elementen in einer Tabelle. Verwenden Sie beispielsweise die Klausel ADD COLUMN, um eine neue Spalte in die Tabelle einzufügen, oder DROP CONSTRAINT, um eine Beschränkung aus der Tabelle zu löschen.

Sie können diese Metadaten-Anweisungen mit der ADO-Befehlskomponente bzw. der ADO-Abfragekomponente einsetzen. Informationen zum Ausführen von Befehlen mit der ADO-Befehlskomponente finden Sie unter »Befehle ausführen« auf Seite 23-30.

Datenbankanwendungen mit unstrukturierten Daten

Datenbankanwendungen mit unstrukturierten Daten sind einschichtige Anwendungen, die zur Darstellung aller Datenmengen *TClientDataSet*-Komponenten verwenden. Da eine Client-Datenmenge sämtliche Daten im Speicher hält, eignet sich dieser Anwendungstyp nicht für sehr umfangreiche Datenmengen.

Datenbankanwendungen mit unstrukturierten Daten benötigen nicht die Borland Database Engine (BDE) oder ActiveX Data Objects (ADO). Sie benutzen statt dessen nur die Datei MIDAS.DLL. Dadurch wird die Weitergabe von Datenbankanwendungen mit unstrukturierten Dateien vereinfacht, da ja die Installation, Konfiguration und Pflege der Software zur Verwaltung der Datenbankverknüpfungen entfällt.

Weil solche Anwendungen keine Datenbank benutzen, bieten Sie auch keine Mehrbenutzerunterstützung. Vielmehr stehen die Datenmengen ausschließlich der betreffenden Anwendung zur Verfügung. Die Daten können in unstrukturierten Dateien auf einem Datenträger gespeichert und später von dort abgerufen werden, doch gibt es keinen integrierten Schutzmechanismus, der in einer Mehrbenutzerumgebung das gegenseitige Überschreiben von Daten verhindern würde.

Datenbankanwendungen mit unstrukturierten Dateien basieren auf Client-Datenmengen (in der Registerkarte *MIDAS* der Komponentenpalette), die die meisten Datenbankoperationen mit anderen Datenmengen unterstützen. Sie verwenden dieselben datensensitiven Steuerelemente und Datenquellenkomponenten wie einschichtige BDE-Anwendungen. Der Einsatz von Datenbank-Komponenten entfällt, da weder die Verwaltung von Datenbankverbindungen noch die Unterstützung von Transaktionen erforderlich ist. Sitzungskomponenten sind nur bei Multithread-Anwendungen erforderlich. Weitere Informationen über die Verwendung von Client-Datenmengen finden Sie in Kapitel 24, »Client-Datenmengen«.

Der Hauptunterschied beim Schreiben von Anwendungen mit unstrukturierten Dateien und anderen einschichtigen Datenbankanwendungen besteht in den Methoden zum Erstellen von Datenmengen und zum Laden und Speichern von Daten.

Datenmengen erstellen

Da Datenbankanwendungen mit unstrukturierten Dateien auf keine Datenbanken zugreifen, ist der Entwickler solcher Anwendungen für die Erstellung der Datenmengen selbst verantwortlich. Eine Datenmenge kann in Tabellenform in einer Datei gespeichert und aus ihr wieder geladen werden. Die Indizes werden dagegen nicht mit der Tabelle gespeichert und müssen bei jedem Laden der Tabelle neu erstellt werden.

Vor dem Schreiben der eigentlichen Datenbankanwendung sollten Sie zunächst leere Dateien für die Datenmengen erstellen und speichern. Damit entfällt die Definition der Metadaten für die Client-Datenmengen in der Anwendung.

Das Verfahren zur Erstellung einer Client-Datenmenge hängt davon ab, ob eine vollständig neue Datenmenge erstellt oder eine bestehende BDE-Anwendung konvertiert werden soll.

Neue Datenmengen mit persistenten Feldern erstellen

So erstellen Sie eine neue Client-Datenmenge mit dem Felder-Editor:

1. Fügen Sie der Anwendung eine *TClientDataSet*-Komponente aus der Registerkarte *MIDAS* der Komponentenpalette hinzu.
2. Klicken Sie mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie *Felder-Editor*. Klicken Sie im *Felder-Editor* erneut mit der rechten Maustaste, und wählen Sie *NeuesFeld*. Legen Sie anschließend die Grundeigenschaften der Felddefinition fest. Nach der Erstellung des Feldes können Sie seine Eigenschaften im Objektinspektor ändern, indem Sie das Feld im *Felder-Editor* auswählen.

   Fügen Sie weitere Felder hinzu, bis die Client-Datenmenge vollständig ist.
3. Klicken Sie mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie den Befehl *Datenmenge* erstellen. Dieser Befehl erstellt eine leere Client-Datenmenge aus den persistenten Feldern, die im *Felder-Editor* hinzugefügt wurden.
4. Klicken Sie mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie den Befehl *In Datei speichern*. Dieser Befehl steht nur bei einer leeren Client-Datenmenge zur Verfügung.
5. Wählen Sie im Dialogfeld *Datei speichern* einen Dateinamen, und speichern Sie die Client-Datenmenge in einer unstrukturierten Datei.

Hinweis Aus den persistenten Feldern, die zusammen mit einer Client-Datenmenge gespeichert wurden, können Client-Datenmengen auch zur Laufzeit erstellt werden. Rufen Sie dazu einfach die Methode *CreateDataSet* auf.

Datenmengen mit Feld- und Indexdefinitionen erstellen

Das Erstellen einer Client-Datenmenge mit Feld- und Indexdefinitionen ähnelt der Erstellung einer Datenbanktabelle mit einer *TTable*-Komponente. Die Eigenschaften *DatabaseName*, *TableName* und *TableType* brauchen hier jedoch nicht festgelegt zu werden, da sie für Client-Datenmengen keine Rolle spielen. Wie bei *TTable* dient die Eigenschaft *FieldDefs* zur Festlegung der Tabellenfelder und die Eigenschaft *IndexDefs* zur Definition beliebiger Indizes. Klicken Sie zur Entwurfszeit nach der Festlegung der Tabelleneigenschaften mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie den Befehl *Datenmenge erstellen*. Zur Laufzeit rufen Sie die Methode *CreateDataSet* auf.

Zum Festlegen der Indexdefinitionen für Client-Datenmengen gibt es zwei spezielle Eigenschaften: *TIndexDef.DescFields* und *TIndexDef.CaseInsFields*.

Die Eigenschaft *DescFields* ermöglicht die Festlegung von Indizes, mit denen Datensätze nach einigen Feldern in aufsteigender Reihenfolge und nach anderen Feldern in absteigender Reihenfolge sortiert werden können. Anstatt mit *ixDescending* alle Indexfelder in absteigender Reihenfolge zu sortieren, geben Sie die Felder, die in absteigender Reihenfolge sortiert werden sollen, in der Eigenschaft *DescFields* an. Wenn Sie beispielsweise einen Index definieren wollen, der zuerst nach Field1, dann nach Field2 und dann nach Field3 sortiert, ergeben die Werte

```
Field1;Field3
```

in *DescFields* einen Index, der nach Field2 in aufsteigender und nach Field1 und Field3 in absteigender Reihenfolge sortiert.

Die Eigenschaft *CaseInsFields* dient zur Definition von Indizes, die die Groß-/Kleinschreibung bei manchen Feldern berücksichtigen, bei anderen dagegen nicht. Anstatt mit der Option *isCaseInsensitive* alle Indexfelder unter Nichtberücksichtigung der Groß-/Kleinschreibung zu sortieren, geben Sie die Felder, die auf diese Weise sortiert werden sollen, in der Eigenschaft *CaseInsFields* an. Wie *DescFields* erwartet auch *CaseInsFields* eine Liste von Feldnamen, die durch Semikolons voneinander getrennt sind.

Feld- und Indexdefinitionen können zur Entwurfszeit mit dem Kollektions-Editor festgelegt werden. Wählen Sie dazu im Objektinspektor die entsprechende Eigenschaft (*FieldDefs* oder *IndexDefs*) aus, und doppelklicken sie in der Wertespalte, um den Kollektions-Editor aufzurufen. Mit dem Kollektions-Editor können Definitionen hinzugefügt, gelöscht und neu zusammengesetzt werden. Wenn Sie eine Definition im Kollektions-Editor auswählen, können Sie ihre Eigenschaften im Objektinspektor bearbeiten.

Feld- und Indexdefinitionen können auch zur Laufzeit erstellt werden. So erzeugt und aktiviert der folgende Quelltext eine Client-Datenmenge in der *OnCreate*-Ereignisbehandlungsroutine eines Formulars:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with ClientDataSet1 do
begin
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftInteger;
    Name := 'Field1';
  end;
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftString;
    Size := 10;
    Name := 'Field2';
  end;
  with IndexDefs.AddIndexDef do
  begin
    Fields := 'Field1';
    Name := 'IntIndex';
  end;
  CreateDataSet;
end;
end;
```

Datenmengen auf Grundlage bestehender Tabellen erstellen

Bei der Konvertierung einer bestehenden BDE-Anwendung in eine einschichtige Anwendung mit unstrukturierten Dateien können bestehende Tabellen kopiert und aus der IDE heraus als unstrukturierte Dateien auf der Festplatte gespeichert werden. So kopieren Sie eine bestehende Tabelle:

1. Fügen Sie der Anwendung eine *TTable*-Komponente aus der Registerkarte *Datenzugriff* der Komponentenpalette hinzu, und weisen Sie den Eigenschaften *DatabaseName* und *TableName* die entsprechende Datenbank und Tabelle zu. Setzen Sie die Eigenschaft *Active* auf *True*.
2. Fügen Sie eine *TClientDataSet*-Komponente aus der Registerkarte *MIDAS* der Komponentenpalette hinzu.
3. Klicken Sie mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie *Lokale Daten zuweisen*. Wählen Sie im angezeigten Dialogfeld die in Schritt 1 hinzugefügte Tabellenkomponente aus, und klicken Sie auf *OK*.
4. Klicken Sie mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie den Befehl *In Datei speichern*. Dieser Befehl steht nur bei einer leeren Client-Datenmenge zur Verfügung.
5. Wählen Sie im Dialogfeld *Datei speichern* einen Dateinamen, und speichern Sie die Datenbanktabelle in einer unstrukturierten Datei.

Daten laden und speichern

Bei Datenbankanwendungen mit unstrukturierten Dateien existieren alle Tabellenänderungen nur in einem temporären Änderungsprotokoll. Dieses Protokoll wird getrennt von den Daten verwaltet, ist jedoch gegenüber Objekten, die die Client-Datenmenge verwenden, vollständig transparent. Steuerelemente, die in der Client-Datenmenge navigieren und ihre Daten anzeigen, operieren daher auf einer Ansicht der Daten, die auch die Änderungen enthält. Sie können das Änderungsprotokoll mit Hilfe der Methode *MergeChangeLog* mit den Daten der Client-Datenmenge zusammenführen. Weitere Informationen über das Änderungsprotokoll finden Sie unter »Daten bearbeiten« auf Seite 24-5.

Die Änderungen befinden sich auch nach dem Zusammenführen mit den Daten der Client-Datenmenge nur im Speicher. Sie bleiben dort auch, wenn Sie die Client-Datenmenge schließen und wieder öffnen. Erst beim Beendigen der Anwendung werden sie endgültig gelöscht. Um sie dauerhaft zu speichern, müssen sie also auf einen Datenträger geschrieben werden. Dazu dient die Methode *SaveToFile*. Diese Methode benötigt als einzigen Parameter den Namen der zu erstellenden (oder zu überschreibenden) Tabellendatei.

Eine Tabelle, die zuvor mit der Methode *SaveToFile* gespeichert wurde, kann mit der Methode *LoadFromFile* wieder geladen werden. Auch diese Methode erwartet als einzigen Parameter den Namen der Tabellendatei.

Client-Datenmengen werden immer zusammen mit den Metadaten gespeichert, die ihre Datensatzstruktur beschreiben, nicht aber zusammen mit den Indizes. Deshalb ist zur Wiederherstellung der Indizes beim Laden der Daten aus einer Datei entsprechender Quelltext erforderlich. Alternativ dazu können Anwendungen so geschrieben werden, daß sie die benötigten Indizes immer bei Bedarf erstellen.

Wenn immer dieselbe Datei geladen und gespeichert wird, kann anstelle der Methoden *SaveToFile* und *LoadFromFile* die Eigenschaft *FileName* verwendet werden. Wenn die Eigenschaft einen gültigen Dateinamen enthält, werden die Daten beim öffnen

der Client-Datenmenge automatisch aus dieser Datei geladen und beim Schließen in dieser Datei gespeichert.

Das Aktenkoffer-Modell

In diesem Abschnitt wurde schwerpunktmäßig das Erstellen und Verwenden einer Client-Datenmenge in einer einschichtigen Anwendung beschrieben. Ein solches einschichtiges Modell kann zusammen mit einem mehrschichtigen Modell zum sogenannten »Aktenkoffer-Modell« kombiniert werden.

Hinweis Das Aktenkoffer-Modell wird zuweilen auch als unverbundenes Modell oder Mobil-Computing bezeichnet.

Betrieben am Standort eines Unternehmens, sieht eine Anwendung, die auf dem Aktenkoffer-Modell basiert, aus wie eine mehrschichtige Anwendung: Ein Benutzer startet auf einem Computer eine Client-Anwendung und stellt über ein Netzwerk eine Verbindung zu einem Anwendungsserver auf einem Remote-Computer her. Der Client-Computer fordert Daten vom Anwendungsserver an und sendet Aktualisierungen zurück. Die Aktualisierungen werden durch den Anwendungsserver in eine Datenbank geschrieben, auf die vermutlich auch andere Clients einer Organisation zugreifen.

Ein Beispiel für dieses Modell ist eine Firmendatenbank mit wichtigen Kontaktdaten, die von Vertretern unterwegs verwendet und aktualisiert werden können. Die Vertreter laden sich einige oder alle Datenbankinformationen herunter, bearbeiten diese während ihrer Reisen auf ihren Laptop-Computern und aktualisieren die Daten an bestehenden oder neuen Kundenstandorten. Nach ihrer Rückkehr werden die Änderungen in der Firmendatenbank gespeichert und können von allen Mitarbeitern verwendet werden. Diese Fähigkeit der Offline-Datenbearbeitung und der späteren Online-Aktualisierung wird als Aktenkoffer-Modell bezeichnet.

Das Modell bedient sich der Fähigkeit von Client-Datenmengenkomponenten zum Lesen und Schreiben von Daten in unstrukturierten Dateien. Auf dieser Grundlage können Client-Anwendungen erstellt werden, die sowohl online mit einem Anwendungsserver als auch offline als temporäre einschichtige Anwendung fungieren.

So implementieren Sie das Aktenkoffer-Modell:

1. Erstellen Sie eine mehrschichtige Server-Anwendung. Informationen dazu finden Sie unter »Anwendungsserver erstellen« auf Seite 14-13.
2. Erstellen Sie als Client-Anwendung eine Datenbankanwendung mit unstrukturierten Dateien. Fügen Sie der Anwendung eine Verbindungskomponente hinzu, und weisen Sie diese der Eigenschaft *RemoteServer* der Client-Datenmengen zu. Dadurch können die Datenmengen mit dem in Schritt 1 erstellten Anwendungsserver kommunizieren. Weitere Informationen über Verbindungskomponenten finden Sie unter »Verbindung zum Anwendungsserver einrichten« auf Seite 14-21.
3. Versuchen Sie in der Client-Anwendung eine Verbindung zum Anwendungsserver herzustellen. Falls die Verbindung nicht funktioniert, geben Sie dem Benutzer die Möglichkeit, eine Datei anzugeben, aus der eine lokale Kopie der Daten eingelesen werden kann.

4 Fügen Sie der Client-Anwendung den Quelltext hinzu, der die Aktualisierungen auf dem Anwendungsserver durchführt. Weitere Informationen über das Senden von Aktualisierungen einer Client-Anwendung an einen Anwendungsserver finden Sie unter »Daten aktualisieren« auf Seite 24-25.

Skalierung auf eine dreischichtige Anwendung

In einer zweischichtigen Client/Server-Anwendung ist die Anwendung ein Client, der direkt mit einem Datenbankserver kommuniziert. Diese Anwendung besteht gewissermaßen aus zwei Teilen: einer Datenbankverbindung und einer Benutzeroberfläche. Zur Umwandlung einer zweischichtigen Client/Server-Anwendung in eine mehrschichtige Anwendung sind folgende Schritte erforderlich:

- Unterteilen Sie die bestehende Anwendung in einen Anwendungsserver (zur Realisierung der Datenbankverbindung) und in eine Client-Anwendung (die die Benutzeroberfläche enthält).
- Fügen Sie zwischen dem Client und dem Anwendungsserver eine Schnittstelle ein.

Nun gehen Sie am einfachsten folgendermaßen vor:

1 Erstellen Sie für den Anwendungsserver ein neues Projekt, und beginnen Sie mit einem Remote-Datenmodul. Informationen hierzu finden Sie unter »Anwendungsserver erstellen« auf Seite 14-13.

2 Duplizieren Sie die für die Datenbankverbindung wichtigen Teile der früheren zweischichtigen Anwendung, und fügen Sie für jede Datenmenge eine Provider-Komponente ein, die als Datenleitung zwischen dem Anwendungsserver und dem Client fungiert. Informationen über die Arbeit mit Provider-Komponenten finden Sie in Kapitel 15, »Provider-Komponenten«.

3 Kopieren Sie das bestehende zweischichtige Projekt, entfernen Sie seine direkten Datenbankverbindungen, und fügen Sie eine geeignete Verbindungskomponente hinzu. Weitere Informationen über die Erstellung und Verwendung von Verbindungskomponenten finden Sie unter »Verbindung zum Anwendungsserver einrichten« auf Seite 14-21.

4 Ersetzen Sie jede BDE- oder ADO-bezogene Datenmengenkomponente im Originalprojekt durch eine Client-Datenmenge. Allgemeine Informationen über Client-Datenmengen finden Sie in Kapitel 24, »Client-Datenmengen«.

5 Fügen Sie der Client-Anwendung Quelltext hinzu, der die Aktualisierungen auf dem Anwendungsserver durchführt. Weitere Informationen über das Senden von Aktualisierungen einer Client-Anwendung an einen Anwendungsserver finden Sie unter »Daten aktualisieren« auf Seite 24-25.

6 Verschieben Sie die Datenmengenkomponenten in die Datenmodule des Anwendungsservers. Weisen Sie der Eigenschaft *DataSe*t der einzelnen Provider die entsprechende Datenmenge zu. Weitere Informationen über das Verknüpfen einer Datenmenge mit einer Provider-Komponente finden Sie in Kapitel 15, »Provider-Komponenten«.

Kapitel

14

Mehrschichtige Anwendungen erstellen

Dieses Kapitel beschreibt die Erstellung mehrschichtiger Client/Server-Datenbankanwendungen. Eine mehrschichtige Client/Server-Anwendung besteht aus logischen Einheiten, die auf unterschiedlichen Rechnern ausgeführt werden, aber zusammenarbeiten. Mehrschichtige Anwendungen greifen gemeinsam auf Daten zu und kommunizieren über ein lokales Netzwerk oder auch das Internet miteinander. Sie bieten viele Vorteile gegenüber herkömmlichen Anwendungen, beispielsweise eine zentralisierte Anwendungslogik und schlanke Client-Anwendungen.

In der einfachsten Form, die auch als »dreischichtiges Modell« bezeichnet wird, besteht eine mehrschichtige Anwendung aus den folgenden Dritteln:

- **Client-Anwendung:** Stellt die Benutzeroberfläche auf dem Rechner des Benutzers bereit.
- **Anwendungsserver:** Befindet sich an einer zentralen Position im Netzwerk, auf die alle Clients zugreifen können. Er stellt die allgemeinen Datendienste bereit.
- **Remote-Datenbankserver:** Stellt das relationale Datenbankverwaltungssystem (RDBMS) bereit.

In diesem dreischichtigen Modell verwaltet der Anwendungsserver den Datenfluß zwischen den Clients und dem Remote-Datenbankserver. Er wird deshalb auch als »Daten-Broker« bezeichnet. Mit Delphi erstellen Sie normalerweise nur den Anwendungsserver und die Clients. Es ist jedoch durchaus möglich, auch eigene Datenbank-Backends zu erstellen.

In komplexeren mehrschichtigen Anwendungen können sich zusätzliche Dienste zwischen den Clients und dem Remote-Datenbankserver befinden. Dabei kann es sich beispielsweise um einen Broker für Sicherheitsdienste handeln, der sichere Internet-Transaktionen gewährleistet, oder um Brückendienste für den gemeinsamen Zugriff auf Datenbankdaten auf Plattformen, die nicht direkt von Delphi unterstützt werden.

Die Unterstützung mehrschichtiger Anwendungen von Delphi basiert auf MIDAS (Multi-tier Distributed Application Services Suite). Dieses Kapitel befaßt sich in erster Linie mit der Erstellung einer dreischichtigen Datenbankanwendung unter Verwendung der MIDAS-Technologie. Nachdem Sie das Erstellen und Verwalten dreischichtiger Anwendungen erlernt haben, können Sie nach Maßgabe der jeweiligen Anforderungen weitere Dienstschichten hinzufügen.

Vorteile des mehrschichtigen Datenbankmodells

Das mehrschichtige Datenbankmodell teilt eine Datenbankanwendung in mehrere logische Komponenten auf. Die Client-Anwendung ist nur für die Anzeige der Daten und die Verarbeitung der Benutzereingaben erforderlich. Sie besitzt und benötigt im Idealfall keinerlei Informationen zur Datenspeicherung und -verwaltung. Der Anwendungsserver (mittlere Schicht) koordiniert und verarbeitet Anforderungen und Aktualisierungen von mehreren Clients. Er ist für die Definition von Datenmengen und die Interaktion mit dem Remote-Datenbankserver verantwortlich.

Nachstehend werden einige der Vorteile des mehrschichtigen Modells zusammengefaßt:

- **Kapselung der Anwendungslogik (Business Rules) in einer freigegebenen Mittelschicht.** Die verschiedenen Client-Anwendungen greifen alle auf dieselbe Mittelschicht zu. Dadurch können Redundanzen vermieden und Wartungskosten reduziert werden, da die Business Rules nicht für jede neue Client-Anwendung erneut implementiert werden müssen.
- **Schlanke Client-Anwendungen (Thin-Client-Anwendungen).** Client-Anwendungen können extrem klein gehalten werden, indem der Großteil der Verarbeitung an die Mittelschicht delegiert wird. Daraus ergibt sich der weitere Vorteil einer vereinfachten Weitergabe von Anwendungen, da keine Installation, Konfiguration und Wartung der Software zur Herstellung der Datenbankkonnektivität (beispielsweise die Borland Database Engine) erforderlich ist. Thin-Client-Anwendungen eignen sich ideal für die Weitergabe über das Internet.
- **Verteilte Datenverarbeitung.** Die Verteilung verschiedener Anwendungsoperationen auf mehrere Rechner kann die Leistung durch die bessere Lastverteilung erhöhen. Gleichzeitig können redundante Systeme aktiviert werden, wenn beispielsweise ein Server heruntergefahren werden muß.
- **Mehr Schutzmöglichkeiten.** Sie können sensitive Funktionen in Schichten isolieren, die über verschiedene Zugriffsbeschränkungen verfügen. Dies ermöglicht die Einrichtung flexibler Schutzstufen, die problemlos konfiguriert werden können. Mittelschichten können die Zugriffspunkte auf sensitive Daten überwachen. Wenn Sie HTTP, CORBA oder MTS einsetzen, können Sie auch die von diesen Technologien unterstützten Sicherheitsmodelle nutzen.

Grundlagen der MIDAS-Technologie

MIDAS stellt den Mechanismus bereit, über den Client-Anwendungen und Anwendungsserver Datenbankinformationen austauschen. Der Einsatz von MIDAS setzt MIDAS.DLL voraus. Diese Bibliothek wird in Client- und in Server-Anwendungen zur Verwaltung von Datenmengen benötigt, die in Datenpaketen gespeichert sind. Bei der Erstellung von MIDAS-Anwendungen ist unter Umständen auch der SQL-Explorer erforderlich, der die Datenbankverwaltung erleichtert und das Importieren von Server-Beschränkungen in das Daten-Dictionary ermöglicht, damit diese Beschränkungen in jeder Schicht einer mehrschichtigen Anwendung überprüft werden können.

Hinweis Sie müssen Server-Lizenzen erwerben, damit Sie Ihre MIDAS-Anwendungen weitergeben dürfen.

Mehrschichtige MIDAS-Anwendungen verwenden die Komponenten der Registerkarte *MIDAS* der Komponentenpalette sowie ein Remote-Datenmodul, das mit einem Experten in der Registerkarte *Mehrschichtig* des Dialogfeldes *Objektgalerie* erstellt wird. Diese Komponenten werden in der folgenden Tabelle beschrieben.

Tabelle 14.1 MIDAS-Komponenten

Komponente	Beschreibung
Remote-Datenmodule	Spezialisierte Datenmodule, die als COM-Automatisierungsserver oder CORBA-Server dienen, um Client-Anwendungen Zugriff auf die enthaltenen Provider zu ermöglichen. Wird auf dem Anwendungsserver eingesetzt.
Provider-Komponenten	Ein Daten-Broker, der Daten durch Erstellung von Datenpaketen bereitstellt und Client-Aktualisierungen verarbeitet. Wird auf dem Anwendungsserver eingesetzt.
Client-Datenmengenkomponenten	Eine spezialisierte Datenmenge, die MIDAS.DLL einsetzt, um die in Datenpaketen gespeicherten Daten zu verwalten.
Verbindungskomponenten	Eine Komponentenfamilie, die für die Server-Suche, die Einrichtung von Verbindungen und die Bereitstellung der Schnittstelle *IAppServer* für Client-Datenmengen verantwortlich ist. Jede Verbindungskomponente wurde für ein bestimmtes Verbindungsprotokoll entwickelt.

Struktur einer mehrschichtigen MIDAS-Anwendung

Die folgenden Schritte illustrieren den normalen Ereignisfluß in einer mehrschichtigen MIDAS-Anwendung:

1. Ein Benutzer startet die Client-Anwendung. Der Client richtet eine Verbindung zum Anwendungsserver ein (der Anwendungsserver kann zur Laufzeit oder bereits bei der Programmerstellung eingerichtet werden). Wenn der Anwendungsserver noch nicht läuft, wird er jetzt gestartet. Der Client erhält eine *IAppServer*-Schnittstelle vom Anwendungsserver.
2. Der Client fordert Daten vom Anwendungsserver an. Ein Client kann alle Daten oder einzelne Datenbruchstücke im Verlauf der Sitzung anfordern (Bedarfsanforderung).

3. Der Anwendungsserver ruft die Daten ab (zuvor wird gegebenenfalls eine Datenbankverbindung eingerichtet), ordnet sie für den Client in einem Paket an und sendet dann dieses Paket an den Client. In die Metadaten des Datenpakets können weitere Informationen eingefügt werden (beispielsweise zu Datenbeschränkungen der Datenbank). Dieser Prozeß der Anordnung von Daten in Datenpaketen wird als »Providing« bezeichnet.
4. Der Client dekodiert das Datenpaket und zeigt dem Benutzer die Daten an.
5. Durch Benutzereingaben in der Client-Anwendung werden die Daten aktualisiert (Datensätze werden hinzugefügt, gelöscht oder geändert). Diese Änderungen werden vom Client in einem Änderungsprotokoll gespeichert.
6. Gelegentlich – normalerweise als Reaktion auf eine bestimmte Aktion des Benutzers – sendet der Client die Aktualisierungen an den Anwendungsserver. Dazu wird das Änderungsprotokoll als Datenpaket an den Server gesendet.
7. Der Anwendungsserver dekodiert das Paket und trägt Aktualisierungen im Rahmen einer Transaktion ein. Wenn ein Datensatz nicht auf dem Server eingetragen werden kann (weil beispielsweise eine andere Anwendung den Datensatz geändert hat, nachdem dieser vom Client angefordert und bevor er vom Client aktualisiert wurde), gleicht der Anwendungsserver die Änderungen des Clients entweder mit den aktuellen Daten ab oder speichert die Datensätze, die nicht eingetragen werden konnten. Dieser Prozeß – das Eintragen von Datensätzen und das Zwischenspeichern von problematischen Datensätzen – wird auch als »Auflösen« bezeichnet.
8. Nachdem der Anwendungsserver den Auflösungsprozeß abgeschlossen hat, werden die nicht eingetragenen Datensätze zur weiteren Bearbeitung wieder an den Client gesendet.
9. Der Client gleicht die nicht aufgelösten Datensätze ab. Es gibt mehrere Methoden, diesen Abgleich durchzuführen. Normalerweise versucht der Client, die Bedingungen zu beseitigen, die das Eintragen des Datensatzes verhinderten. Gegebenenfalls werden die Änderungen verworfen. Kann die Fehlerbedingung beseitigt werden, sendet der Client die Aktualisierungen erneut an den Anwendungsserver.
10. Der Client ruft die aktuelle Version der Daten vom Server ab.

Die Struktur der Client-Anwendung

Für den Endbenutzer ergeben sich keine Unterschiede zwischen der Client-Anwendung einer mehrschichtigen Anwendung und einer klassischen zweischichtigen Anwendung mit zwischengespeicherten Aktualisierungen. Strukturell ähnelt die Client-Anwendung einer einschichtigen Anwendung mit unstrukturierten Dateien. Die Benutzeroperationen erfolgen mit datensensitiven Standardsteuerelementen, die Daten aus einer Client-Datenmengenkomponente anzeigen. Ausführliche Informationen zu den Eigenschaften, Ereignissen und Methoden der Client-Datenmengen finden Sie in Kapitel 24, »Client-Datenmengen«.

Im Unterschied zu einschichtigen Anwendungen ruft die Client-Datenmenge einer mehrschichtigen Anwendung die Daten über die *IAppServer*-Schnittstelle des Anwendungsservers ab. Die Datenmenge leitet auch Aktualisierungen über diese Schnittstelle an den Anwendungsserver weiter. Informationen zur Schnittstelle *IAppServer* finden Sie unter »Schnittstelle IAppServer verwenden« auf Seite 14-8. Der Client erhält diese Schnittstelle von einer Verbindungskomponente.

Die Verbindungskomponente richtet die Verbindung zum Anwendungsserver ein. Für die verschiedenen Kommunikationsprotokolle sind jeweils eigene Verbindungskomponenten verfügbar. Diese Verbindungskomponenten werden in der folgenden Tabelle zusammengefaßt:

Tabelle 14.2 Verbindungskomponenten

Komponente	Protokoll
TDCOMConnection	DCOM
TSocketConnection	Windows Sockets (TCP/IP)
TWebConnection	HTTP
TOLEnterpriseConnection	OLEnterprise (RPCs)
TCorbaConnection	CORBA (IIOP)

Hinweis Die beiden Verbindungskomponenten *TRemoteServer* und *TMIDASConnection* wurden zur Gewährleistung der Abwärtskompatibilität bereitgestellt.

Weitere Informationen zur Verwendung von Verbindungskomponenten finden Sie unter »Verbindung zum Anwendungsserver einrichten« auf Seite 14-21.

Die Struktur des Anwendungsservers

Der Anwendungsserver enthält ein Remote-Datenmodul, das eine *IAppServer*-Schnittstelle bereitstellt. Über diese Schnittstelle kommunizieren die Client-Anwendungen mit den Daten-Providern. Es gibt drei Typen von Remote-Datenmodulen:

- **TRemoteDataModule:** Dies ist ein Automatisierungsserver mit zwei Schnittstellen. Verwenden Sie dieses Remote-Datenmodul, wenn Clients die Verbindung zum Anwendungsserver über DCOM, HTTP, Sockets oder OLEnterprise einrichten und der Anwendungsserver nicht mit MTS installiert werden soll.
- **TMTSDataModule:** Dies ist ein Automatisierungsserver mit zwei Schnittstellen. Verwenden Sie dieses Remote-Datenmodul, wenn Sie den Anwendungsserver als ActiveX-Bibliothek (DLL) erstellen, die mit MTS installiert wird. Sie können MTS-Remote-Datenmodule mit DCOM, HTTP, Sockets oder OLEnterprise einsetzen.
- **TCorbaDataModule:** Dies ist ein CORBA-Server. Verwenden Sie dieses Remote-Datenmodul, um Daten für CORBA-Clients bereitzustellen.

Sie können in das Remote-Datenmodul wie in jedes andere Datenmodul nichtvisuelle Komponenten einfügen. Zusätzlich enthält das Remote-Datenmodul eine Datenmengen-Provider-Komponente für jede Datenmenge, die der Anwendungsserver für Client-Anwendungen verfügbar macht. Ein Datenmengen-Provider ermöglicht folgende Aktionen:

- Datenanforderungen können vom Client empfangen, die angeforderten Daten vom Datenbankserver abgerufen, für die Übertragung in einem Paket zusammengefaßt und dann an die Datenmenge des Clients gesendet werden. Dieser Vorgang wird als »Providing« oder »Bereitstellen« bezeichnet.
- Aktualisierte Daten werden vom Client empfangen und in die Datenbank (die Quelldatenmenge) eingetragen. Fehlgeschlagene Aktualisierungen werden protokolliert und zur weiteren Bearbeitung an den Client zurückgesendet. Dieser Prozeß wird als »Auflösen« bezeichnet.

Der Provider verwendet häufig BDE- oder ADO-konforme Datenmengen, wie sie auch in zweischichtigen Anwendungen eingesetzt werden. Entsprechend können Sie auch nach Bedarf Datenbank- und Sitzungskomponenten (wie in einer zweischichtigen BDE-basierten Anwendung) bzw. ADO-Verbindungskomponenten (wie in einer zweischichtigen ADO-basierten Anwendung) hinzufügen.

Hinweis ADO-Verbindungskomponenten entsprechen den Datenbankkomponenten in BDE-basierten Anwendungen und dürfen nicht mit den von Client-Anwendungen genutzten Verbindungskomponenten in einer mehrschichtigen Anwendung verwechselt werden.

Weitere Informationen zu zweischichtigen Anwendungen finden Sie in Kapitel 13, »Ein- und zweischichtige Anwendungen erstellen«.

Bei einem MTS-konformen Anwendungsserver enthält das MTS-Datenmodul Ereignisse für den aktivierten und deaktivierten Zustand des Anwendungsservers. Dies ermöglicht den automatischen Aufbau von Datenbankverbindungen bei der Aktivierung des Servers und den automatischen Verbindungsabbau bei seiner Deaktivierung.

MTS verwenden

Der Einsatz von MTS erweitert die Funktionalität des Remote-Datenmoduls:

- **MTS-Sicherheit:** MTS ermöglicht die funktionsbezogene Sicherheit im Anwendungsserver. Clients werden Funktionen zugewiesen. Diese bestimmen, wie auf die Schnittstelle des MTS-Datenmoduls zugegriffen werden kann. Das MTS-Datenmodul implementiert die Methode *IsCallerInRole* , mit der Sie die Funktion des aktuell verbundenen Clients ermitteln und in Abhängigkeit von dieser Funktion bestimmte Anwendungsmerkmale bereitstellen können. Weitere Informationen zur MTS-Sicherheit finden Sie unter »Rollenbasierte Sicherheit« auf Seite 51-13.
- **Datenbank-Handle-Verwaltung:** MTS-Datenmodule verwalten Datenbankverbindungen automatisch in einem Pool (Ressourcen-Pooling). Gibt ein Client seine Datenbankverbindung frei, kann diese von einem anderen Client genutzt werden. Der Datenverkehr im Netzwerk wird reduziert, da die Mittelschicht keine Abmeldung und anschließende Neuanmeldung beim Datenbankserver durchführen muß. Wenn die Verwaltung der Datenbank-Handles genutzt wird, sollte die Datenbankkomponente in der Eigenschaft *KeepConnection* den Wert *False* enthalten, damit die Anwendung die Freigabe von Verbindungen optimiert.
- **MTS-Transaktionen:** Wenn Sie MTS einsetzen, können Sie eigene MTS-Transaktionen in den Anwendungsserver integrieren, um die Transaktionsunterstützung

zu erweitern. MTS-Transaktionen können sich über mehrere Datenbanken erstrekken oder Funktionen enthalten, die sich nicht auf Datenbanken beziehen. Weitere Informationen zu MTS-Transaktionen finden Sie unter »Transaktionen in mehrschichtigen Anwendungen verwalten« auf Seite 14-28.

- **Bedarfsaktivierung und schnellstmögliche Deaktivierung:** Sie können einen MTS-Server schreiben, damit die Instanzen von Remote-Datenmodulen nach Bedarf aktiviert und deaktiviert werden können. Beim Einsatz der Bedarfsaktivierung und der schnellstmöglichen Deaktivierung wird das Remote-Datenmodul nur instantiiert, wenn es zur Verarbeitung von Client-Anforderungen benötigt wird. Dieses Vorgehen verhindert das Belegen von Datenbank-Handles, wenn diese nicht benötigt werden.

  Die Bedarfsaktivierung und schnellstmögliche Deaktivierung stellt einen Kompromiß zwischen der Vermittlung aller Clients über eine einzelne Instanz des Remote-Datenmoduls und dem Erstellen einer separaten Instanz für jede Client-Verbindung dar. Liegt nur eine Instanz des Remote-Datenmoduls vor, muß der Anwendungsserver alle Datenbankaufrufe über eine einzelne Datenbankverbindung durchführen. Dieser Engpaß kann die Leistung stark beeinträchtigen, wenn viele Clients auf die Datenbank zugreifen. Liegen dagegen mehrere Instanzen des Remote-Datenmoduls vor, kann jede Instanz eine eigene Datenbankverbindung verwalten. Eine serielle Durchführung der Datenbankzugriffe wird also vermieden. Durch dieses Vorgehen kann es jedoch zu Ressourcenmangel kommen, da andere Clients die Datenbankverbindungen nicht nutzen können, während diese den Remote-Datenmodulen von Clients zugeordnet sind.

Um die Vorteile von Transaktionen, Bedarfsaktivierung und schnellstmöglicher Deaktivierung zu nutzen, dürfen die Instanzen des Remote-Datenmoduls keinen Status besitzen. Es ist aber eine zusätzliche Unterstützung erforderlich, wenn der Client auf Statusinformationen angewiesen ist (beispielsweise muß der Client bei der Durchführung inkrementeller Abrufe Informationen zum aktuellen Datensatz übergeben). Weitere Einzelheiten über Statusinformationen und Remote-Datenmodule in mehrschichtigen Anwendungen finden Sie im Abschnitt »Statusinformationen in Remote-Datenmodulen unterstützen« auf Seite 14-30.

Alle automatisch generierten Aufrufe eines MTS-Datenmoduls sind transaktionsbezogen (es wird davon ausgegangen, daß das MTS-Datenmodul nach dem Ende des Aufrufs deaktiviert werden kann und alle aktuellen Transaktionen eingetragen oder rückgängig gemacht werden können). Sie können ein MTS-Datenmodul schreiben, das vom Vorhandensein von Statusinformationen abhängig ist, indem Sie die Eigenschaft *AutoComplete* ~ mit *False* belegen. Die Vorteile von Transaktionen, Bedarfsaktivierung und schnellstmöglicher Deaktivierung können dann aber nicht genutzt werden.

Achtung Beim Einsatz von MTS sollten Datenbankverbindungen erst nach Aktivierung des Remote-Datenmoduls geöffnet werden. Stellen Sie bei der Entwicklung der Anwendung sicher, daß vor dem Starten der Anwendung keine Datenmengen aktiv sind und keine Verbindung zur Datenbank besteht. In die Anwendung müssen Sie Quelltext zum Öffnen der Datenbankverbindung beim Aktivieren des Datenmoduls und zum Schließen dieser Verbindungen beim Deaktivieren des Datenmoduls einfügen.

Remote-Datenmodule verwalten

Durch die Objektverwaltung können Sie einige Vorteile von MTS auch dann nutzen, wenn Sie DCOM nicht verwenden. Die Objektverwaltung ermöglicht es, die Anzahl der erstellten Remote-Datenmodulinstanzen zu beschränken. Dadurch wird die Zahl der zu verwaltenden Datenbankverbindungen und der vom Remote-Datenmodul benötigten Ressourcen verringert.

Der Server übergibt empfangene Client-Anforderungen an das erste verfügbare Remote-Datenmodul im Pool. Wenn gerade kein Remote-Datenmodul verfügbar ist, wird ein neues erstellt (bis zu der von Ihnen festgelegten Maximalzahl). Dies stellt einen Kompromiß zwischen der Vermittlung aller Clients über eine einzelne Instanz des Remote-Datenmoduls (führt eventuell zu einem Leistungsengpaß) und dem Erstellen einer separaten Instanz für jede Client-Verbindung dar (erfordert sehr viele Ressourcen).

Um einen Ressourcenmangel zu verhindern, werden Remote-Datenmodule, die längere Zeit keine Client-Anforderungen empfangen haben, automatisch freigegeben.

Da eine Instanz eines Remote-Datenmoduls Anforderungen mehrerer Clients verarbeiten kann, darf sie keinen Status besitzen. Im Abschnitt »Statusinformationen in Remote-Datenmodulen unterstützen« auf Seite 14-30 wird beschrieben, wie sichergestellt werden kann, daß ein Remote-Datenmodul statuslos ist.

Um die Vorteile der Objektverwaltung nutzen zu können, müssen die Remote-Datenmodule die Methode *UpdateRegistry* überschreiben. In der überschriebenen Methode kann *RegisterPooled* aufgerufen werden, um das Remote-Datenmodul zu registrieren, und *UnregisterPooled* , um es aus der Registrierung zu entfernen.

Die Objektverwaltung kann nur genutzt werden, wenn die Verbindung mittels HTTP eingerichtet wird.

Schnittstelle IAppServer verwenden

Remote-Datenmodule auf dem Anwendungsserver unterstützen die Schnittstelle *IAppServer* . Die Verbindungskomponenten der Client-Anwendungen greifen auf diese Schnittstelle zu, um Verbindungen einzurichten.

IAppServer stellt das Bindeglied zwischen Client-Anwendungen und den Provider-Komponenten auf dem Anwendungsserver bereit. Die meisten Client-Anwendungen greifen nicht direkt auf die Schnittstelle *IAppServer* zu. Statt dessen erfolgt der Zugriff indirekt über die Eigenschaften und Methoden der Client-Datenmenge. Gegebenenfalls können Sie über die Eigenschaft *AppServer* der Client-Datenmenge direkt auf die Schnittstelle *IAppServer* zugreifen.

Tabelle 14.3 enthält die Methoden der Schnittstelle *IAppServer* sowie die entsprechenden Methoden und Ereignisse der Provider-Komponente und der Client-Datenmenge. Die *IAppServer*-Methoden übernehmen einen Provider-Parameter, mit dem angegeben wird, welche Provider-Komponente auf dem Anwendungsserver Daten bereitstellen und Aktualisierungen verarbeiten soll. Außerdem kann an die meisten Methoden ein OleVariant-Parameter namens *OwnerData* übergeben werden, der es Client-Anwendung und Anwendungsserver ermöglicht, benutzerdefinierte Informationen auszutauschen. *OwnerData* wird nicht standardmäßig verwendet, kann aber

an alle Ereignisbehandlungsroutinen übergeben. Sie können somit Quelltext schreiben, der es dem Anwendungsserver ermöglicht, vor und nach jedem Client-Aufruf bestimmte Anpassungen vorzunehmen.

Tabelle 14.3 Elementfunktionen der Schnittstelle *IAppServer*

IAppServer	Provider-Komponente	TClientDataSet
Methode AS_ApplyUpdates	Methode ApplyUpdates, Ereignis BeforeApplyUpdates, Ereignis AfterApplyUpdates	Methode ApplyUpdates, Ereignis BeforeApplyUpdates, Ereignis AfterApplyUpdates.
Methode AS_DataRequest 1	Methode DataRequest, Ereignis OnDataRequest	Methode DataRequest.
Methode AS_Execute	Methode Execute, Ereignis BeforeExecute, Ereignis AfterExecute	Methode Execute, Ereignis BeforeExecute, Ereignis AfterExecute.
Methode AS_GetParams	Methode GetParams, Ereignis BeforeGetParams, Ereignis AfterGetParams	Methode FetchParams, Ereignis BeforeGetparams, Ereignis AfterGetParams.
Methode AS_GetProviderNames	Dient der Identifizierung aller verfügbaren Provider.	Dient zur Entwurfszeit der Erstellung einer Liste für die Eigenschaft ProviderName.
Methode AS_GetRecords	Methode GetRecords, Ereignis BeforeGetRecords, Ereignis AfterGetRecords	Methode GetNextPacket, Eigenschaft Data, Ereignis BeforeGetRecords, Ereignis AfterGetRecords.
Methode AS_RowRequest	Methode RowRequest, Ereignis BeforeRowRequest, Ereignis AfterRowRequest	Methode FetchBlobs, Methode FetchDetails, Methode RefreshRecord, Ereignis BeforeRowRequest, Ereignis AfterRowRequest.

Verbindungsprotokoll wählen

Jedes Kommunikationsprotokoll zur Einrichtung von Verbindungen zwischen Client-Anwendungen und dem Anwendungsserver besitzt spezielle Vorteile. Bevor Sie ein Protokoll auswählen, sollten Sie die voraussichtliche Anzahl der Clients bestimmen, die Weitergabeart der Anwendung berücksichtigen und geplante Weiterentwicklungen beachten.

DCOM-Verbindungen einsetzen

DCOM stellt die direkteste Kommunikationsmöglichkeit dar. Auf dem Server werden keine zusätzlichen Laufzeitanwendungen benötigt. Da jedoch DCOM nicht im Lieferumfang von Windows 95 enthalten ist, ist DCOM wahrscheinlich nicht auf allen Client-Rechnern installiert.

DCOM stellt den einzigen Ansatz dar, der den Einsatz der MTS-Sicherheitsmerkmale ermöglicht. Diese basiert auf Funktionen, die den Aufrufern von MTS-Objekten zu-

gewiesen werden. Beim Aufrufen von MTS-Objekten über DCOM informiert DCOM MTS über die Client-Anwendung, die den Aufruf generiert hat. MTS kann anschließend die Funktion des Aufrufers genau bestimmen. Bei Verwendung anderer Protokolle wird unabhängig vom Anwendungsserver eine ausführbare Laufzeitdatei benötigt, die Client-Aufrufe empfängt. Diese ausführbare Laufzeitdatei führt die COM-Aufrufe an den Anwendungsserver für den Client durch. MTS kann die Funktionen nicht unterschiedlichen Clients zuordnen, da die Aufrufe an den Anwendungsserver von der ausführbaren Laufzeitdatei stammen. Weitere Informationen zur MTS-Sicherheit finden Sie im Abschnitt »Rollenbasierte Sicherheit« auf Seite 51-13.

Socket-Verbindungen einsetzen

Mit TCP/IP Sockets können Sie extrem kleine Clients erstellen. Wenn Sie beispielsweise eine Web-basierte MIDAS-Client-Anwendung erstellen, ist nicht sichergestellt, daß die Client-Systeme DCOM unterstützen. Sockets stellen eine Grundlage dar, die sicher zur Einrichtung der Verbindung zum Anwendungsserver verfügbar ist. Weitere Informationen zu Sockets finden Sie in Kapitel 30, »Arbeiten mit Sockets«.

Statt wie DCOM das Remote-Datenmodul direkt vom Client zu instantiieren, verwenden Sockets eine separate Anwendung auf dem Server (SCKTSRVR.EXE), die Client-Anforderungen entgegennimmt und das Remote-Datenmodul mit COM instantiiert. Die Verbindungskomponente des Clients und SCKTSRVR.EXE auf dem Server sind für die Sequenzierung (Marshaling) von *IAppServe*r-Aufrufen verantwortlich.

Hinweis SCKTSRVR.EXE kann als NT-Dienstanwendung ausgeführt werden. Um sie im Dienste-Manager zu registrieren, starten Sie SCKTSRVR.EXE mit der Kommandozeilenoption -install. Mit der Option -uninstall können Sie die Registrierung von SCKTSRVR.EXE aufheben.

Bevor eine Socket-Verbindung genutzt werden kann, muß der Anwendungsserver ihre Verfügbarkeit für Clients registrieren, die Socket-Verbindungen verwenden. Standardmäßig registrieren sich alle neuen Remote-Datenmodule selbst durch einen Aufruf von *EnableSocketTransport* in der Methode *UpdateRegistry*. Wenn Sie Socket-Verbindungen zum Anwendungsserver verhindern möchten, entfernen Sie diesen Aufruf.

Hinweis Ältere Server nehmen diese Registrierung nicht vor. Sie haben deshalb die Möglichkeit, die Überprüfung auf Registrierung des Anwendungsservers auszuschalten. Deaktivieren Sie dazu den Menübefehl *Verbindungen / Nur registrierte Objekte* für die Datei SCKTSRVR.EXE.

Wenn Sie Sockets verwenden, existiert auf dem Server kein Schutz vor Client-Systemen, die vor der Freigabe einer Referenz auf eine Schnittstelle des Anwendungsservers fehlschlagen. Dies reduziert zwar im Vergleich zu DCOM den Umfang der über das Netzwerk gesendeten Botschaften (DCOM sendet regelmäßig Botschaften, damit die Verbindung aufrechterhalten wird), kann aber dazu führen, daß ein Anwendungsserver seine Ressourcen nicht freigeben kann, da die entsprechenden Informationen vom Client fehlen.

Web-Verbindungen einsetzen

HTTP ermöglicht die Erstellung von Clients, die mit einem durch eine »Brandmauer« geschützten Anwendungsserver kommunizieren können. Aufgrund des kontrollierten Zugriffs auf interne Anwendungen mittels HTTP-Botschaften ist ein sicherer Betrieb von Client-Anwendungen auf unterschiedlichsten Systemen möglich. Wie Sokket-Verbindungen stellen auch HTTP-Botschaften eine Grundlage dar, die immer zur Einrichtung der Verbindung zum Anwendungsserver verfügbar ist. Weitere Informationen über HTTP-Botschaften finden Sie in Kapitel 29, »Internet-Server-Anwendungen«.

Statt wie DCOM das Remote-Datenmodul direkt vom Client zu instantiieren, verwenden HTTP-basierte Verbindungen eine Web-Server-Anwendung auf dem Server (HTTPSRVR.DLL), die Client-Anforderungen entgegennimmt und das Remote-Datenmodul mit COM instantiiert. HTTP-basierte Verbindungen werden deshalb auch Web-Verbindungen genannt. Die Verbindungskomponente des Clients und HTTPSRVR.DLL auf dem Server sind für die Verwaltung von *IAppServer*-Aufrufen verantwortlich.

Für Web-Verbindungen können die von WININET.DLL bereitgestellten SSL-Sicherheitsfunktionen eingesetzt werden. WININET.DLL ist eine Bibliothek mit Internet-Dienstprogrammen, die auf dem Client-System ausgeführt werden. Nachdem Sie den Web-Server auf dem Server-System für die Authentifizierung konfiguriert haben, können Sie den Benutzernamen und das Kennwort mit den entsprechenden Eigenschaften der Web-Verbindungskomponente angeben.

Als zusätzliche Sicherheitsmaßnahme muß der Anwendungsserver seine Verfügbarkeit für Clients registrieren, die eine Web-Verbindung benutzen. Standardmäßig registrieren sich alle neuen Remote-Datenmodule selbst durch einen Aufruf von *EnableWebTransport* in der Methode *UpdateRegistry*. Wenn Sie Web-Verbindungen zum Anwendungsserver verhindern möchten, entfernen Sie diesen Aufruf.

Web-Verbindungen können die Vorteile der Objektverwaltung nutzen. Dabei wird die Zahl der Instanzen des Remote-Datenmoduls beschränkt, die der Server für Client-Anforderungen zur Verfügung stellen kann. Auf diese Weise läßt sich sicherstellen, daß der Server nur dann Ressourcen für das Datenmodul und dessen Datenverbindung belegt, wenn diese benötigt werden. Weitere Informationen zur Objektverwaltung finden Sie im Abschnitt »Remote-Datenmodule verwalten« auf Seite 14-8.

Im Gegensatz zu anderen Verbindungskomponenten sind für Verbindungen, die via HTTP eingerichtet werden, keine Callbacks möglich.

OLEnterprise verwenden

Mit OLEnterprise können Sie den Business Object Broker einsetzen und auf einen Broker auf dem Client verzichten. Der Business Object Broker stellt die Lastverteilung, die Wiederherstellung nach Fehlern und die Positionstransparenz bereit.

Für OLEnterprise muß das OLEnterprise-Laufzeitmodul auf den Client- und Server-Systemen installiert werden. Das OLEnterprise-Laufzeitmodul verwaltet die Durchführung von Automatisierungsaufrufen und wickelt die Kommunikation zwischen

den Client- und Server-Systemen über RPCs (Remote Procedure Calls) ab. Weitere Informationen finden Sie in der Dokumentation zu OLEnterprise.

CORBA-Verbindungen einsetzen

Mit CORBA können Sie mehrschichtige Datenbankanwendungen in eine CORBA-Umgebung integrieren. Eine CORBA-Verbindung wird z. B. vom MIDAS-Client für Java-Komponenten eingesetzt. CORBA und Java werden von vielen Plattformen unterstützt. Sie haben deshalb die Möglichkeit, plattformübergreifende MIDAS-Anwendungen zu erstellen. Weitere Informationen zum Einsatz von CORBA in Delphi finden Sie in Kapitel 28, »CORBA-Anwendungen«.

Durch den Einsatz von CORBA nutzt Ihre Anwendung automatisch die Vorteile der Lastverteilung, der Positionstransparenz und der Wiederherstellung nach Fehlern. Diese Funktionalität wird von der ORB-Laufzeitsoftware bereitgestellt. Außerdem können Sie Hooks hinzufügen, um weitere CORBA-Dienste zu nutzen.

Mehrschichtige Anwendungen erstellen

Nachstehend werden die grundlegenden Schritte zum Erstellen einer mehrschichtigen Datenbankanwendung aufgeführt.

1 Erstellen Sie den Anwendungsserver.

2 Registrieren oder installieren Sie den Anwendungsserver.

- Wenn der Anwendungsserver DCOM, HTTP, Sockets oder OLEnterprise als Kommunikationsprotokoll verwendet, agiert er als Automatisierungsserver und muß wie alle anderen ActiveX- und COM-Server registriert werden. Informationen zur Registrierung einer Anwendung finden Sie unter »Eine Anwendung als Automatisierungsserver registrieren« auf Seite 47-6.
- Wenn Sie MTS verwenden, muß es sich beim Anwendungsserver um eine ActiveX-Bibliothek und nicht um eine EXE-Datei handeln. Da alle COM-Aufrufe über den MTS-Proxy erfolgen müssen, muß der Anwendungsserver nicht registriert werden. Sie installieren ihn vielmehr mit MTS. Informationen zum Installieren von Bibliotheken mit MTS finden Sie im Abschnitt »MTS-Objekte in einem MTS-Package installieren« auf Seite 51-24.
- Verwendet der Anwendungsserver CORBA, ist die Registrierung optional. Damit die Client-Anwendungen die dynamische Bindung an ihre Schnittstelle nutzen können, müssen Sie die Schnittstelle des Servers im Schnittstellen-Repository installieren. Außerdem muß der Anwendungsserver gegebenenfalls beim OAD (Object Activation Daemon) registriert werden, damit Client-Anwendungen den Anwendungsserver starten können, wenn dieser noch nicht ausgeführt wird. Weitere Informationen zur Registrierung eines CORBA-Servers finden Sie im Abschnitt »Server-Schnittstellen registrieren« auf Seite 28-9.

3 Erstellen Sie die Client-Anwendung.

Die Reihenfolge der Erstellung ist wichtig. Sie müssen den Anwendungsserver erstellen und ausführen, bevor Sie einen Client erstellen. Während der Erstellung soll-

ten Sie den Client testen, indem Sie eine Verbindung zum Anwendungsserver einrichten. Sie können den Client natürlich auch ohne Angabe des Anwendungsservers erstellen und dann zur Laufzeit den Namen des Servers bereitstellen. Bei diesem Vorgehen können Sie jedoch während der Erstellung des Clients nicht überprüfen, ob die Anwendung wie gewünscht arbeitet. Außerdem können Sie mit dem Objektinspektor keine Server oder Provider auswählen.

Hinweis Wenn Sie die Client-Anwendung nicht auf demselben System wie den Server erstellen und keine Web- oder Socket-Verbindung verwenden, können Sie den Anwendungsserver gegebenenfalls auf dem Client-System installieren oder registrieren. Anschließend können die Verbindungskomponenten den Anwendungsserver während des Entwurfs erkennen, und Sie erhalten die Möglichkeit, die Namen von Servern und Providern in Dropdown-Listen im Objektinspektor auszuwählen. (Bei Verwendung einer Web- oder Socket-Verbindung ruft die Verbindungskomponente die Namen der registrierten Server vom Server-Rechner ab.)

Anwendungsserver erstellen

Sie können Anwendungsserver weitgehend wie andere Datenbankanwendungen erstellen. Der Hauptunterschied besteht in dem Datenmengen-Provider, der in den Anwendungsserver eingefügt wird.

Erstellen Sie ein neues Projekt, speichern Sie es, und führen Sie die folgenden Schritte durch, um einen Anwendungsserver zu erstellen:

1 Fügen Sie ein neues Remote-Datenmodul in das Projekt ein. Wählen Sie im Hauptmenü *Datei / Neu* und anschließend die Registerkarte *Mehrschichtig* im Dialogfeld *Objektgalerie*. Wählen Sie dann eine der folgenden Optionen:

   - *Externes Datenmodul*, wenn Sie einen COM-Automatisierungsserver erstellen, auf den die Clients über DCOM, HTTP, Sockets oder OLEnterprise zugreifen.
   - *MTS Data Module*, wenn Sie eine ActiveX-Bibliothek erstellen, auf die Clients über MTS zugreifen. Verbindungen können mit DCOM, HTTP, Sockets oder OLEnterprise eingerichtet werden.
   - *CORBA-Datenmodul*, wenn Sie einen CORBA-Server erstellen.

   Ausführliche Informationen zur Einrichtung eines Remote-Datenmoduls finden Sie im Abschnitt »Remote-Datenmodul einrichten« auf Seite 14-15.

Hinweis Remote-Datenmodule sind nicht nur einfache Datenmodule. Das CORBA-Remote-Datenmodul agiert als CORBA-Server. Andere Datenmodule sind COM-Automatisierungsobjekte.

2 Fügen Sie die gewünschten Datenmengenkomponenten in das Datenmodul ein. Richten Sie die Komponenten für den Zugriff auf den Datenbankserver ein.

3 Fügen Sie für jede Datenmenge eine *TDataSetProvider*-Komponente in das Datenmodul ein. Dieser Provider fungiert als Broker für Client-Anforderungen und sorgt für die Erstellung der erforderlichen Datenpakete.

4. Weisen Sie der Eigenschaft *DataSet* jeder Provider-Komponente den Namen der Datenmenge zu, auf die zugegriffen werden soll. Sie können auch andere Eigenschaften des Providers einstellen. Ausführliche Informationen zur Einrichtung eines Providers finden Sie in Kapitel 15, »Provider-Komponenten«.
5. Schreiben Sie den Quelltext für den Anwendungsserver, um Ereignisse sowie die Business Rules, die Datenprüfung und die Sicherheit für den gemeinsamen Zugriff zu implementieren. Sie können die Schnittstelle des Anwendungsservers erweitern, um weitere Methoden des Zugriffs der Client-Anwendung auf den Server bereitzustellen. Ausführliche Informationen zur Erweiterung der Schnittstelle des Anwendungsservers finden Sie im Abschnitt »Schnittstelle des Anwendungsservers erweitern« auf Seite 14-19.
6. Speichern, kompilieren und registrieren oder installieren Sie den Anwendungsserver.
   - Wenn der Anwendungsserver DCOM, HTTP, Sockets oder OLEnterprise als Kommunikationsprotokoll verwendet, agiert er als Automatisierungsserver, der wie jeder andere ActiveX- bzw. COM-Server registriert werden muß. Informationen zur Registrierung einer Anwendung finden Sie im Abschnitt »Eine Anwendung als Automatisierungsserver registrieren« auf Seite 47-6.
   - Wenn Sie MTS verwenden, muß es sich beim Anwendungsserver um eine ActiveX-Bibliothek handeln, nicht um eine EXE-Datei. Da alle COM-Aufrufe über den MTS-Proxy erfolgen müssen, ist eine Registrierung des Anwendungsservers nicht erforderlich. Statt dessen installieren Sie den Server mit MTS. Informationen zum Installieren von Bibliotheken mit MTS finden Sie im Abschnitt »MTS-Objekte in einem MTS-Package installieren« auf Seite 51-24.
   - Verwendet der Anwendungsserver CORBA, ist die Registrierung optional. Wenn Sie die dynamische Bindung von Client-Anwendungen an die Schnittstelle zulassen, müssen Sie die Schnittstelle des Servers im Schnittstellen-Repository installieren. Sollen Client-Anwendungen den Server starten, wenn dieser noch nicht ausgeführt wird, muß der Server beim OAD (Object Activation Daemon) registriert werden. Weitere Informationen zur Registrierung eines CORBA-Servers finden Sie im Abschnitt »Server-Schnittstellen registrieren« auf Seite 28-9.
7. Wenn Ihre Server-Anwendung nicht mit DCOM arbeitet, müssen Sie die Laufzeit-Software installieren, die Client-Botschaften empfängt, das Remote-Datenmodul instantiiert und Sequenzen der Schnittstellenaufrufe bildet.
   - Bei TCP/IP-Sockets handelt es sich um eine Socket-Dispatcher-Anwendung namens SCKTSRVR.EXE.
   - Für HTTP-Verbindungen wird HTTPSRVR.DLL verwendet, eine ISAPI/NSAPI-DLL, die mit dem Web-Server installiert werden muß.
   - Für OLEnterprise wird die OLEnterprise-Laufzeitdatei verwendet.
   - Für CORBA wird der VisiBroker ORB eingesetzt.

Remote-Datenmodul einrichten

Wenn Sie einen Anwendungsserver einrichten und ausführen, baut dieser keine Verbindungen zu Client-Anwendungen auf. Die Verbindungsverwaltung erfolgt statt dessen durch die Client-Anwendungen. Die Client-Anwendung verwendet ihre Verbindungskomponente, um eine Verbindung zum Anwendungsserver einzurichten. Über diese Verbindung wird dann mit dem ausgewählten Provider kommuniziert. Diese Operationen erfolgen automatisch. Sie müssen weder Quelltext zur Verarbeitung eingehender Anforderungen schreiben noch Schnittstellen bereitstellen.

Wenn Sie das Remote-Datenmodul erstellen, müssen Sie verschiedene Informationen bereitstellen, um die Reaktion auf Client-Anforderungen festzulegen. Diese Informationen sind vom Typ des Remote-Datenmoduls abhängig. Einzelheiten zum erforderlichen Remote-Datenmodultyp finden Sie im Abschnitt »Die Struktur des Anwendungsservers« auf Seite 14-5.

TRemoteDataModule konfigurieren

Wählen Sie *Datei / Neu* und anschließend in der Registerkarte *Mehrschichtig* des Dialogfeldes *Objektgalerie* die Option *Externes Datenmodul*, um eine *TRemoteDataModule*-Komponente in die Anwendung einzufügen. Der Experte für externe Datenmodule wird gestartet.

Sie müssen einen Klassennamen für das Remote-Datenmodul angeben. Dabei handelt es sich um den Basisnamen eines Nachkommen von *TRemoteDataModule*, der in der Anwendung erstellt wird. Gleichzeitig ist dies der Basisname der Schnittstelle für diese Klasse. Wenn Sie beispielsweise den Klassennamen *MyDataServer* angeben, erzeugt der Experte eine neue Unit, die *TMyDataServer* als Nachkommen von *TRemoteDataModule* deklariert. In diesem Nachkommen wird *IMyDataServer* (Nachfahre von *IAppServer*) implementiert.

Hinweis Sie können eigene Eigenschaften und Methoden in die neue Schnittstelle einfügen. Weitere Informationen finden Sie im Abschnitt »Schnittstelle des Anwendungsservers erweitern« auf Seite 14-19.

Wenn Sie eine DLL (ActiveX-Bibliothek) erstellen, müssen Sie das Thread-Modell im Remote-Datenmodulexperten angeben. Sie können die Option *Einfach*, *Apartment*, *Frei* oder *Beides* auswählen.

- Verwenden Sie *Einfach*, dann stellt COM sicher, daß zu einem bestimmten Zeitpunkt nur eine Client-Anforderung verarbeitet wird. Sie müssen also keine Verwaltung für gleichzeitig bearbeitete Client-Anforderungen implementieren.
- Verwenden Sie *Apartment*, dann stellt COM sicher, daß jede Instanz des Remote-Datenmoduls zu einem bestimmten Zeitpunkt nur jeweils eine Client-Anforderung verarbeitet. Beim Schreiben des Quelltextes für eine mit der Option Apartment erstellte Bibliothek müssen Sie Thread-Konflikte vermeiden, die durch den Einsatz globaler Variablen oder Objekte verursacht werden, die sich nicht im Remote-Datenmodul befinden. Bei Verwendung von BDE-konformen Datenmengen sollte immer das Apartment-Modell eingesetzt werden. (Zur Verwaltung von Threads für BDE-konforme Datenmengen wird außerdem eine Sitzungskomponente benötigt, deren Eigenschaft *AutoSessionName* den Wert *True* hat.)

- Verwenden Sie *Frei*, kann die Anwendung gleichzeitige Client-Anforderungen über mehrere Threads verarbeiten. In diesem Fall sind Sie für die Verhinderung von Thread-Konflikten verantwortlich. Da mehrere Clients gleichzeitig auf das Remote-Datenmodul zugreifen, müssen die Instanzdaten (Eigenschaften, enthaltene Objekte usw.) und die globalen Variablen geschützt werden. Dieses Modell sollte bei Verwendung von ADO-Datenmengen eingesetzt werden.
- Verwenden Sie *Beides*, arbeitet die Bibliothek weitgehend wie mit der Option *Frei*. Es gibt jedoch eine Ausnahme: Alle Callbacks (Aufrufe an die Client-Schnittstelle) werden automatisch sequentiell durchgeführt.

Wenn Sie eine EXE-Datei erstellen, müssen Sie angeben, wie die Instanzenbildung erfolgen soll. Sie können *Einfache Instanz* oder *Mehrfache Instanz* wählen. (Die Option Intern kann nur verwendet werden, wenn der Client-Quelltext in demselben Prozeß-Speicherraum ausgeführt wird.)

- Verwenden Sie *Einfache Instanz*, startet jede Client-Verbindung eine eigene Instanz der ausführbaren Datei. Dieser Prozeß initiiert eine einzelne Instanz des Remote-Datenmoduls für die betreffende Client-Verbindung.
- Verwenden Sie *Mehrfache Instanz*, instantiiert eine einzelne Instanz der Anwendung (Prozeß) alle Remote-Datenmodule, die für Clients erstellt werden. Jedes Remote-Datenmodul ist einer einzelnen Client-Verbindung zugeordnet, die Module werden jedoch alle in einem Prozeß-Speicherraum ausgeführt.

TMTSDataModule konfigurieren

Wählen Sie *Datei / Neu* und anschließend *MTS-Datenmodul* in der Registerkarte *Mehrschichtig* im Dialogfeld *Objektgalerie*, um eine *TMTSDataModule*-Komponente in die Anwendung einzufügen. Der MTS-Datenmodulexperte wird gestartet.

Sie müssen einen Klassennamen für das Remote-Datenmodul angeben. Dabei handelt es sich um den Basisnamen eines Nachfahren von *TMTSDataModule*, der von der Anwendung erstellt wird. Gleichzeitig ist dies der Basisname der Schnittstelle für diese Klasse. Wenn Sie z. B. den Klassennamen *MyDataServer* angeben, erstellt der Experte eine neue Unit, in der *TMyDataServer* als Nachfahre von *TMTSDataModule* deklariert wird. Dieser Nachfahre implementiert *IMyDataServer*, einen Nachfahren von *IAppServer*.

Hinweis Sie können eigene Methoden und Eigenschaften in die neue Schnittstelle einfügen. Weitere Informationen finden Sie im Abschnitt »Schnittstelle des Anwendungsservers erweitern« auf Seite 14-19.

MTS-Anwendungen sind immer DLLs (ActiveX-Bibliotheken). Sie müssen das Thread-Modell im MTS-Datenmodulexperten angeben. Wählen Sie eine der Optionen *Einfach, Apartment, Frei oder Beide*.

- Verwenden Sie *Einfach*, stellt MTS sicher, daß zu einem bestimmten Zeitpunkt nur eine Client-Anforderung verarbeitet wird. In diesem Fall müssen Sie nicht auf die Vermeidung von Konflikten zwischen den Client-Anforderungen achten.
- Verwenden Sie *Apartment* oder *Frei*, ist das Resultat identisch: MTS stellt sicher, daß jede Instanz des Remote-Datenmoduls zu einem bestimmten Zeitpunkt genau eine Client-Anforderung verarbeitet. Die Aufrufe verwenden jedoch nicht immer

denselben Thread. Sie können keine Thread-Variablen verwenden, da nicht sichergestellt ist, daß nachfolgende Aufrufe an das Remote-Datenmodul über den vorherigen Thread erfolgen. Sie müssen sicherstellen, daß keine Konflikte durch den Einsatz globaler Variablen oder durch Objekte auftreten, die sich nicht im Remote-Datenmodul befinden. Anstelle globaler Variablen können Sie den Manager für gemeinsam genutzte Eigenschaften verwenden. Weitere Informationen dazu finden Sie unter »Shared Property Manager« auf Seite 51-14.

- Verwenden Sie *Beide*, dann ruft MTS die Schnittstelle des Remote-Datenmoduls wie bei Verwendung der Optionen *Apartment* bzw. *Frei* auf. Callbacks an die Client-Anwendung werden jedoch automatisch sequentiell durchgeführt. Sie sind also nicht für die Vermeidung von Konflikten verantwortlich.

Hinweis Die Modelle *Apartment* und *Frei* unter MTS sind nicht mit den gleichnamigen Modellen unter DCOM identisch.

Sie müssen MTS-Transaktionsattribute für das Remote-Datenmodul angeben. Wählen Sie eine der folgenden Optionen:

- *Transaktion wird benötigt*. Wenn Sie diese Option wählen, wird jeder Aufruf der Schnittstelle des Remote-Datenmoduls durch den Client im Kontext einer MTS-Transaktion durchgeführt. Wenn der Aufrufer eine Transaktion bereitstellt, muß keine neue Transaktion erstellt werden.
- *Eine neue Transaktion wird benötigt*. Wenn Sie diese Option wählen, wird bei jedem Zugriff eines Clients auf die Schnittstelle des Remote-Datenmoduls automatisch eine neue Transaktion für diesen Aufruf erstellt.
- *Transaktionen unterstützen*. Wenn Sie diese Option wählen, kann das Remote-Datenmodul im Kontext einer MTS-Transaktion eingesetzt werden. Der Aufrufer muß die Transaktion jedoch im Schnittstellenaufruf bereitstellen.
- *Transaktionen nicht unterstützen*. Wenn Sie diese Option wählen, kann das Remote-Datenmodul nicht im Kontext von MTS-Transaktionen eingesetzt werden.

TCorbaDataModule konfigurieren

Um eine *TCorbaDataModule*-Komponente in die Anwendung einzufügen, wählen Sie *Datei / Neu* und anschließend *CORBA-Datenmodul* in der Registerkarte *Mehrschichtig* des Dialogfeldes *Objektgalerie*. Der CORBA-Datenmodulexperte wird gestartet.

Sie müssen einen Klassennamen für das Remote-Datenmodul angeben. Dies ist der Basisname eines Nachfahren von *TCorbaDataModule*, der von der Anwendung erstellt wird. Gleichzeitig handelt es sich um den Basisnamen der Schnittstelle für die Klasse. Geben Sie beispielsweise den Klassennamen *MyDataServer* an, erstellt der Experte eine neue Unit, die *TMyDataServer* als Nachfahren von *TCorbaDataModule* deklariert. Dieser Nachfahre implementiert *IMyDataServer* als Nachfahren von *IAppServer*.

Hinweis Sie können eigene Methoden und Eigenschaften in die neue Schnittstelle einfügen. Weitere Informationen zum Erweitern der Schnittstelle des Datenmoduls finden Sie im Abschnitt »Schnittstelle des Anwendungsservers erweitern« auf Seite 14-19.

Der CORBA-Datenmodulexperte ermöglicht die Festlegung des Typs der Instanzenbildung für das Remote-Datenmodul. Sie können freigegebene Instanzen oder separate Instanzen für jeden Client verwenden.

- Wenn Sie freigegebene Instanzen wählen, erstellt die Anwendung eine einzelne Instanz des Remote-Datenmoduls, das alle Client-Anforderungen verarbeitet. Dies ist das konventionelle Modell zur Entwicklung von CORBA-Anwendungen.
- Verwenden Sie dagegen eine Instanz pro Client, wird für jede Client-Verbindung eine neue Instanz des Remote-Datenmoduls erstellt. Diese Instanz bleibt bestehen, bis der Timeout-Zeitraum verstrichen ist, ohne daß Client-Botschaften eingegangen sind. Mit diesem Modell kann der Server Instanzen freigeben, die nicht mehr von den Clients benötigt werden. Es besteht jedoch das Risiko, daß der Server vorzeitig freigegeben wird, wenn der Client über einen längeren Zeitraum nicht auf die Serverschnittstelle zugreift.

Hinweis Im Unterschied zur Instanzenbildung für COM-Server – hier bestimmt das Modell die Anzahl der ausgeführten Prozeßinstanzen – legt unter CORBA die Instanzenbildung die Anzahl der von einem Objekt erstellten Instanzen fest. Diese Instanzen werden in einer einzelnen Instanz der ausführbaren Datei der Server-Anwendung erstellt.

Außer dem Instanzenbildungsmodell müssen Sie das Thread-Modell im CORBA-Datenmodulexperten angeben. Sie können die Option *Einfach-Threads* oder *Multi-Threads* wählen.

- Wenn Sie *Einfach-Threads* wählen, empfängt jede Instanz des Remote-Datenmoduls zu einem bestimmten Zeitpunkt nur eine Client-Anforderung. Sie können also problemlos auf die Objekte im Remote-Datenmodul zugreifen. Dabei müssen Sie allerdings Konflikte vermeiden, die sich aus dem Einsatz globaler Variablen oder durch Objekte ergeben können, die sich nicht im Remote-Datenmodul befinden.
- Wenn Sie *Multi-Threads* wählen, erhält jede Client-Verbindung einen eigenen Thread. Die Anwendung kann also gleichzeitig von mehreren Clients aufgerufen werden, die jeweils einen separaten Thread verwenden. Sie müssen Konflikte vermeiden, die durch den gleichzeitigen Zugriff auf Instanzdaten und auf den globalen Speicher entstehen können. Das Erstellen eines auf dem Multithread-Modell basierenden Servers ist nicht unkompliziert, da jeder Zugriff auf die Objekte im Remote-Datenmodul geschützt erfolgen muß.

Daten-Provider für Anwendungsserver erstellen

Jedes Remote-Datenmodul eines Anwendungsservers enthält eine oder mehrere Provider-Komponenten. Jede Client-Datenmenge verwendet einen bestimmten Provider. Der Provider fungiert als Bindeglied zwischen der Client-Datenmenge und den entsprechenden Daten. Eine Provider-Komponente (*TDataSetProvider*) ordnet die Daten in einem Datenpaket an, sendet dieses Paket an den Client und trägt die vom Client empfangenen Aktualisierungen ein.

Der größte Teil der Datenlogik des Anwendungsservers wird von den im Remote-Datenmodul enthaltenen Provider-Komponenten abgewickelt. Die Anwendungslo-

gik (Business Rules) und die Datenlogik werden in Ereignisbehandlungsroutinen implementiert, die Client-Anforderungen beantworten. Über die Eigenschaften der Provider-Komponenten wird gesteuert, welche Informationen in den Datenpaketen enthalten sind. In Kapitel 15, »Provider-Komponenten« wird ausführlich beschrieben, wie mit einer Provider-Komponente die Interaktion mit Client-Anwendungen gesteuert wird.

Schnittstelle des Anwendungsservers erweitern

Client-Anwendungen arbeiten mit dem Anwendungsserver zusammen, indem sie eine Instanz des Remote-Datenmoduls erstellen oder eine Verbindung zum Remote-Datenmodul einrichten. Dessen Schnittstelle bildet die Grundlage der Kommunikation mit dem Anwendungsserver.

Sie können die Schnittstelle des Remote-Datenmoduls erweitern, um zusätzliche Unterstützung für Client-Anwendungen bereitzustellen. Diese Schnittstelle ist ein Nachkomme von *IAppServer*. Sie wird automatisch vom Experten erstellt, wenn Sie das Remote-Datenmodul erstellen .

Folgendermaßen können Sie die Schnittstelle des Remote-Datenmoduls erweitern:

- Wählen Sie die Option *Zur Schnittstelle hinzufügen* im Menü *Bearbeiten* der IDE. Geben Sie an, ob eine Prozedur, eine Funktion oder eine Eigenschaft hinzugefügt wird. Nachdem Sie die Syntax eingegeben und auf *OK* geklickt haben, gelangen Sie in den Quelltexteditor, in dem Sie das neue Schnittstellenelement implementieren können.
- Verwenden Sie den Typbibliothekseditor. Wählen Sie dort die Schnittstelle für den Anwendungsserver, und klicken Sie auf die Werkzeugschaltfläche für den jeweils hinzugefügten Typ des Schnittstellenelements (Methode oder Eigenschaft). Weisen Sie dem Schnittstellenelement in der Registerkarte *Attribute* einen Namen zu, geben Sie die Parameter in der Registerkarte *Parameter* an, und aktualisieren Sie anschließend die Typbibliothek. Weitere Informationen zum Typbibliothekseditor finden Sie in Kapitel 50, »Mit Typbibliotheken arbeiten«. Viele der Merkmale, die Sie im Typbibliothekseditor angeben können (beispielsweise Hilfekontext, Version usw.), gelten nicht für CORBA-Schnittstellen. Wenn Sie Werte für diese Merkmale angeben, werden sie ignoriert.

Die Operationen von Delphi nach dem Erweitern der Schnittstelle sind davon abhängig, ob Sie einen COM- (*TRemoteDataModule* oder *TMTSDataModule*) oder einen CORBA-Server (*TCorbaDataModule*) erstellen.

- Wenn Sie eine COM-Schnittstelle erweitern, werden die Änderungen in den Quelltext der Unit und in die Typbibliothek (TLB-Datei) eingefügt.
- Wenn Sie eine CORBA-Schnittstelle erweitern, werden die Änderungen in den Quelltext Ihrer Unit und in die automatisch generierte Unit _TLB übernommen. Die Unit _TLB wird in die **uses**-Klausel Ihrer Unit eingefügt. Sie müssen die Unit in die **uses**-Klausel der Client-Anwendung einfügen, wenn Sie die frühe Bindung nutzen wollen. Zusätzlich können Sie mit der Schaltfläche *In IDL exportieren* eine IDL-Datei im Typbibliothekseditor speichern. Die IDL-Datei wird für die Regi-

strierung der Schnittstelle in der Schnittstellenablage und beim OAD (Object Activation Daemon) benötigt.

Hinweis Die TLB-Datei muß explizit gespeichert werden. Wählen Sie dazu im Typbibliothekseditor den Befehl *Aktualisieren*, und speichern Sie die Änderungen dann in der IDE.

Nachdem Sie die Schnittstelle des Remote-Datenmoduls erweitert haben, können Sie die Implementierung abschließen, indem Sie den Quelltext für die betreffenden Eigenschaften und Methoden schreiben.

Client-Anwendungen rufen Schnittstellenerweiterungen über die Eigenschaft *AppServer* ihrer Verbindungskomponente auf. Weitere Informationen finden Sie im Abschnitt »Serverschnittstellen aufrufen« auf Seite 14-27.

Callbacks der Schnittstelle des Anwendungsservers hinzufügen

Sie können es dem Anwendungsserver ermöglichen, mittels eines Callback auf die Client-Anwendung zuzugreifen. Die Client-Anwendung übergibt dazu eine Schnittstelle an eine Methode des Anwendungsservers, die vom Anwendungsserver später nach Bedarf aufgerufen wird. Wenn Sie die Schnittstelle des Remote-Datenmoduls um Callbacks erweitern, können jedoch keine HTTP-basierten Verbindungen benutzt werden, da *TWebConnection* keine Callbacks unterstützt. Bei Verwendung einer Socket-Verbindung müssen Client-Anwendungen angeben, ob sie Callbacks unterstützen. Dies geschieht mit Hilfe der Eigenschaft *SupportCallbacks*. Alle anderen Verbindungstypen unterstützen Callbacks automatisch.

Schnittstelle des Anwendungsservers unter MTS erweitern

Wenn Sie Transaktionen oder Bedarfsaktivierungen unter MTS verwenden, müssen Sie sicherstellen, daß alle neuen Methoden die Methode *SetComplete* aufrufen, um MTS den Abschluß der Operation mitzuteilen. Anschließend kann die Transaktion abgeschlossen und das Remote-Datenmodul deaktiviert werden. Die neuen Methoden dürfen keine Werte zurückliefern, die es dem Client ermöglichen, direkt mit Objekten oder Schnittstellen auf dem Anwendungsserver zu kommunizieren. Jede Kommunikation, die nicht über die Schnittstelle des Remote-Datenmoduls erfolgt, umgeht den MTS-Proxy. Dies kann zum Fehlschlagen von Transaktionen führen. Wenn Sie ein statusloses MTS-Datenmodul verwenden, kann das Umgehen des MTS-Proxy zu Systemabstürzen führen, da nicht sichergestellt ist, daß das Remote-Datenmodul aktiv ist.

Client-Anwendung erstellen

In den meisten Bereichen entspricht das Erstellen einer mehrschichtigen Client-Anwendung dem Erstellen einer konventionellen zweischichtigen Client-Anwendung. Die grundlegenden Unterschiede ergeben sich jedoch aus dem Einsatz der folgenden Komponenten:

- Eine Verbindungskomponente, die zur Einrichtung der Verbindung mit dem Anwendungsserver dient.

- Mindestens eine *TClientDataSet*-Komponente, die eine Verbindung mit dem Daten-Provider des Anwendungsservers herstellt. Datensensitive Steuerelemente des Clients werden über Datenquellenkomponenten und nicht über *TTable*-, *TQuery*-, *TStoredProc*- oder *TADODataSet*-Komponenten mit diesen Client-Datenmengen verbunden.

Richten Sie ein neues Projekt ein, und führen Sie die folgenden Schritte durch, um eine mehrschichtige Client-Anwendung zu erstellen:

1. Fügen Sie ein neues Datenmodul in das Projekt ein.
2. Fügen Sie eine Verbindungskomponente in das Datenmodul ein. Der Typ dieser Verbindungskomponente ist vom Kommunikationsprotokoll abhängig, das Sie verwenden wollen. Im Abschnitt »Die Struktur der Client-Anwendung« auf Seite 14-4 finden Sie weitere Informationen.
3. Geben Sie in den Eigenschaften der Verbindungskomponente den Anwendungsserver an, zu dem eine Verbindung eingerichtet werden soll. Weitere Informationen zur Einrichtung der Verbindungskomponente finden Sie im Abschnitt »Verbindung zum Anwendungsserver einrichten« auf Seite 14-21.
4. Stellen Sie die weiteren Eigenschaften der Verbindungskomponente den Anforderungen der Anwendung entsprechend ein. Sie können beispielsweise die Eigenschaft *ObjectBroker* setzen, um der Verbindungskomponente die dynamische Auswahl eines Servers zu ermöglichen. Weitere Informationen zur Verwendung der Verbindungskomponenten finden Sie im Abschnitt »Server-Verbindungen verwalten« auf Seite 14-26.
5. Fügen Sie die benötigten *TClientDataSet*-Komponenten in das Datenmodul ein, und weisen Sie der Eigenschaft *RemoteServer* jeder Komponente den Namen der in Schritt 2 eingefügten Verbindungskomponente zu. Eine ausführliche Beschreibung der Client-Datenmengen finden Sie in Kapitel 24, »Client-Datenmengen«.
6. Stellen Sie die Eigenschaft *ProviderName* der *TClientDataSet*-Komponenten ein. Wenn die Verbindungskomponente während des Entwurfs mit dem Anwendungsserver verbunden ist, können Sie einen der verfügbaren Anwendungsserver-Provider in der Dropdown-Liste der Eigenschaft *ProviderName* auswählen.
7. Erstellen Sie die Client-Anwendung dann wie jede andere Datenbankanwendung. Gegebenenfalls können Sie spezielle Funktionen von Client-Datenmengen nutzen, die eine Interaktion mit den Provider-Komponenten des Anwendungsservers ermöglichen. Eine Beschreibung dieser Funktionen finden Sie im Abschnitt »Eine Client-Datenmenge mit einem Daten-Provider verwenden« auf Seite 24-15.

Verbindung zum Anwendungsserver einrichten

Eine Client-Anwendung verwendet Verbindungskomponenten, um Verbindungen zu einem Anwendungsserver einzurichten und zu verwalten. Sie finden diese Komponenten in der Registerkarte *MIDAS* der Komponentenpalette.

Verbindungskomponenten dienen folgenden Zwecken:

- Festlegen des Protokolls für die Kommunikation mit dem Anwendungsserver. Jeder Komponententyp repräsentiert ein anderes Kommunikationsprotokoll. Im Abschnitt »Verbindungsprotokoll wählen« auf Seite 14-9 finden Sie Informationen zu den Vorteilen und Beschränkungen der verfügbaren Protokolle.
- Angeben der Methode für die Suche nach dem Server-Rechner. Diese Methode ist protokollabhängig.
- Identifizieren des Anwendungsservers auf dem Server-Rechner.

  Wenn Sie CORBA nicht verwenden, können Sie den Server in der Eigenschaft *ServerName* oder *ServerGUID* angeben. *ServerName* gibt den Basisnamen der Klasse an, die Sie beim Erstellen des Remote-Datenmoduls im Anwendungsserver verwendet haben. Im Abschnitt »Remote-Datenmodul einrichten« auf Seite 14-15 finden Sie Informationen zur Festlegung dieses Wertes auf dem Server. Wurde der Server auf dem Client-Rechner registriert oder installiert bzw. die Verbindungskomponente mit dem Server-Rechner verbunden, können Sie der Eigenschaft *ServerName* während des Entwurfs einen Wert zuweisen, indem Sie diesen in einer Dropdown-Liste im Objektinspektor auswählen. *ServerGUID* gibt die GUID der Schnittstelle des Remote-Datenmoduls an. Sie können diesen Wert mit dem Typbibliothekseditor ermitteln.

  Wenn Sie CORBA verwenden, geben Sie den Server in der Eigenschaft *RepositoryID* an. RepositoryID enthält die Repository-ID der Generatorschnittstelle des Anwendungsservers. Diese bildet das dritte Argument im Aufruf der Methode *TCorbaVCLComponentFactory.Create*, die automatisch in den Initialisierungsabschnitt der Implementierungs-Unit des CORBA-Servers eingefügt wird. Sie können dieser Eigenschaft auch den Basisnamen der Schnittstelle des CORBA-Datenmoduls zuweisen (wie in der Eigenschaft *ServerName* anderer Verbindungskomponenten). Der String wird dann automatisch in die entsprechende Repository-ID konvertiert.
- Server-Verbindungen verwalten . Mit Verbindungskomponenten können Verbindungen eingerichtet oder getrennt und Schnittstellen des Anwendungsservers aufgerufen werden.

Normalerweise befindet sich der Anwendungsserver auf einem anderen Rechner als die Client-Anwendung. Auch wenn sich der Server auf demselben Rechner wie die Client-Anwendung befindet (beispielsweise beim Entwickeln und Testen der gesamten mehrschichtigen Anwendung), können Sie die Verbindungskomponente verwenden, um den Anwendungsserver über seinen Namen sowie einen Server-Rechner anzugeben und auf die Schnittstelle des Anwendungsservers zuzugreifen.

Verbindung über DCOM angeben

Wenn Sie DCOM zur Kommunikation mit dem Anwendungsserver verwenden, fügen Client-Anwendungen eine *TDCOMConnection* -Komponente für die Einrichtung der Verbindung zum Anwendungsserver ein. *TDCOMConnection* gibt den Rechner mit dem Server in der Eigenschaft *ComputerName* an.

Wenn *ComputerName* leer ist, unterstellt die DCOM-Verbindungskomponente, daß sich der Anwendungsserver auf dem Client-Rechner befindet oder ein entsprechender Eintrag in der Registrierungsdatenbank des Client-Rechners vorliegt. Stellen Sie

bei Verwendung von DCOM keinen Eintrag in der Registrierungsdatenbank für den Anwendungsserver auf dem Client bereit und befindet sich der Server nicht auf dem Client-Rechner, müssen Sie einen Wert für *ComputerName* angeben.

Hinweis Existiert in der Registrierungsdatenbank ein Eintrag für den Anwendungsserver, können Sie diesen Eintrag mit *ComputerName* überschreiben. Dies ist insbesondere beim Entwickeln, Testen und Debuggen hilfreich.

Kann die Client-Anwendung einen von mehreren Servern auswählen, verwenden Sie die Eigenschaft *ObjectBroker* anstelle von ComputerName. Weitere Informationen finden Sie im Abschnitt »Broker-Verbindungen« auf Seite 14-25.

Wenn Sie den Namen eines Host-Computers oder Servers angeben, der nicht gefunden werden kann, löst die DCOM-Verbindungskomponente beim Öffnen der Verbindung eine Exception aus.

Verbindung über Sockets angeben

Sie können über Sockets Verbindungen zu Anwendungsservern auf jedem Rechner einrichten, der über eine TCP/IP-Adresse verfügt. Diese Verbindungsmethode hat den Vorteil, daß sie für fast alle Rechner möglich ist. Sie stellt jedoch keinerlei Sicherheitsprotokolle bereit. Wenn Sie Sockets verwenden, müssen Sie eine *TSocketConnection* -Komponente zur Einrichtung der Verbindung mit dem Anwendungsserver einfügen.

TSocketConnection identifiziert den Server-Rechner über die IP-Adresse oder den Host-Namen des Server-Systems sowie die Port-Nummer des Socket-Dispatcher-Programms (SCKTSRVR.EXE), das auf dem Server-Rechner ausgeführt wird. Weitere Informationen zu IP-Adressen und Port-Nummern finden Sie im Abschnitt »Die Sokkets beschreiben« auf Seite 30-4.

Diese Informationen werden in drei Eigenschaften von *TSocketConnection* angegeben:

- *Address* gibt die IP-Adresse des Servers an.
- *Host* gibt den Host-Namen des Servers an.
- *Port* gibt die Port-Nummer des Socket-Dispatcher-Programms auf dem Anwendungsserver an.

Address und Host schließen sich gegenseitig aus. Durch das Einstellen eines Wertes wird der andere gelöscht. Informationen zum Einsatz dieser Werte finden Sie im Abschnitt »Den Host beschreiben« auf Seite 30-4.

Wenn die Client-Anwendung einen Server auswählen kann, können Sie die Eigenschaft *ObjectBroker* verwenden, statt einen Wert für *Address* oder *Host* anzugeben. Weitere Informationen finden Sie im Abschnitt »Broker-Verbindungen« auf Seite 14-25.

Standardmäßig enthält *Port* den Wert 211, die Port-Standardnummer des Socket-Dispatcher-Programms, das mit Delphi geliefert wird. Wenn das Socket-Dispatcher-Programm für einen anderen Port konfiguriert wurde, müssen Sie Port den entsprechenden Wert zuweisen.

Hinweis Sie können den Port des Socket-Dispatchers ändern, während der Dispatcher ausgeführt wird, indem Sie mit der rechten Maustaste auf das Symbol *Borland Socket Server* klicken und Eigenschaften wählen.

Socket-Verbindungen unterstützen zwar keine Sicherheitsprotokolle, es ist aber möglich, sie anzupassen und eine Verschlüsselung zu implementieren. Dazu erstellen und registrieren Sie ein COM-Objekt, das die Schnittstelle *IDataIntercept* unterstützt. Diese Schnittstelle ermöglicht das Ver- und Entschlüsseln von Daten. Geben Sie in der Eigenschaft *InterceptGUID* der Socket-Verbindungskomponente die GUID für das neue COM-Objekt an. Klicken Sie dann mit der rechten Maustaste auf das Symbol *Borland Socket Server*, wählen Sie Eigenschaften, und geben Sie in der Registerkarte *Eigenschaften* dieselbe GUID als Auffang-GUID an. Diese Technik können Sie auch zum Komprimieren und Dekomprimieren von Daten einsetzen.

Verbindung über HTTP angeben

Von jedem Rechner, der über eine TCP/IP-Adresse verfügt, läßt sich via HTTP eine Verbindung zum Anwendungsserver einrichten. Im Gegensatz zu einer Socket-Verbindung ist bei Verwendung von HTTP eine Kommunikation mit Servern möglich, die durch eine Firewall geschützt sind, da die von WININET.DLL bereitgestellten SSL-Sicherheitsfunktionen genutzt werden können. Um eine Verbindung zum Anwendungsserver via HTTP einzurichten, wird eine *TWebConnection* -Komponente benötigt.

Die Web-Verbindungskomponente baut eine Verbindung zur Web-Server-Anwendung (HTTPSRVR.DLL) auf, die dann mit dem Anwendungsserver kommuniziert. *TWebConnection* lokalisiert HTTPSRVR.DLL mit Hilfe einer URL (Uniform Resource Locator). In dieser URL sind folgende Parameter festgelegt: das Protokoll (http bzw. https, wenn die SSL-Sicherheitsfunktionen genutzt werden), der Host-Name des Rechners, auf dem sich der Web-Server und HTTPSRVR.DLL befinden, sowie der Pfad zur Web-Server-Anwendung (HTTPSRVR.DLL). Die URL wird in der Eigenschaft *URL* angegeben.

Hinweis Wenn eine Verbindung mit *TWebConnection* eingerichtet wird, muß auf dem Client-Rechner WININET.DLL installiert sein. Arbeiten Sie mit IE3 oder höher, befindet sich WININET.DLL im Windows-Systemverzeichnis.

Falls der Web-Server eine Authentifizierung anfordert oder auf einen Proxy-Server zugegriffen wird, für den eine Authentifizierung erforderlich ist, müssen Benutzername und Kennwort in den Eigenschaften *UserName* und *Password* angegeben werden, damit sich die Verbindungskomponente beim Server anmelden kann.

Sind für die Client-Anwendung mehrere Server verfügbar, können Sie die Eigenschaft *ObjectBroker* anstelle von *URL* verwenden. Weitere Informationen finden Sie im Abschnitt »Broker-Verbindungen« auf Seite 14-25.

Verbindung über OLEnterprise angeben

Wenn Sie OLEnterprise zur Kommunikation mit dem Anwendungsserver einsetzen, muß die Client-Anwendung eine *TOLEnterpriseConnection* -Komponente für die Einrichtung der Verbindung zum Anwendungsserver enthalten. Mit OLEnterprise kön-

nen Sie eine direkte Verbindung oder eine Verbindung über den Business Object Broker zum Server-Rechner einrichten.

- Wenn Sie OLEnterprise ohne Business Object Broker einsetzen, weisen Sie der Eigenschaft *ComputerName* den Namen eines Server-Rechners zu (entspricht der Eigenschaft *ComputerName* für eine DCOM-Verbindung).
- Wenn Sie die Dienste zur Lastverteilung und zur Wiederherstellung nach Fehlern des Business Object Broker nutzen wollen, weisen Sie der Eigenschaft *BrokerName* den Namen des Business Object Broker zu.

ComputerName und *BrokerName* schließen sich gegenseitig aus. Wenn Sie einer Eigenschaft einen Wert zuweisen, wird der Wert der anderen Eigenschaft gelöscht.

Weitere Informationen zum Einsatz von OLEnterprise finden Sie in der zugehörigen Dokumentation.

Verbindung über CORBA angeben

Zur Angabe einer CORBA-Verbindung wird nur die Eigenschaft *RepositoryID* benötigt. Ein Smart Agent im lokalen Netzwerk sucht automatisch einen verfügbaren Server für den CORBA-Client.

Mit den anderen Eigenschaften der CORBA-Verbindungskomponente können Sie gegebenenfalls Server angeben, zu denen die Client-Anwendung keine Verbindung einrichten soll. Soll nicht der CORBA Smart Agent einen verfügbaren Server ermitteln, können Sie in der Eigenschaft *HostName* den Server angeben, zu dem die Verbindung eingerichtet werden soll. Wird die Serverschnittstelle von mehr als einer Objektinstanz implementiert, können Sie das zu verwendende Objekt in der Eigenschaft *ObjectName* angeben.

Die *TCorbaConnection*-Komponente kann eine Schnittstelle zum CORBA-Datenmodul auf dem Anwendungsserver auf zwei Arten erhalten:

- Wenn Sie die frühe (statische) Bindung verwenden, müssen Sie die vom Typbibliothekseditor generierte Datei _TLB.PAS in die Client-Anwendung einfügen. Die frühe Bindung wird empfohlen, da sie schneller als die späte (dynamische) Bindung ist und zusätzliche Typprüfungen beim Kompilieren durchgeführt werden.
- Wenn Sie die späte (dynamische) Bindung verwenden, muß die Schnittstelle im Schnittstellen-Repository registriert werden. Weitere Informationen zur Registrierung einer Schnittstelle im Schnittstellen-Repository finden Sie im Abschnitt »CORBA-Clients schreiben« auf Seite 28-13.

Weitere Informationen zum Vergleich zwischen früher und später Bindung finden Sie im Abschnitt »Serverschnittstellen aufrufen« auf Seite 14-27.

Broker-Verbindungen

Wenn für die Client-Anwendung verschiedene Server verfügbar sind, kann ein Object Broker die verfügbaren Server-Systeme ermitteln. Der Object Broker verwaltet eine Liste der Server, aus der die Verbindungskomponente ausgewählt werden kann. Wenn die Verbindungskomponente eine Verbindung zum Anwendungsserver ein-

richten muß, fordert sie vom Object Broker einen Computernamen (bzw. eine IP-Adresse, einen Host-Namen oder eine URL) an. Der Broker stellt einen Namen bereit, und die Verbindungskomponente richtet die Verbindung ein. Kann dieser Name nicht verwendet werden, weil beispielsweise der betreffende Server heruntergefahren wurde, stellt der Broker einen anderen Namen bereit. Dieser Vorgang wird wiederholt, bis eine Verbindung eingerichtet ist.

Sobald die Verbindungskomponente eine Verbindung mit dem vom Broker angegebenen Namen eingerichtet hat, speichert sie diesen Namen in der entsprechenden Eigenschaft (*ComputerName*, *Address*, *Host*, *RemoteHost* oder *URL*). Wird die Verbindung geschlossen und zu einem späteren Zeitpunkt wieder benötigt, greift die Verbindungskomponente zunächst auf den Wert dieser Eigenschaft zu. Nur wenn die Verbindung nicht eingerichtet werden kann, fordert sie einen neuen Namen vom Broker an.

Der Object Broker wird in der Eigenschaft *ObjectBroker* der Verbindungskomponente angegeben. Wird dieser Eigenschaft ein Wert zugewiesen, speichert die Verbindungskomponente den Wert von *ComputerName*, *Address*, *Host*, *RemoteHost* bzw. *URL* nicht.

Hinweis Verwenden Sie die Eigenschaft *ObjectBroker* nicht für OLEnterprise- oder CORBA-Verbindungen. Diese Protokolle besitzen eigene Broker-Dienste.

Server-Verbindungen verwalten

Verbindungskomponenten dienen in erster Linie der Suche nach dem Anwendungsserver und der Einrichtung der Verbindung zu diesem. Da sie Server-Verbindungen verwalten, können Sie auch Verbindungskomponenten zum Aufruf der Schnittstellenmethoden des Anwendungsservers verwenden.

Verbindung zum Server einrichten

Zur Suche nach dem Anwendungsserver und zur Einrichtung der Verbindung müssen Sie zunächst die Eigenschaften der Verbindungskomponente einstellen, um den Anwendungsserver zu identifizieren. Dieser Prozeß wird im Abschnitt »Verbindung zum Anwendungsserver einrichten« auf Seite 14-21 beschrieben. Zusätzlich sollten alle Client-Datenmengen in der Eigenschaft *RemoteServer* die Verbindungskomponente angeben, wenn sie über diese Verbindungskomponente mit dem Anwendungsserver kommunizieren.

Die Verbindung wird automatisch geöffnet, wenn Client-Datenmengen auf den Anwendungsserver zugreifen. Wenn Sie beispielsweise der Eigenschaft *Active* der Client-Datenmenge den Wert *True* zuweisen, wird die Verbindung geöffnet (vorausgesetzt, der Wert der Eigenschaft *RemoteServer* wurde festgelegt).

Wenn Sie die Client-Datenmengen nicht mit Verbindungskomponenten verknüpfen, können Sie die Verbindung öffnen, indem Sie der Eigenschaft *Connected* der Verbindungskomponente *True* zuweisen.

Bevor eine Verbindungskomponente eine Verbindung zum Anwendungsserver einrichtet, generiert sie das Ereignis *BeforeConnect*. Sie können hier spezielle Operationen in der Ereignisbehandlungsroutine für *BeforeConnect* definieren, die vor Einrichtung

der Verbindung ausgeführt werden. Nach der Einrichtung der Verbindung generiert die Verbindungskomponente das Ereignis *AfterConnect*, mit dessen Hilfe weitere Operationen definiert werden können.

Server-Verbindung schließen oder ändern

Eine Verbindungskomponente schließt eine Verbindung zum Anwendungsserver unter folgenden Bedingungen:

- Der Eigenschaft *Connected* wird *False* zugewiesen.
- Die Verbindungskomponente wird freigegeben. Ein Verbindungsobjekt wird automatisch freigegeben, wenn ein Benutzer die Client-Anwendung schließt.
- Die Eigenschaften zur Festlegung des Anwendungsservers (*ServerName*, *ServerGUID*, *ComputerName* usw.) werden geändert. Durch eine Änderung dieser Eigenschaften können Sie zur Laufzeit zwischen den verschiedenen Anwendungsservern umschalten. Die Verbindungskomponente schließt die aktuelle Verbindung und richtet eine neue ein.

Hinweis Eine Client-Anwendung kann anstelle einer einzelnen Verbindungskomponente, die zwischen den verschiedenen Anwendungsservern umschalten, auch mehrere Verbindungskomponenten enthalten, die Verbindungen zu jeweils unterschiedlichen Anwendungsservern einrichten.

Bevor eine Verbindungskomponente eine Verbindung schließt, ruft sie automatisch die Ereignisbehandlungsroutine für *BeforeDisconnect* auf, sofern diese verfügbar ist. Sollen vor Abbruch der Verbindung spezielle Operationen durchgeführt werden, müssen Sie diese in der Ereignisbehandlungsroutine für *BeforeDisconnect* definieren. Nach dem Schließen der Verbindung wird die Ereignisbehandlungsroutine für *AfterDisconnect* aufgerufen. Hier können Sie spezielle Operationen definieren, die nach dem Schließen der Verbindung durchgeführt werden sollen.

Serverschnittstellen aufrufen

Anwendungen müssen die Schnittstelle *IAppServer* nicht direkt aufrufen, da die entsprechenden Aufrufe automatisch durchgeführt werden, wenn Sie die Eigenschaften und Methoden der Client-Datenmenge verwenden. Gegebenenfalls sind aber Erweiterungen der Schnittstelle des Remote-Datenmoduls erforderlich. Wenn Sie die Schnittstelle des Anwendungsservers erweitern , benötigen Sie eine Möglichkeit zum Aufrufen der Erweiterungen über die Verbindung, die von der Verbindungskomponente erstellt wird. Dies ist mit der Eigenschaft *AppServer* der Verbindungskomponente möglich. Weitere Informationen zum Erweitern der Schnittstelle des Anwendungsservers finden Sie im Abschnitt »Schnittstelle des Anwendungsservers erweitern« auf Seite 14-19.

AppServer ist eine Variante, welche die Schnittstelle des Anwendungsservers repräsentiert. Sie können Schnittstellenmethoden mit Hilfe von *AppServer* aufrufen, indem Sie Anweisungen wie die folgende schreiben:

```
MyConnection.AppServer.SpecialMethod(x,y);
```

Diese Technik arbeitet jedoch mit später (dynamischer) Bindung im Schnittstellenaufruf. Der Aufruf der Prozedur *SpecialMethod* wird also erst gebunden, wenn der Aufruf zur Laufzeit ausgeführt wird. Die späte Bindung ist sehr flexibel, Vorzüge wie die Programmierhilfe und die Typprüfung können jedoch nicht genutzt werden. Zudem arbeitet die späte Bindung langsamer als die frühe Bindung, da der Compiler zusätzliche Aufrufe an den Server generiert, um Einrichtungsoperationen für Schnittstellenaufrufe vor deren Verarbeitung durchzuführen.

Wenn Sie DCOM oder CORBA als Kommunikationsprotokoll einsetzen, können Sie die frühe Bindung für *AppServer*-Aufrufe nutzen. Verwenden Sie den Operator **as**, um die *AppServer*-Variable in den *IAppServer*-Nachfahren umzuwandeln, den Sie mit dem Remote-Datenmodul erstellt haben. Das folgende Beispiel nutzt die frühe Bindung für eine DCOM-Verbindung:

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

Um die frühe Bindung unter DCOM zu nutzen, muß die Typbibliothek des Servers auf dem Client-Rechner registriert werden. Sie können das mit Delphi ausgelieferte Programm TREGSVR.EXE verwenden, um die Typbibliothek zu registrieren.

Hinweis Die Demo TRegSvr (der Quelltext für TREGSVR.EXE) enthält ein Beispiel zur programmtechnischen Implementierung der Typbibliothek.

Wollen Sie die frühe Bindung unter CORBA nutzen, müssen Sie die Unit _TLB hinzufügen, die vom Typbibliothekseditor im Projekt generiert wird. Geben Sie diese Unit in der **uses**-Klausel Ihrer Unit an.

Wenn Sie TCP / IP oder OLEnterprise verwenden, können Sie die frühe Bindung nicht nutzen, da das Remote-Datenmodul eine duale Schnittstelle besitzt. Sie können jedoch die Disp-Schnittstelle des Anwendungsservers einsetzen, um eine bessere Leistung als bei der normalen späten Bindung zu erreichen. Der Name der Disp-Schnittstelle entspricht dem Namen der Schnittstelle des Remote-Datenmoduls, an den der String »Disp« angefügt wird. Sie können die Eigenschaft *AppServer* einer Variablen dieses Typs zuweisen, um auf die Dispatch-Schnittstelle zuzugreifen. Ein Beispiel:

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface := MyConnection.AppServer;
...
  TempInterface.SpecialMethod(x,y);
...
end;
```

Hinweis Sie müssen die beim Speichern der Typbibliothek generierte Unit _TLB in die **uses**-Klausel des Client-Moduls einfügen, um die Dispatch-Schnittstelle nutzen zu können.

Transaktionen in mehrschichtigen Anwendungen verwalten

Wenn Client-Anwendungen Aktualisierungen an den Anwendungsserver übergeben, kapselt die Provider-Komponente die Aktualisierung und gegebenenfalls das

Beheben von Aktualisierungsfehlern automatisch in einer Transaktion. Diese Transaktion wird eingetragen, wenn die Anzahl der problematischen Datensätze den in *MaxErrors* als Argument für die Methode *ApplyUpdates* angegebenen Wert nicht übersteigt. Andernfalls werden die Änderungen im Rahmen dieser Transaktion rückgängig gemacht.

Sie können die Transaktionsunterstützung durch die Server-Anwendung erweitern, indem Sie eine Datenbankkomponente einfügen oder Passthrough-SQL verwenden. Das Vorgehen entspricht der Transaktionsverwaltung in zweischichtigen Anwendungen. Weitere Informationen zu dieser Form der Transaktionssteuerung finden Sie in den Abschnitten »Transaktionen« auf Seite 13-5 und »Mit Verbindungstransaktionen arbeiten« auf Seite 23-12.

Wenn Sie MTS verwenden, können Sie die Transaktionsunterstützung erweitern, indem Sie MTS-Transaktionen einsetzen. MTS-Transaktionen können für jede Operation im Anwendungsserver verwendet werden, nicht nur für den Datenbankzugriff. Da MTS-Transaktionen das zweiphasige Eintragen unterstützen, können sie sich auf mehrere Datenbanken erstrecken.

Achtung Das zweiphasige Eintragen ist nur in Oracle7- und MS-SQL-Datenbanken vollständig implementiert. Wenn sich eine Transaktion über mehrere Datenbanken erstreckt, bei denen es sich nicht ausnahmslos um Oracle7- oder MS-SQL-Datenbanken handelt, besteht ein kleines Risiko, daß die Transaktionen nicht vollständig verarbeitet werden können. In einer einzelnen Datenbank ist die Transaktionsunterstützung dagegen immer gewährleistet.

Um MTS-Transaktionen einsetzen zu können, müssen Sie gegebenenfalls die Schnittstelle des Anwendungsservers erweitern. Die Schnittstelle muß alle Methoden enthalten, die der Kapselung der Transaktion dienen (sofern dies erforderlich ist). Geben Sie bei der Konfiguration des MTS-Remote-Datenmoduls an, daß dieses in Transaktionen verwendet werden muß. Wenn ein Client eine Methode der Schnittstelle des Anwendungsservers aufruft, wird das Modul automatisch in einer Transaktion gekapselt. Alle Client-Aufrufe an den Anwendungsserver werden dann in der Transaktion protokolliert, bis das Beenden der Transaktion angefordert wird. Die Aufrufe können entweder insgesamt ausgeführt werden oder werden vollständig verworfen.

Hinweis Kombinieren Sie MTS-Transaktionen nicht mit den expliziten Transaktionen einer Datenbankkomponente oder dem Einsatz von Passthrough-SQL. Wird das Remote-Datenmodul in einer MTS-Transaktion aufgeführt, übernimmt es automatisch auch alle direkten Datenbankaufrufe in die Transaktion.

Weitere Informationen zu MTS-Transaktionen finden Sie im Abschnitt »Unterstützung von Transaktionen in MTS« auf Seite 51-8.

Haupt/Detail-Beziehungen unterstützen

Das Einrichten von Haupt/Detail-Beziehungen zwischen Client-Datenmengen in einer Client-Anwendung entspricht dem Einrichten von Haupt/Detail-Formularen in ein- und zweischichtigen Anwendungen. Informationen zum Einrichten von Haupt/Detail-Formularen finden Sie im Abschnitt »Haupt/Detail-Formulare erstellen« auf Seite 20-28.

Diese Vorgehensweise bringt im wesentlichen zwei Nachteile mit sich:

- Die Detailtabelle muß alle Datensätze vom Anwendungsserver abrufen und speichern, auch wenn zu einem bestimmten Zeitpunkt nur ein Detaildatensatz benötigt wird. Dieses Problem kann durch den Einsatz von Parametern weitgehend umgangen werden. Weitere Informationen finden Sie im Abschnitt »Datensätze durch Parameter einschränken« auf Seite 24-18.
- Das Eintragen von Aktualisierungen wird relativ schwierig, da diese Operation von den Client-Datenmengen auf der Ebene der Datenmengen vorgenommen wird und mehrere Datenmengen beteiligt sind. Bereits in einer zweischichtigen Umgebung, in der die Datenbank zum Eintragen der Aktualisierungen in mehrere Tabellen im Rahmen einer Transaktion verwendet werden kann, ist das Eintragen von Aktualisierungen in Haupt / Detail-Formulare nicht unkompliziert. Im Abschnitt »Haupt / Detailtabellen aktualisieren« auf Seite 25-7 finden Sie weitere Informationen zu diesem Thema.

In mehrschichtigen Anwendungen können Sie diese Probleme vermeiden, indem Sie verschachtelte Tabellen zur Repräsentation der Haupt / Detail-Beziehung verwenden. Richten Sie zu diesem Zweck auf dem Anwendungsserver eine Haupt / Detail-Beziehung zwischen den Tabellen ein. Weisen Sie anschließend der Eigenschaft *DataSet* der Provider-Komponente die Haupttabelle zu.

Wenn Clients die Methode *GetRecords* des Providers aufrufen, nimmt sie die Detail-Datenmengen automatisch als *DataSet*-Feld in die Datensätze im Datenpaket auf. Rufen Clients die Methode *ApplyUpdates* des Providers auf, wird das Eintragen der Aktualisierungen automatisch in der richtigen Reihenfolge durchgeführt.

Im Abschnitt »Haupt / Detail-Beziehungen« auf Seite 24-3 finden Sie weitere Informationen zum Einsatz verschachtelter Datenmengen für Haupt / Detail-Beziehungen in Client-Datenmengen.

Statusinformationen in Remote-Datenmodulen unterstützen

Die Schnittstelle *IAppServer* ist für die Steuerung der gesamten Kommunikation zwischen Client-Datenmengen und Providern auf dem Anwendungsserver zuständig. *IAppServer* ist fast immer statuslos. Eine statuslose Anwendung »erinnert« sich nicht an die Vorgänge in den vorhergehenden Client-Aufrufen. Diese Statuslosigkeit ist bei der Verwaltung von Datenbankverbindungen unter MTS erwünscht, da der Anwendungsserver nicht zwischen Datenbankverbindungen für persistente Informationen (aktueller Datensatz) unterscheiden muß. Entsprechendes gilt, wenn Instanzen von Remote-Datenmodulen von mehreren Clients genutzt werden, wie das bei der Bedarfsaktivierung und der Objektverwaltung unter MTS sowie bei typischen CORBA-Servern der Fall ist.

Es gibt aber Situationen, in denen Statusinformationen aus vorhergehenden Aufrufen an den Anwendungsserver benötigt werden. So benötigt der Provider auf dem Anwendungsserver beim inkrementellen Abrufen von Daten Informationen zum aktuellen Datensatz.

Wenn jeder Client über eine eigene Instanz des Remote-Datenmoduls verfügt, ist dies kein Problem, da der Status der jeweiligen Instanz nicht durch Aufrufe anderer Clients geändert werden kann.

Es ist möglich, die Vorzüge gemeinsam genutzter Remote-Datenmodulinstanzen zu nutzen und trotzdem persistente Statusinformationen zu verwenden. Dies ist z. B. hilfreich, wenn eine sehr große Datenmenge inkrementell abgerufen werden muß, weil sie nicht vollständig in den Speicher geladen werden kann.

Vor und nach Aufrufen der *IAppServer*-Schnittstelle, die von der Client-Datenmenge an den Anwendungsserver gesendet werden (*AS_ApplyUpdates*, *AS_Execute*, *AS_GetParams*, *AS_GetRecords* oder *AS_RowRequest*), empfängt dieser ein Ereignis, das es ermöglicht, Statusinformationen zu senden oder abzurufen. Die Provider empfangen entsprechende Ereignisse vor und nach der Beantwortung der Client-Anforderungen. Damit ergibt sich die Möglichkeit, persistente Statusinformationen zwischen Client-Anwendungen und einem statuslosen Anwendungsserver auszutauschen. Das folgende Beispiel zeigt, wie das inkrementelle Abrufen für einen statuslosen Anwendungsserver aktiviert werden kann.

- Mit dem Ereignis *BeforeGetRecords* der Client-Datenmenge wird der Schlüsselwert des letzten Datensatzes an den Anwendungsserver übergeben:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
var
   CurRecord: TBookMark;
begin
   with Sender as TClientDataSet do
   begin
    CurRecord := GetBookmark; { Aktuellen Datensatz speichern }
    try
      Last; { Im neuen Paket nach dem letzten Datensatz suchen }
      OwnerData := FieldValues['Key']; { Schlüsselwert an Anwendungsserver senden }
      GotoBookmark(CurRecord); { Zurück zum aktuellen Datensatz }
    finally
      FreeBookmark(CurRecord);
    end;
   end;
end;
```

- Auf dem Server wird mit dem Ereignis *BeforeGetRecords* des Providers nach der entsprechenden Datensatzmenge gesucht:

```
TRemoteDataModule1.Provider1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TProvider do
    DataSet.Locate('Key', OwnerData, []);
end;
```

Hinweis Im diesem Beispiel wird anstelle einer Positionsmarke ein Schlüsselwert verwendet, um das Ende der Datensatzmenge anzuzeigen. Grund dafür ist, daß Positionsmarken zwischen *IAppServer*-Aufrufen ihre Gültigkeit verlieren können, wenn die Server-Anwendung Datenbank-Handles verwaltet.

Web-basierte MIDAS-Anwendungen erstellen

Um für eine mehrschichtige Datenbankanwendung Web-basierte Clients zu erstellen, müssen Sie die Client-Schicht durch eine spezielle Web-Anwendung ersetzen. Diese muß einerseits als Client für den Anwendungsserver agieren und andererseits eine Web-Server-Anwendung darstellen, die zusammen mit einem Web-Server auf demselben Rechner installiert ist. Die folgende Abbildung zeigt die Architektur einer Web-basierten Datenbankanwendung.

Abbildung 14.1 Web-basierte mehrschichtige Datenbankanwendung

Sie können die MIDAS-Web-Anwendung auf zwei Arten erstellen:

- Kombinieren Sie die MIDAS-Architektur mit der ActiveX-Unterstützung von Delphi, und verteilen Sie eine Client-Anwendung als ActiveX-Steuerelement . Die Client-Anwendung kann auf jedem Browser, der ActiveX unterstützt, als In-Process-Server ausgeführt werden.
- Verwenden Sie XML-Datenpakete zur Erstellung einer InternetExpress-Anwendung. Browser, die Javascript unterstützen, können über HTML-Seiten mit der Client-Anwendung kommunizieren.

Diese beiden Vorgehensweisen unterscheiden sich sehr stark voneinander und haben verschiedene Vor- und Nachteile. Folgendes sollten Sie wissen, bevor Sie sich für eine Methode entscheiden:

- Die Vorgehensweisen basieren auf unterschiedlichen Technologien (ActiveX einerseits und Javascript bzw. XML andererseits). Ein wichtiges Entscheidungskriterium für die Wahl einer Methode ist deshalb das System, mit dem die Endbenutzer Ihrer Anwendung arbeiten. Wenn Sie die erste Methode verwenden, muß ein Browser mit ActiveX-Unterstützung verwendet werden. Die resultierenden Clients lassen sich deshalb nur auf Windows-Plattformen ausführen. Bei der zweiten Vorgehensweise wird ein Browser benötigt, der Javascript und die DHTML-Fähigkeiten von Netscape 4 bzw. Internet Explorer 4 unterstützt.
- ActiveX-Steuerelemente müssen auf den Browser heruntergeladen werden und fungieren dort als In-Process-Server. Entsprechende Clients beanspruchen deshalb mehr Speicherplatz als die Clients einer HTML-basierten Anwendung.
- Bei Verwendung von InternetExpress ist eine Integration in andere HTML-Seiten möglich. Ein ActiveX-Client muß dagegen immer in einem separaten Fenster ausgeführt werden.

- Bei der InternetExpress-Methode wird Standard-HTTP verwendet. Im Gegensatz zu einer ActiveX-Anwendung kann es daher nicht zu Problemen mit Firewalls kommen.
- Die ActiveX-Methode bietet eine größere Flexibilität beim Programmieren, da sich der Entwickler nicht auf die Funktionen der Javascript-Bibliotheken beschränken muß. Die Client-Datenmengen, die beim ActiveX-Ansatz genutzt werden, bieten weitaus mehr Möglichkeiten als die bei der InternetExpress-Methode verwendeten XML-Broker (Filter, Bereiche, aktive Zusammenfassungen, optionale Parameter, verzögertes Abrufen von BLOBs oder verschachtelten Detail-Datensätzen usw.).

Achtung Eine Web-Client-Anwendung kann auf verschiedenen Browsern ein unterschiedliches Verhalten zeigen. Testen Sie deshalb die Anwendung mit allen Browsern, auf denen sie später eingesetzt werden soll.

Client-Anwendung als ActiveX-Steuerelement weitergeben

Die MIDAS-Architektur kann mit den ActiveX-Merkmalen von Delphi kombiniert werden, um Client-Anwendungen als ActiveX-Steuerelemente weiterzugeben.

Wenn Sie eine Client-Anwendung als ActiveX-Steuerelement weitergeben wollen, erstellen Sie den Anwendungsserver wie jeden anderen Anwendungsserver für eine mehrschichtige Anwendung. Sie sollten jedoch DCOM, HTTP oder Sockets als Kommunikationsprotokoll verwenden, da nicht sichergestellt ist, daß auf den Client-Rechnern die OLEnterprise- oder CORBA-Laufzeitsoftware installiert wurde. Ausführliche Informationen zum Erstellen des Anwendungsservers finden Sie im Abschnitt »Anwendungsserver erstellen« auf Seite 14-13.

Beim Erstellen der Client-Anwendung müssen Sie als Basis ein ActiveX-Formular anstatt eines normalen Formulars verwenden. Weitere Informationen finden Sie im Abschnitt»ActiveX-Formular für die Client-Anwendung erstellen«.

Nachdem Sie die Client-Anwendung erstellt und weitergegeben haben, kann der Zugriff auf die Anwendung mit jedem ActiveX-konformen Web-Browser auf einem anderen Rechner erfolgen. Damit ein Web-Browser die Client-Anwendung erfolgreich starten kann, muß der Web-Server auf dem Rechner mit der Client-Anwendung ausgeführt werden.

Wenn die Client-Anwendung DCOM zur Kommunikation mit dem Anwendungsserver verwendet, muß DCOM auf dem Rechner mit dem Web-Browser aktiviert sein. Arbeitet der Rechner mit dem Web-Browser unter Windows 95, wird das von Microsoft erhältliche DCOM95 benötigt.

ActiveX-Formular für die Client-Anwendung erstellen

1 Da die Client-Anwendung als ActiveX-Steuerelement weitergegeben wird, muß auf dem System mit der Client-Anwendung auch ein Web-Server ausgeführt werden. Sie können einen betriebsbereiten Server, beispielsweise den Personal Web Server von Microsoft, verwenden oder mit den in Kapitel 30, »Arbeiten mit Sokkets« beschriebenen Socket-Komponenten einen eigenen Web-Server schreiben.

2. Erstellen Sie die Client-Anwendung, indem Sie die Schritte im Abschnitt »Client-Anwendung erstellen« auf Seite 14-20 durchführen. Beginnen Sie jedoch mit *Datei / Neu / ActiveForm*, und starten Sie das Client-Projekt nicht als normales Delphi-Projekt.
3. Wenn die Client-Anwendung ein Datenmodul verwendet, müssen Sie einen Aufruf hinzufügen, der das Datenmodul explizit im Rahmen der Initialisierung des ActiveX-Formulars erstellt.
4. Nachdem die Client-Anwendung fertiggestellt wurde, müssen Sie das Projekt kompilieren und *Projekt / Optionen für Distribution über das Web* wählen. Im Dialogfeld *Optionen für Distribution über das Web* nehmen Sie folgende Einstellungen vor:
   1. Legen Sie in der Registerkarte *Projekt* das Zielverzeichnis, die URL des Zielverzeichnisses und das HTML-Verzeichnis fest. Normalerweise sind Ziel- und HTML-Verzeichnis mit den Projektverzeichnissen des Web-Servers identisch. Die Ziel-URL ist normalerweise der Name des Server-Rechners, der in den Windows-Netzwerkeinstellungen unter DNS festgelegt wurde.
   2. Nehmen Sie auf der Seite *Zusätzliche Dateien* die Datei MIDAS.DLL in die Client-Anwendung auf.
5. Wählen Sie abschließend *Projekt/Distribution über das Web*, um die Client-Anwendung als ActiveX-Formular weiterzugeben.

Jeder Web-Browser, der ActiveX-Formulare ausführen kann, kann auch Ihre Client-Anwendung ausführen, indem die bei der Weitergabe der Client-Anwendung erstellte HTM-Datei aufgerufen wird. Diese HTM-Datei besitzt denselben Namen wie das Projekt mit der Client-Anwendung. Sie wird in dem als Zielverzeichnis angegebenen Verzeichnis gespeichert.

Web-Anwendungen mit InternetExpress erstellen

MIDAS-Clients können vom Anwendungsserver die Bereitstellung von Datenpaketen anfordern, die nicht als OleVariant sondern in XML codiert sind. Durch die Kombination von XML-codierten Datenpaketen, speziellen Javascript-Bibliotheken mit Datenbankfunktionen und den Delphi-Funktionen zur Unterstützung von Web-Server-Anwendungen lassen sich Thin-Client-Anwendungen erstellen, auf die mit einem Java-fähigen Web-Browser zugegriffen werden kann. Diese Anwendungen bilden die Grundlage für die InternetExpress-Unterstützung durch Delphi.

Hinweis Für die Erstellung einer InternetExpress-Anwendung ist die Kenntnis der Web-Server-Architektur von Delphi und der MIDAS-Datenbankarchitektur erforderlich. Informationen hierzu finden Sie in Kapitel 29, »Internet-Server-Anwendungen«.

Die Registerkarte *InternetExpress* enthält Komponenten, mit der Sie diese Architektur für Web-Server-Anwendungen um die Funktionalität eines MIDAS-Clients erweitern können. Mit Hilfe dieser Komponenten kann die Web-Anwendung HTML-Seiten generieren, die eine Kombination aus HTML, XML und Javascript enthalten. Mittels HTML wird das Layout und das Erscheinungsbild der Seiten im Browser des Endbenutzers festgelegt. XML codiert die Datenpakete als Deltapakete, die Datenbankin-

formationen enthalten. Javascript ermöglicht es den HTML-Steuerelementen, die Daten in den XML-Datenpaketen zu interpretieren und zu verarbeiten.

Bei InternetExpress-Anwendungen, die DCOM für die Verbindung zum Anwendungsserver einsetzen, muß sichergestellt werden, daß der Anwendungsserver den Clients die Zugriffs- und Startberechtigung gewährt. Weitere Informationen finden Sie im Abschnitt »Zugriffs- und Startberechtigung für den Anwendungsserver gewähren« auf Seite 14-37.

Tip Mit den Komponenten der Registerkarte InternetExpress können Web-Server-Anwendungen auch dann mit »echten« Daten erstellt werden, wenn kein Anwendungsserver zur Verfügung steht. Sie müssen dazu lediglich den Provider und seine Datenmenge in das Web-Modul einfügen.

InternetExpress-Anwendung erstellen

Mit den nachstehenden Schritten erstellen Sie eine Web-Anwendung, die HTML-Seiten generiert, über die Benutzer mittels eines Javascript-fähigen Web-Browsers auf die Daten eines Anwendungsservers zugreifen können.

1. Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* anzuzeigen. Wählen Sie anschließend in der Registerkarte *Neu* die Option *Web-Server-Anwendung*. Das weitere Vorgehen wird im Abschnitt »Web-Server-Anwendungen erstellen« auf Seite 29-7 beschrieben.
2. Fügen Sie aus der Registerkarte *MIDAS* der Komponentenpalette eine Verbindungskomponente in das Web-Modul ein, das angezeigt wird, wenn Sie eine neue Web-Server-Anwendung erstellen. Der Typ der Verbindungskomponente ist vom Kommunikationsprotokoll abhängig, das Sie verwenden wollen. Weitere Informationen finden Sie im Abschnitt »Verbindungsprotokoll wählen« auf Seite 14-9.
3. Bestimmten Sie über die Eigenschaften der Verbindungskomponenten den Anwendungsserver, zu dem eine Verbindung eingerichtet werden soll. Informationen hierzu finden Sie im Abschnitt »Verbindung zum Anwendungsserver einrichten« auf Seite 14-21.
4. Fügen Sie anstatt einer Client-Datenmenge einen XML-Broker aus der Registerkarte *InternetExpress* der Komponentenpalette in das Web-Modul ein. Wie *TClientDataSet* repräsentiert auch die *TXMLBroker*-Komponente die Daten von einem Provider auf dem Anwendungsserver und interagiert mit diesem über die *IAppServer*-Schnittstelle. XML-Broker fordern Datenpakete aber nicht wie Client-Datenmengen als OleVariant-Werte, sondern in XML codiert an. Außerdem interagieren sie über InternetExpress-Komponenten und nicht über Datensteuerelemente.
5. Geben Sie in der Eigenschaft *RemoteServer* des XML-Brokers die Verbindungskomponente an, die Sie in Schritt 2 hinzugefügt haben. Legen Sie in der Eigenschaft *ProviderName* den Provider auf dem Anwendungsserver fest, der Daten bereitstellt und Aktualisierungen einträgt. Weitere Informationen zum Einrichten des XML-Brokers finden Sie unter »XML-Broker verwenden« auf Seite 14-371.
6. Fügen Sie für jede Seite, die in den Browsern der Benutzer angezeigt werden soll, einen MIDAS-Seitengenerator in das Web-Modul ein. In der Eigenschaft *Include-*

PathURL jedes MIDAS-Seitengenerators muß die Position der Javascript-Bibliotheken angegeben werden, mit deren Hilfe die generierten HTML-Steuerelemente um Merkmale zur Datenverwaltung erweitert werden.

7 Klicken Sie mit der rechten Maustaste auf eine Web-Seite, und wählen Sie *Aktionseditor*, um den Aktionseditor zu öffnen. Fügen Sie für jede Browser-Botschaft, die verarbeitet werden soll, ein Aktionselement hinzu. Verknüpften Sie die in Schritt 6 eingefügten Seitengeneratoren mit diesen Aktionen. Sie können zu diesem Zweck die Eigenschaft *Producer* der Aktionselemente verwenden oder eine entsprechende Behandlungsroutine für das Ereignis *OnAction* schreiben. Weitere Informationen zum Hinzufügen von Aktionselementen finden Sie im Abschnitt »Aktionen zum Dispatcher hinzufügen« auf Seite 29-10.

8 Doppelklicken Sie auf jede Web-Seite, um den Web-Seiteneditor anzuzeigen. (Sie können zu diesem Zweck auch im Objektinspektor auf die Ellipsenschaltfläche neben der Eigenschaft *WebPageItems* klicken.) In diesem Editor können Sie die Seiten gestalten, die in den Browsern der Benutzer angezeigt werden, indem Sie Web-Elemente in die Seiten einfügen. Weitere Informationen zum Entwerfen von Web-Seiten für eine InternetExpress-Anwendung finden Sie im Abschnitt »Web-Seiten mit einem MIDAS-Seitengenerator erstellen« auf Seite 14-40.

9 Erstellen Sie die Web-Anwendung. Nachdem diese Anwendung auf dem Web-Server installiert wurde, kann sie von Browsern aufgerufen werden. Diese müssen dazu den Namen der Anwendung im *ScriptName*-Abschnitt der URL und den Namen der Web-Seitenkomponenten im *PathInfo*-Abschnitt angeben.

Javascript-Bibliotheken verwenden

Die von den InternetExpress-Komponenten generierten HTML-Seiten sowie die darin enthaltenen Web-Elemente greifen auf verschiedene Javascript-Bibliotheken zu, die mit Delphi geliefert werden:

Tabelle 14.4 Javascript-Bibliotheken

Bibliothek	Funktion
XMLDOM.JS	Diese Bibliothek ist ein DOM-kompatibler XML-Parser, der in Javascript geschrieben ist. Mit seiner Hilfe können Parser, die XML nicht unterstützen, XML-Datenpakete nutzen.
XMLDB.JS	In dieser Bibliothek sind Datenzugriffsklassen definiert, die *TClientDataSet* und *TField* entsprechen.
XMLBIND.JS	In dieser Bibliothek sind Klassen definiert, welche die Datenzugriffsklassen in XMLDB mit HTML-Steuerelementen in der HTML-Seite verknüpfen.

Diese Bibliotheken befinden sich im Verzeichnis SOURCE/WEBMIDAS. Nach der Installation der Bibliotheken müssen Sie ihren Speicherort in der Eigenschaft *IncludePathURL* aller MIDAS-Seitengeneratoren angeben.

Sie können mit Hilfe der Javascript-Klassen in diesen Bibliotheken eigene HTML-Seiten schreiben, anstatt die Web-Seiten mit Web-Elementen zu erstellen. Um die Größe der generierten Web-Seiten möglichst gering zu halten, ist in diesen Klassen nur eine

minimale Fehlerprüfung implementiert. Sie müssen deshalb sicherstellen, daß der Quelltext Ihrer HTML-Seiten keine unzulässigen Aktionen enthält.

Die Klassen in den Javascript-Bibliotheken werden ständig weiterentwickelt und regelmäßig aktualisiert. Wenn Sie den Javascript-Quelltext nicht mit Web-Elementen sondern direkt mit diesen Klassen erstellen, sollten Sie immer die aktuellste Version verwenden. Sie erhalten die neuesten Versionen dieser Bibliotheken und die zugehörige Dokumentation unter der Adresse www.borland.com/CodeCentral.

Zugriffs- und Startberechtigung für den Anwendungsserver gewähren

Aus der Sicht des Anwendungsservers stammen die Anforderungen der InternetExpress-Anwendung von einem Gastkonto mit dem Namen *IUSR_Computername*. Dabei steht Computername für den Namen des Systems, auf dem die Web-Anwendung ausgeführt wird. Dieses Konto hat per Vorgabe keine Zugriffs- und Startberechtigung für den Anwendungsserver. Wenn Sie die Web-Anwendung ohne diese Berechtigungen ausführen und der Web-Browser versucht, eine angeforderte Seite zu laden, wird der Vorgang mit dem Fehler `EOLE_ACCES_ERROR` abgebrochen.

Hinweis Da der Anwendungsserver immer unter diesem Gastkonto ausgeführt wird, kann er nicht von anderen Konten heruntergefahren werden.

Um der Web-Anwendung die Zugriffs- und Startberechtigung für den Anwendungsserver zu gewähren, muß die Datei DCOMCNFG.EXE ausgeführt werden. Sie befindet sich im Verzeichnis SYSTEM32 des Rechners, auf dem der Anwendungsserver ausgeführt wird. Für die entsprechende Konfiguration sind folgende Schritte erforderlich:

1. Starten Sie DCOMCNFG, und wählen Sie in der Registerkarte *Anwendungen* den Anwendungsserver aus.
2. Klicken Sie auf die Schaltfläche *Eigenschaften*. Aktivieren Sie im angezeigten Dialogfeld die Registerkarte *Sicherheit*.
3. Markieren Sie die Option *Use Custom Access Permissions*, und klicken Sie auf die Schaltfläche *Bearbeiten*. Fügen Sie den Namen *IUSR_Computername* zur Liste der Konten hinzu, die über eine Zugriffsberechtigung verfügen (ersetzen Sie *Computername* durch den Namen des Rechners, auf dem die Web-Anwendung ausgeführt wird).
4. Markieren Sie die Option *Use Custom Launch Permissions*, und klicken Sie auf die Schaltfläche *Bearbeiten*. Fügen Sie *IUSR_Computername* auch in diese Liste ein.
5. Klicken Sie auf die Schaltfläche *Übernehmen*.

XML-Broker verwenden

Der XML-Broker erfüllt zwei wichtige Aufgaben:

- Er ruft XML-Datenpakete vom Anwendungsserver ab und stellt sie für die Web-Elemente bereit, welche die HTML-Seiten für die InternetExpress-Anwendung generieren.

- Er empfängt Aktualisierungen in Form von Deltapaketen von Browsern und trägt sie in den Anwendungsserver ein.

XML-Datenpakete abrufen

Bevor der XML-Broker XML-Datenpakete an die Komponenten übergeben kann, die für die Generierung der HTML-Seiten zuständig sind, muß er die Daten vom Anwendungsserver abrufen. Dazu fordert er zunächst mittels einer Verbindungskomponente die *IAppServer*-Schnittstelle des Anwendungsservers an. Damit der XML-Generator diese Schnittstelle nutzen kann, müssen die Werte folgender Eigenschaften eingestellt werden:

- Geben Sie in der Eigenschaft *RemoteServer* die Verbindungskomponente an, mit der die Verbindung zum Anwendungsserver eingerichtet und dessen *IAppServer*-Schnittstelle abgerufen wird. Während des Entwurfs kann dieser Wert in einer Dropdown-Liste des Objektinspektors ausgewählt werden.
- Geben Sie in der Eigenschaft *ProviderName* den Namen der Verbindungskomponente auf dem Anwendungsserver an, welche die Datenmenge repräsentiert, für die XML-Datenpakete angefordert werden. Dieser Provider stellt die XML-Datenpakete bereit und trägt Aktualisierungen aus XML-Datenpaketen ein. Wenn zur Entwurfszeit die Eigenschaft RemoteServer mit einem Wert belegt ist und von der Verbindungskomponente eine Verbindung eingerichtet wurde, wird im Objektinspektor eine Liste mit allen verfügbaren Providern angezeigt. (Bei Verwendung einer DCOM-Verbindung muß zusätzlich der Anwendungsserver auf dem Client-Rechner registriert sein.)

Mit den beiden folgenden Eigenschaften legen Sie fest, was in den Datenpaketen enthalten ist.

- Wenn der Provider auf dem Anwendungsserver eine Abfrage oder eine Stored Procedure repräsentiert, kann es sinnvoll sein, vor dem Abrufen eines XML-Datenpakets zusätzliche Parameterwerte zu übergeben. Diese können in der Eigenschaft *Params* angegeben werden.
- Mit Hilfe der Eigenschaft *MaxRecords* können Sie die Zahl der Datensätze begrenzen, die in ein Datenpaket aufgenommen werden.

Die Komponenten, die für die Generierung von HTML und Javascript für die InternetExpress-Client-Anwendung zuständig sind, verwenden automatisch das XML-Datenpaket des XML-Brokers, sobald ihre Eigenschaft *XMLBroker* festgelegt wurde. Um das XML-Datenpaket direkt im Quelltext abzurufen, verwenden Sie die Methode *RequestRecords*.

Hinweis Der XML-Broker empfängt bei jeder Übergabe eines Datenpakets an eine andere Komponente (bzw. bei jedem Aufruf der Methode *RequestRecords*) das Ereignis *OnRequestRecords*. Sie können dieses Ereignis dazu benutzen, anstelle des vom Anwendungsserver abgerufenen Datenpakets einen eigenen XML-String bereitzustellen. So ist es beispielsweise möglich, mit der Methode *GetXMLRecords* ein XML-Datenpaket vom Anwendungsserver abzurufen und es dann zu bearbeiten, bevor es an die entstehende Web-Seite übergeben wird.

Aktualisierung aus XML-Datenpaketen eintragen

Wenn der XML-Broker in das Web-Modul (bzw. in ein Datenmodul mit einer *TWebDispatcher*-Komponente) eingefügt wird, registriert er sich beim Web-Dispatcher automatisch als Auto-Dispatcher-Objekt. Sie müssen deshalb im Unterschied zu anderen Komponenten kein Aktionselement für den XML-Broker erstellen, damit dieser auf Aktualisierungsbotschaften eines Web-Browsers antworten kann. Aktualisierungsbotschaften enthalten XML-Datenpakete, die im Anwendungsserver eingetragen werden sollen. Sie werden normalerweise durch das Klicken auf eine Schaltfläche in einer der HTML-Seiten generiert, die von der Web-Client-Anwendung erstellt werden.

Damit der Dispatcher die Botschaften des XML-Brokers erkennt, müssen sie in der Eigenschaft *WebDispatch* beschrieben werden. Geben Sie in der Eigenschaft *PathInfo* die Pfadkomponente der URL an, an die Botschaften für den XML-Broker gesendet werden. In der Eigenschaft *MethodType* geben Sie den Methoden-Header der Aktualisierungsbotschaften an, die an diese URL adressiert sind (normalerweise mtPost). Sollen alle Botschaften mit dem angegebenen Pfad beantwortet werden, setzen Sie *MethodType* auf *mtAny*. Sie können verhindern, daß der XML-Broker direkt auf die Aktualisierungsbotschaften antwortet (um sie beispielsweise explizit mit einem Aktionselement zu verarbeiten). Dazu belegen Sie die Eigenschaft *Enabled* mit dem Wert *False*. Im Abschnitt »Anforderungsbotschaften verteilen« auf Seite 29-10 wird ausführlich beschrieben, wie der Web-Dispatcher die Komponente bestimmt, die Botschaften des Web-Browsers verarbeitet.

Bei der Weitergabe einer Aktualisierungsbotschaft durch den Dispatcher an den XML-Broker werden gleichzeitig die Aktualisierungen an den Anwendungsserver übergeben. Falls Aktualisierungsfehler vorliegen, empfängt der Dispatcher ein XML-Deltapaket, in dem diese Fehler beschrieben sind. Zuletzt sendet der Dispatcher eine Antwortbotschaft an den Browser zurück, die den Browser entweder dazu veranlaßt, zu der Seite zurückzukehren, die das XML-Deltapaket generiert hat, oder irgendeine andere Seite zu senden.

Eine Reihe von Ereignissen gibt Ihnen die Möglichkeit, in die verschiedenen Stadien dieses Aktualisierungsprozesses einzugreifen:

1. Wenn der Dispatcher die Aktualisierungsbotschaft an den XML-Broker übergibt, empfängt er das Ereignis *BeforeDispatch*. In einer Behandlungsroutine für dieses Ereignis können Sie eine Vorverarbeitung der Anforderung durchführen oder sogar die gesamte Botschaft verarbeiten. Außerdem können Sie den XML-Broker in die Lage versetzen, neben Aktualisierungsbotschaften auch andere Botschaften zu bearbeiten.
2. Wird die Botschaft nicht in der Ereignisbehandlungsroutine für *BeforeDispatch* verarbeitet, empfängt der XML-Broker das Ereignis *OnRequestUpdate*. Damit haben Sie die Möglichkeit, die Standardverarbeitung zu umgehen und eine eigene Routine zum Eintragen der Aktualisierungen zu schreiben.
3. Wenn die Anforderung nicht in einer *OnRequestUpdate*-Ereignisbehandlungsroutine verarbeitet wird, trägt der XML-Broker die Aktualisierungen ein. Dabei auftretende Fehler werden in einem Deltapaket an den XML-Broker zurückgesendet.

4 Liegen keine Aktualisierungsfehler vor, empfängt der XML-Broker ein *OnGetResponse*-Ereignis. In einer Behandlungsroutine für dieses Ereignis können Sie eine Antwortbotschaft erstellen, die anzeigt, daß das Eintragen erfolgreich war oder aktualisierte Daten an den Browser sendet. Wenn die *OnGetResponse*-Ereignisbehandlungsroutine nicht abgeschlossen wird (der Parameter *Handled* hat nicht den Wert *True*), sendet der XML-Broker eine Antwortbotschaft, die den Browser veranlaßt, zu dem Dokument zurückzukehren, das das Deltapaket generiert hat.

5 Wenn beim Eintragen der Aktualisierungen Fehler aufgetreten sind, empfängt der XML-Broker das Ereignis *OnGetErrorResponse*. In einer entsprechenden Ereignisbehandlungsroutine können Sie dann versuchen, die Aktualisierungsfehler zu beseitigen. Sie können aber auch eine Web-Seite erstellen, in der die Fehler für den Endbenutzer beschrieben werden. Wenn die *OnGetErrorResponse*-Ereignisbehandlungsroutine nicht beendet wird (und damit den Parameter *Handled* nicht auf *True* setzt), ruft der XML-Broker, um den Inhalt der Antwortbotschaft zu erstellen, einen speziellen Inhalts-Producer namens *ReconcileProducer* auf.

6 Zuletzt empfängt der XML-Broker das Ereignis *AfterDispatch*. In einer Behandlungsroutine für dieses Ereignis können weitere Aktionen implementiert werden, die vor dem Senden der Antwort an den Web-Browser ausgeführt werden sollen.

Web-Seiten mit einem MIDAS-Seitengenerator erstellen

Jeder MIDAS-Seitengenerator erstellt ein HTML-Dokument, das in den Browsern der Anwendungs-Clients angezeigt wird. Enthält die Anwendung mehrere Web-Dokumente, muß für jede ein separater MIDAS-Seitengenerator verwendet werden.

Ein MIDAS-Seitengenerator ist eine spezielle Seitengeneratorkomponente. Diese kann wie jeder andere Seitengenerator in der Eigenschaft *Producer* eines Aktionselements angegeben oder explizit in einer *OnAction*-Ereignisbehandlungsroutine aufgerufen werden. Weitere Informationen zur Verwendung von String-Generatoren mit Aktionselementen finden Sie unter »Mit Aktionselementen auf Anforderungsbotschaften antworten« auf Seite 29-13. Einzelheiten zu Seitengeneratoren enthält der Abschnitt »Seitengeneratoren einsetzen« auf Seite 29-20.

Im Unterschied zu den meisten Seitengeneratoren verwendet der MIDAS-Seitengenerator eine Standardvorlage als Wert der Eigenschaft *HTMLDoc*. Diese Vorlage enthält eine Gruppe HTML-transparenter Tags, mit deren Hilfe der MIDAS-Seitengenerator ein HTML-Dokument (mit integriertem Javascript und XML) sowie den von anderen Komponenten erzeugten Inhalt zusammenführt. Damit der Generator diese HTML-transparenten Tags übersetzen und das Dokument erstellen kann, müssen Sie die Position der Javascript-Bibliotheken bereitstellen, die für eingebettetes Javascript auf der Seite erforderlich sind. Dies geschieht mit Hilfe der Eigenschaft *IncludePathURL*.

Mit dem Web-Seiteneditor können die Komponenten angegeben werden, die Teile der Web-Seite erstellen. Zum Öffnen des Web-Seiteneditors doppelklicken Sie auf die Web-Seitenkomponente oder klicken im Objektinspektor auf die Ellipsenschaltfläche neben der Eigenschaft *WebPageItems*.

Eine im Web-Seiteneditor hinzugefügte Komponente generiert HTML-Code, durch den eines der HTML-transparenten Tags in der Standardvorlage des MIDAS-Seitengenerators ersetzt wird. Die Komponenten werden dann in der Eigenschaft *WebPageItems* angegeben. Nachdem Sie alle Komponenten in der gewünschten Reihenfolge hinzugefügt haben, können Sie die Vorlage anpassen, indem Sie eigenen HTML-Code hinzufügen oder die Standard-Tags ändern.

Web-Seiteneditor verwenden

Im Web-Seiteneditor können Sie Web-Elemente für einen Web-Seitengenerator erstellen und die resultierende HTML-Seite anzeigen. Zum Öffnen des Web-Seiteneditors doppelklicken Sie auf eine MIDAS-Seitengeneratorkomponente.

Hinweis Zur Ausführung des Web-Seiteneditors benötigen Sie Internet Explorer 4 oder höher.

Im oberen Teil des Web-Seiteneditors werden die Web-Elemente angezeigt, die das HTML-Dokument generieren. Diese Elemente sind verschachtelt. Jeder Web-Elementtyp verarbeitet den HTML-Code, der von seinen Unterelementen generiert wird. Die einzelnen Typen können unterschiedliche Unterelemente enthalten. Auf der linken Seite des Editors wird die Hierarchie der Web-Elemente angezeigt. Sie erkennen daran, wie die Elemente verschachtelt sind. Auf der rechten Seite sehen Sie die Web-Elemente, die im aktuell ausgewählten Element enthalten sind. Nachdem Sie eine Komponente im oberen Teil des Editors ausgewählt haben, können Sie im Objektinspektor ihre Eigenschaften einstellen.

Wenn Sie dem aktuellen Element ein neues Unterelement hinzufügen möchten, klikken Sie auf die Schaltfläche *Objektgalerie*. Im Dialogfeld *Web-Komponente hinzufügen* werden dann die Elemente angezeigt, die dem aktuellen Element hinzugefügt werden können.

Der MIDAS-Seitengenerator kann einen von zwei möglichen Elementtypen enthalten. Jeder Typ generiert ein HTML-Formular:

- *TDataForm* erzeugt ein HTML-Formular zur Anzeige von Daten sowie die Steuerelemente, mit denen diese Daten bearbeitet oder Aktualisierungen übergeben werden können.

  Die Elemente, die Sie *TDataForm* hinzufügen, können Daten auf zwei Arten anzeigen: in Form eines Gitters für mehrere Datensätze (*TDataGrid*) oder in einer Gruppe von Steuerelementen, von denen jedes ein Feld aus einem Datensatz darstellt (*TFieldGroup*). Sie können Schaltflächen in das Formular einfügen, die das Navigieren und das Eintragen von Aktualisierungen ermöglichen (*TDataNavigator*). Außerdem kann eine Schaltfläche eingefügt werden, die Aktualisierungen an den Web-Client übergibt und dort einträgt (*TApplyUpdatesButton*). Jedes dieser Elemente enthält Unterelemente für einzelne Felder und Schaltflächen. Wie bei den meisten Web-Elementen können Sie auch in *TDataForm* ein Layout-Gitter einfügen (*TLayoutGroup*), um das Layout der enthaltenen Elemente anzupassen.

- *TQueryForm* generiert ein HTML-Formular zum Anzeigen oder Lesen anwendungsdefinierter Werte. Damit lassen sich beispielsweise Parameterwerte anzeigen und übergeben.

Sie können zwei Arten von Elementen in *TQueryForm* einfügen: Elemente, die anwendungsdefinierte Werte anzeigen (*TQueryFieldGroup*) oder Schaltflächen, mit denen diese Werte übergeben oder zurückgesetzt werden können (*TQueryButtons*). Jedes dieser Elemente enthält Unterelemente für einzelne Werte oder Schaltflächen. Wie bei *TDataForm* kann auch in *TQueryForm* ein Layout-Gitter eingefügt werden.

Im unteren Teil des Web-Seiteneditors sehen Sie den generierten HTML-Code und die resultierende Seite, wie sie in einem Browser (IE4) angezeigt wird.

Eigenschaften von Web-Elementen festlegen

Web-Elemente, die Sie im Web-Seiteneditor hinzufügen, sind spezialisierte Komponenten, die HTML-Code generieren. Jede Web-Elementklasse erstellt ein bestimmtes Steuerelement bzw. einen bestimmten Teil des resultierenden HTML-Dokuments. Es gibt aber auch eine Gruppe universeller Eigenschaften, mit denen das endgültige Erscheinungsbild des HTML-Dokuments beeinflußt werden kann.

Ein Web-Element, das Informationen aus einem XML-Datenpaket darstellt (das z. B. Anzeigesteuerelemente für Feld- oder Parameterwerte oder eine Schaltfläche zur Datenbearbeitung generiert), wird über die Eigenschaft *XMLBroker* mit dem XML-Broker verknüpft, der dieses Datenpaket verwaltet. In der Eigenschaft *XMLDataSetField* können Sie zusätzlich eine Datenmenge angeben, die in einem Datenmengenfeld dieses Datenpakets enthalten ist. Web-Elemente, die einen bestimmten Feld- oder Parameterwert darstellen, besitzen die Eigenschaft *FieldName* bzw. *ParamName*.

Jedem Web-Element kann ein Stilattribut zugewiesen werden, um das Erscheinungsbild des generierten HTML-Dokuments zu beeinflussen. Stile und Formatvorlagen sind Teil des HTML-4-Standards. Mit Hilfe einer Formatvorlage können Sie für ein HTML-Dokument einen Satz von Anzeigeattributen definieren, die für ein Tag und alle von ihm abhängigen Komponenten gelten. Es gibt verschiedene Möglichkeiten, Web-Elementen Stile zuzuweisen:

- Am einfachsten ist es, ein Stilattribut direkt für das Web-Element zu definieren. Sie können zu diesem Zweck die Eigenschaft *Style* verwenden. Der Wert dieser Eigenschaft ist die Attributdefinition einer HTML-Standardstildefinition:

  ```
  color: red.
  ```

- Eine andere Möglichkeit besteht in der Erstellung einer Formatvorlage, die mehrere Stildefinitionen enthält. Jede Definition besteht aus einem Stilkennzeichner (der Name des Tags, für das der Stil gilt oder ein benutzerdefinierter Stilname) sowie der Attributdefinition in geschweiften Klammern:

  ```
  H2 B  {color: red}
  .MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
  ```

  Die komplette Gruppe mit Definitionen wird vom MIDAS-Seitengenerator in dessen Eigenschaft *Styles* verwaltet. Dadurch kann jedes Web-Element über seine Eigenschaft *StyleRule* mit benutzerdefinierten Namen auf die Stile zugreifen.

- Wenn eine Formatvorlage von mehreren Anwendungen genutzt wird, können Sie anstelle der Eigenschaft *Styles* die Eigenschaft *StylesFile* des MIDAS-Seitengenerators verwenden, um die Stildefinitionen anzugeben.

Alle Web-Elemente verfügen über die Eigenschaft *Custom*. Sie können darin eine Gruppe von Optionen angeben, die dem generierten HTML-Tag hinzugefügt werden sollen. HTML definiert für jeden Tag-Typ eine eigene Optionsgruppe. Die VCL-Referenz enthält Beispiele für diese Optionen für die *Custom*-Eigenschaft der meisten Web-Elemente. Weitere Informationen zu diesen Optionen finden Sie in einer HTML-Referenz.

Vorlage des MIDAS-Seitengenerators anpassen

Die Vorlage eines MIDAS-Seitengenerators ist ein HTML-Dokument mit zusätzlichen Tags, die von der Anwendung dynamisch übersetzt werden. Der MIDAS-Seitengenerator generiert zunächst eine Standardvorlage als Wert der Eigenschaft *HTMLDoc*. Diese Standardvorlage hat folgende Form:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

Die HTML-transparenten Tags in der Standardvorlage werden folgendermaßen übersetzt:

<#INCLUDES> generiert die Anweisungen, mit deren Hilfe die Javascript-Bibliotheken aufgenommen werden. Die entsprechenden Anweisungen haben folgende Form:

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </
SCRIPT>
```

<#STYLES> generiert die Anweisungen, die aus den in der Eigenschaft *Styles* bzw. *StylesFile* des MIDAS-Seitengenerators enthaltenen Definitionen eine Formatvorlage erstellen.

<#WARNINGS> hat zur Laufzeit keine Funktion. Während des Entwurfs generiert <#WARNINGS> Warnmeldungen für Probleme, die bei der Erstellung des HTML-Dokuments auftreten. Diese Meldungen werden im Web-Seiteneditor angezeigt.

<#FORMS> generiert den HTML-Code, der von den im Web-Seiteneditor hinzugefügten Komponenten erstellt wird. Der HTML-Code für jede Komponente wird in der in *WebPageItems* festgelegten Reihenfolge angezeigt.

<#SCRIPT> generiert einen Block mit Javascript-Deklarationen. Diese werden in dem HTML-Code verwendet, der von den im Web-Seiteneditor hinzugefügten Komponenten generiert wird.

Sie können die Standardvorlage ersetzen, indem Sie den Wert von *HTMLDoc* oder der Eigenschaft *HTMLFile* ändern. Die angepaßte HTML-Vorlage kann alle HTML-transparenten Tags enthalten, die in der Standardvorlage vorhanden sind. Der MIDAS-Seitengenerator übersetzt diese Tags automatisch, wenn die Methode *Content* aufgerufen wird. Die drei folgenden Tags werden vom MIDAS-Seitengenerator ebenfalls automatisch übersetzt:

`<#BODYELEMENTS>` wird durch den HTML-Code ersetzt, der sich aus den fünf Tags in der Standardvorlage ergibt. Dies ist bei der Erstellung einer Vorlage in einem HTML-Editor sinnvoll, wenn das Standardlayout durch zusätzliche Elemente ergänzt werden soll.

`<#COMPONENT Name=WebKomponentenName>` wird durch den HTML-Code ersetzt, den die Komponente *WebKomponentenName* generiert. Diese Komponente kann im Web-Seiteneditor hinzugefügt worden sein. Es kann sich aber auch um eine Komponente handeln, die die Schnittstelle *IWebContent* unterstützt und denselben Eigentümer wie der MIDAS-Seitengenerator hat.

`<#DATAPACKET XMLBroker=BrokerName>` wird durch das XML-Datenpaket ersetzt, das von dem in *BrokerName* festgelegten XML-Broker übergeben wurde. Im Web-Seiteneditor enthält der vom MIDAS-Seitengenerator erzeugte HTML-Code dieses Tag anstelle des eigentlichen XML-Datenpakets.

Zusätzlich zu diesen Tags kann eine angepaßte Vorlage weitere HTML-transparente Tags enthalten, die Sie definieren. Wenn der MIDAS-Seitengenerator auf ein Tag trifft, das nicht zu den sieben automatisch übersetzten Tags gehört, generiert er das Ereignis *OnHTMLTag*. In einer Behandlungsroutine für dieses Ereignis können Sie dann eigene Übersetzungen implementieren. Weitere Informationen zu HTML-Vorlagen finden Sie unter »HTML-Vorlagen« auf Seite 29-20.

Tip Die im Web-Seiteneditor angezeigten Komponenten generieren statischen Quelltext. Es wird deshalb immer derselbe HTML-Code erzeugt (sofern die in den Datenpaketen enthaltenen Metadaten nicht vom Anwendungsserver geändert werden). Damit dieser Quelltext zur Laufzeit nicht für jede Anforderungsbotschaft dynamisch erstellt werden muß, können Sie den generierten HTML-Code im Web-Seiteneditor kopieren und als Vorlage verwenden. Die Anwendung kann trotzdem Datenpakete dynamisch vom Anwendungsserver abrufen, da im Web-Seiteneditor anstelle des XML-Datenpakets das Tag <#DATAPACKET> angezeigt wird.

Kapitel

15

Provider-Komponenten

Provider-Komponenten (*TDataSetProvider*) stellen den Mechanismus bereit, durch den Client-Datenmengen ihre Daten erhalten (sofern sie keine unstrukturierten Dateien verwenden). Provider sorgen dafür, daß die Daten in Paketen zusammengefaßt und an die Client-Datenmenge gesendet werden, und daß Aktualisierungen eingetragen werden, die von Client-Datenmengen empfangen wurden. Provider-Komponenten befinden sich normalerweise als Teil einer mehrschichtigen Anwendung auf einem Anwendungsserver. Sie können aber auch zur gleichen Anwendung gehören wie die Client-Datenmenge (oder der XML-Broker). Provider arbeiten mit Resolver-Komponenten zusammen, die für das Eintragen der Daten in die Datenbank oder Datenmenge verantwortlich sind.

Provider-Komponenten erledigen den Großteil der Arbeit automatisch. Um einen funktionsfähigen Anwendungsserver zu erstellen, müssen Sie also keinen Quelltext schreiben. Dennoch verfügen Provider-Komponenten über eine Reihe von Ereignissen und Eigenschaften, mit deren Hilfe Ihre Anwendung direkt kontrollieren kann, welche Daten für die Clients in Paketen zusammengefaßt werden und wie auf Client-Anfragen reagiert werden soll.

Dieses Kapitel beschreibt, wie Sie mit Hilfe einer Provider-Komponente die Interaktion mit Client-Anwendungen steuern können.

Die Datenquelle festlegen

Wenn Sie eine Provider-Komponente verwenden, müssen Sie angeben, aus welcher Datenmenge die Daten stammen, die in Paketen zusammengefaßt werden sollen. Zu diesem Zweck weisen Sie der Eigenschaft *DataSet* des Providers den Namen der gewünschten Datenmenge zu. Zur Entwurfszeit können Sie die Datenmenge im Objektinspektor in der Dropdown-Liste der Eigenschaft *DataSet* auswählen.

TDataSetProvider funktioniert mit allen Datenmengen, die die Schnittstelle *IProvider-Support* unterstützen. Da diese Schnittstelle von *TDataSet* eingeführt wird, steht sie allen Datenmengen zur Verfügung. Die in *TDataSet* implementierten *IProvider-*

Support-Methoden sind allerdings nur Stub-Komponenten, die entweder nichts tun oder Exceptions auslösen. Die meisten Datenmengenklassen, die zum Lieferumfang von Delphi gehören (BDE-Datenmengen, ADO-Datenmengen, Client-Datenmengen und InterBase-Direkt-Komponenten), überschreiben diese Methoden, um die Schnittstelle *IProviderSupport* auf die gewünschte Weise zu implementieren.

Hinweis Da der Provider auf die zur Datenmenge gehörige Schnittstelle zurückgreift, gibt es keine speziellen Abhängigkeiten vom Mechanismus des Datenzugriffs (BDE, DBOLE usw.). Diese Abhängigkeiten betreffen ausschließlich die verwendete Datenmenge.

Komponentenentwickler, die eigene Nachkommen von *TDataSet* erstellen, müssen alle entsprechenden *IProviderSupport*-Methoden überschreiben, wenn ihre Datenmengen in einem Anwendungsserver genutzt werden sollen. Wenn der Provider die Datenpakete nur zum Lesen bereitstellt (also keine Aktualisierungen eingetragen werden), reichen möglicherweise auch die in *TDataSet* implementierten *IProvider-Support*-Methoden aus.

Aktualisierungen eintragen

Wenn *TDataSetProvider*-Komponenten Aktualisierungen eintragen und Aktualisierungsfehler beheben, kommunizieren sie mit dem Datenbankserver standardmäßig über dynamisch generierte SQL-Anweisungen. Das hat den Vorteil, daß Ihre Server-Anwendung Aktualisierungen nicht zweimal eintragen muß (zuerst in der Datenmenge, dann auf dem Remote-Server).

Dieses Vorgehen ist jedoch nicht immer erwünscht. Es kann beispielsweise sein, daß Sie einige Ereignisse der Datenmengenkomponente individuell verwenden wollen, oder daß die Datenmenge keine SQL-Anweisungen unterstützt (wenn es sich z. B. um eine *TClientDataSet*-Komponente handelt).

Bei *TDataSetProvider* können Sie mit Hilfe der Eigenschaft *ResolveToDataSet* selbst entscheiden, ob Aktualisierungen per SQL auf dem Datenbankserver eingetragen werden sollen. Wenn die Eigenschaft *True* ist, werden Aktualisierungen in die Datenmenge eingetragen, andernfalls auf dem zugrundeliegenden Datenbankserver.

Datenpakete zusammenstellen

Es gibt folgende Möglichkeiten, Datenpakete zusammenzustellen, die vom Client gesendet oder empfangen werden:

- Felder für Datenpakete festlegen
- Optionen für Datenpakete einstellen
- Datenpaketen benutzerdefinierte Daten hinzufügen

Felder für Datenpakete festlegen

Um festzulegen, welche Felder in ein Datenpaket aufgenommen werden, erstellen Sie in der Datenmenge, aus der der Provider die Pakete zusammenstellt, persistente Felder. Der Provider verwendet dann nur diese Felder. Felder, deren Werte dynamisch auf dem Server generiert werden (z. B. berechnete Felder oder Lookup-Felder), können zwar ebenfalls in die Pakete aufgenommen werden, sind jedoch für die empfangenden Client-Datenmenge statische Nur-Lesen-Felder. Informationen über das Erstellen persistenter Felder finden Sie unter »Persistente Felder erstellen« auf Seite 19-6.

Wenn die Client-Datenmenge die Daten bearbeiten und Aktualisierungen am Anwendungsserver eintragen soll, müssen Sie ausreichend Felder bereitstellen, damit sich im Datenpaket keine doppelten Datensätze ergeben. Andernfalls kann beim Eintragen der Aktualisierungen nicht mehr festgestellt werden, welcher Datensatz aktualisiert werden soll. Um zu verhindern, daß die Datenmenge Felder anzeigt oder ändert, die nur der Sicherstellung der Eindeutigkeit dienen, setzen Sie in der Eigenschaft *ProviderFlags* dieser Felder das Flag *pfHidden*.

Hinweis Das Vermeiden doppelter Datensätze durch eine ausreichende Anzahl Felder empfiehlt sich auch für Abfragen auf dem Anwendungsserver. Die Abfrage sollte so viele Felder enthalten, daß die Eindeutigkeit der Datensätze auch dann sichergestellt ist, wenn Ihre Anwendung nicht alle Felder verwendet.

Optionen für Datenpakete einstellen

Mit Hilfe der Eigenschaft *Options* der Provider-Komponente können Sie festlegen, ob BLOB-Felder oder verschachtelte Detailtabellen gesendet werden, ob Eigenschaften für die Anzeige der Felder enthalten sind, welche Aktualisierungsarten erlaubt sind usw. Die folgende Tabelle enthält die möglichen Werte der Eigenschaft *Options*.

Tabelle 15.1 Provider-Optionen

Wert	Beschreibung
poFetchBlobsOnDemand	Das Datenpaket enthält keine BLOB-Felder. Client-Anwendungen müssen die benötigten Werte bei Bedarf explizit anfordern. Wenn die Eigenschaft *FetchOnDemand* der Client-Datenmenge *True* ist, geschieht dies automatisch. Andernfalls verwendet die Client-Anwendung die Methode *FetchBlobs* der Client-Datenmenge, um BLOB-Daten abzurufen.
PoFetchDetailsOnDemand	Wenn der Provider die Haupttabelle einer Haupt/Detailbeziehung darstellt, enthält das Datenpaket keine verschachtelten Detailwerte. Client-Anwendungen müssen die benötigten Werte bei Bedarf explizit anfordern. Wenn die Eigenschaft *FetchOnDemand* der Client-Datenmenge *True* ist, geschieht dies automatisch. Andernfalls verwendet die Client-Anwendung die Methode *FetchDetails* der Client-Datenmenge, um verschachtelte Detailwerte abzurufen.
PoIncFieldProps	Das Datenpaket enthält die folgenden Feldeigenschaften (falls vorhanden): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues*.

Tabelle 15.1 Provider-Optionen (Fortsetzung)

Wert	Beschreibung
PoCascadeDeletes	Wenn der Provider die Haupttabelle einer Haupt/Detailbeziehung darstellt, werden Detaildatensätze vom Server automatisch gelöscht, sobald ein Hauptdatensatz gelöscht wird. Damit sich diese Option auswirken kann, muß der Datenbankserver so konfiguriert sein, daß er im Rahmen der referentiellen Integrität mehrstufige Löschvorgänge erlaubt.
PoCascadeUpdates	Wenn der Provider die Haupttabelle einer Haupt/Detailbeziehung bereitstellt, werden Schlüsselwerte in Detailtabellen automatisch aktualisiert, sobald sich der entsprechende Wert in einem Hauptdatensatz ändert. Damit sich diese Option auswirken kann, muß der Datenbankserver so konfiguriert sein, daß er im Rahmen der referentiellen Integrität mehrstufige Aktualisierungen erlaubt.
PoReadOnly	Die Client-Datenmenge kann dem Provider keine Aktualisierungen übergeben.
PoAllowMultiRecord-Updates	Bei einer Aktualisierung ändern sich manchmal mehrere Datensätze der zugrundeliegenden Datenmenge (mögliche Gründe sind Trigger, referentielle Integrität, benutzerdefinierte SQL-Statements usw.). Wenn ein Fehler auftritt, haben Sie über die Ereignisbehandlungsroutinen nur Zugriff auf den aktualisierten Datensatz, nicht aber auf die Datensätze, die als Folge der Aktualisierung geändert wurden.
PoDisableEdits	Clients können keine Daten bearbeiten. Versucht ein Client, ein Feld zu bearbeiten, wird eine Exception ausgelöst (der Client hat aber nach wie vor die Möglichkeit, Datensätze einzufügen oder zu löschen).
PoDisableInserts	Clients können keine neuen Datensätze einfügen. Versucht ein Client, einen Datensatz einzufügen, wird eine Exception ausgelöst (der Client hat aber nach wie vor die Möglichkeit, Datensätze zu löschen oder zu bearbeiten).
poDisableDeletes	Clients können keine Datensätze löschen. Versucht ein Client, einen Datensatz zu löschen, wird eine Exception ausgelöst (der Client hat aber nach wie vor die Möglichkeit, Datensätze einzufügen oder zu ändern).
poNoReset	Clients können den Provider nicht veranlassen, den Datensatzzeiger auf den ersten Datensatz zu positionieren, bevor er Daten bereitstellt.
PoAutoRefresh	Der Provider frischt die Client-Datenmenge mit aktuellen Werten auf, wenn Aktualisierungen eingetragen werden.
PoPropogateChanges	Änderungen, die der Server während des Aktualisierungsvorgangs an Datensätzen vornimmt, werden an den Client zurückgesendet und in die Client-Datenmenge eingetragen.
PoAllowCommand-Text	Der Client kann die SQL-Anweisung der Datenmenge oder den Namen der repräsentierten Tabelle bzw. Stored Procedure überschreiben.

Datenpaketen benutzerdefinierte Daten hinzufügen

Mit Hilfe des Ereignisses *OnGetDataSetProperties* können Provider den Datenpaketen anwendungsspezifische Daten hinzufügen. Diese Daten werden als *OleVariant* übergeben und unter dem von Ihnen angegebenen Namen abgelegt. Mit Hilfe der Methode *GetOptionalParam* können die Datenmengen der Client-Anwendung die Daten abrufen. Dabei können Sie auch festlegen, daß die Daten in die Delta-Pakete aufgenommen werden, die die Client-Datenmenge sendet, wenn sie Datensätze aktualisiert. In diesem Fall erhält die Client-Anwendung keine Kenntnis von den Daten,

während der Server die Möglichkeit hat, eine entsprechende Botschaft an sich selbst zu senden.

Wenn Sie in der Ereignisbehandlungsroutine für *OnGetDataSetProperties* benutzerdefinierte Daten hinzufügen, wird jedes einzelne Attribut (manchmal auch als optionaler Parameter bezeichnet) in einem varianten Array angegeben, das drei Elemente enthält: den Namen (ein String), den Wert (eine Variante) und ein Boolesches Flag, das angibt, ob die Daten in Delta-Pakete aufgenommen werden, wenn der Client Aktualisierungen einträgt. Um mehrere Attribute hinzuzufügen, erstellen Sie ein variantes Array, das aus varianten Arrays besteht. Die folgende Ereignisbehandlungsroutine für *OnGetDataSetProperties* sendet beispielsweise zwei Werte: den Zeitpunkt der Bereitstellung und die Gesamtzahl der Datensätze in der Ausgangsdatenmenge. Wenn Clients Aktualisierungen eintragen, wird nur der Zeitpunkt der Bereitstellung zurückgegeben.

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject;
DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

Wenn der Client Aktualisierungen einträgt, kann der Zeitpunkt, zu dem die ursprünglichen Daten bereitgestellt wurden, in der Behandlungsroutine für das Ereignis *OnUpdateData* des Providers abgerufen werden:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

Auf Datenanforderungen des Client reagieren

In den meisten mehrschichtigen Anwendungen werden Datenanforderungen von Clients automatisch bearbeitet. Eine Client-Datenmenge fordert ein Datenpaket an, indem sie (auf indirektem Weg über die Schnittstelle *IAppServer*) die Methode *GetRecords* aufruft. Der Provider reagiert automatisch, indem er die Daten aus der betreffenden Datenmenge abruft, ein Datenpaket generiert und dieses an den Client sendet.

Nachdem das Paket zusammengestellt wurde, hat der Provider die Möglichkeit, die Daten zu bearbeiten, bevor das Paket an den Client gesendet wird. So können beispielsweise sensible Daten verschlüsselt oder bestimmte Datensätze aus dem Paket entfernt werden (z. B. die Benutzerrolle in einer MTS-Anwendung).

Um ein Datenpaket zu bearbeiten, bevor es an den Client gesendet wird, benötigen Sie eine Behandlungsroutine für das Ereignis *OnGetData* . Dabei wird das Datenpaket

in Form einer Client-Datenmenge als Parameter übergeben. Mit den Methoden dieser Client-Datenmenge können Sie die Daten bearbeiten, bevor sie an den Client gesendet werden.

Wie bei allen Methodenaufrufen, die über die Schnittstelle *IAppServer* erfolgen, hat der Provider auch hier die Möglichkeit, vor und nach dem Aufruf von *GetRecords* persistente Statusinformationen mit der Client-Anwendung auszutauschen. Die erforderliche Kommunikation findet in den Ereignisbehandlungsroutinen für *BeforeGetRecords* und *AfterGetRecords* statt. Eine detaillierte Beschreibung persistenter Statusinformationen in Anwendungsservern finden Sie unter »Statusinformationen in Remote-Datenmodulen unterstützen« auf Seite 14-30.

Auf Aktualisierungsanforderungen des Client reagieren

Die Aktualisierungen, die ein Provider an einem Datensatz vornimmt, basieren auf einem *Delta-Datenpaket*, das er von einer Client-Anwendung empfängt. Der Client fordert Aktualisierungen an, indem er (indirekt über die Schnittstelle *IAppServer*) die Methode *ApplyUpdates* aufruft.

Wie bei allen Methodenaufrufen, die über die Schnittstelle *IAppServer* erfolgen, hat der Provider auch hier die Möglichkeit, vor und nach dem Aufruf von *ApplyUpdates* persistente Statusinformationen mit der Client-Anwendung auszutauschen. Die erforderliche Kommunikation findet in den Ereignisbehandlungsroutinen für *BeforeApplyUpdates* und *AfterApplyUpdates* statt. Eine detaillierte Beschreibung persistenter Statusinformationen in Anwendungsservern finden Sie unter »Statusinformationen in Remote-Datenmodulen unterstützen« auf Seite 14-30.

Wenn ein Provider eine Aktualisierungsanforderung erhält, generiert er ein *OnUpdateData*-Ereignis. In der Behandlungsroutine für dieses Ereignis können Sie entweder das Delta-Paket bearbeiten , bevor es in die Datenmenge geschrieben wird, oder steuern, wie die Aktualisierungen eingetragen werden. Nach dem Ereignis *OnUpdateData* verwendet der Provider die zugeordnete Resolver-Komponente, um die Änderungen in die Datenbank einzutragen.

Die Resolver-Komponente aktualisiert der Reihe nach alle Datensätze. Vor jeder Aktualisierung generiert der Resolver für den Provider das Ereignis *BeforeUpdateRecord*. In der entsprechenden Behandlungsroutine können Sie die Aktualisierungen überwachen, bevor sie eingetragen werden. Wenn dabei ein Fehler auftritt, ruft der Resolver die Behandlungsroutine des Providers für das Ereignis *OnUpdateError* auf, um den Fehler zu beheben. Ein Fehler tritt normalerweise auf, wenn die Änderung eine Server-Beschränkung verletzt. Eine weitere mögliche Fehlerursache besteht darin, daß der Datensatz von einer anderen Anwendung geändert wurde, nachdem er von der Client-Anwendung abgerufen wurde, aber bevor der Client die Aktualisierung angefordert hat.

Aktualisierungsfehler können entweder vom Anwendungsserver oder vom Client bearbeitet werden. Der Anwendungsserver sollte alle Aktualisierungsfehler behandeln, deren Behebung keine Interaktion mit dem Benutzer erfordert. Wenn der Anwendungsserver eine Fehlerbedingung nicht beseitigen kann, legt er eine temporäre Kopie des betreffenden Datensatzes an. Ist die Bearbeitung der Datensätze beendet,

gibt der Anwendungsserver die Anzahl der aufgetretenen Fehler an die Client-Datenmenge zurück und kopiert die nicht bereinigten Datensätze in ein Ergebnis-Datenpaket, das zur weiteren Bearbeitung an den Client zurückgegeben wird.

Die Datenmenge mit den Aktualisierungen wird in Form einer Client-Datenmenge an die Behandlungsroutinen aller Provider-Ereignisse übergeben. Wenn eine Ereignisbehandlungsroutine nur für bestimmte Aktualisierungsarten zuständig ist, können Sie die Datenmenge anhand des Aktualisierungsstatus der Datensätze filtern. Die Routine muß dann nur die relevanten Datensätze verarbeiten. Setzen Sie zu diesem Zweck die Eigenschaft *StatusFilter* der Client-Datenmenge auf den geeigneten Wert.

Hinweis Betreffen die Aktualisierungen eine Datenmenge mit mehreren Tabellen, muß die Anwendung zusätzliche Vorkehrungen treffen. Weitere Informationen dazu finden Sie unter »Aktualisierungen in Datenmengen mit mehreren Tabellen« auf Seite 15-10.

Delta-Pakete vor dem Aktualisieren der Datenbank bearbeiten

Bevor der Provider Aktualisierungen in die Datenbank einträgt, generiert er das Ereignis *OnUpdateData*. Der entsprechenden Behandlungsroutine wird eine Kopie des Delta-Pakets als Client-Datenmenge übergeben.

In der Behandlungsroutine für *OnUpdateData* können Sie alle Eigenschaften und Methoden der Client-Datenmenge verwenden, um das Delta-Paket zu bearbeiten, bevor es in die Datenmenge geschrieben wird. Eine besonders nützliche Eigenschaft ist *UpdateStatus*. Sie gibt an, auf welche Weise sich der aktuelle Datensatz im Delta-Paket geändert hat. In der folgenden Tabelle sind die möglichen Werte für *UpdateStatus* aufgeführt:

Tabelle 15.2 Werte für UpdateStatus

Wert	Bedeutung
usUnmodified	Der Datensatz wurde nicht geändert.
UsModified	Der Datensatz wurde geändert.
UsInserted	Der Datensatz wurde eingefügt.
UsDeleted	Der Datensatz wurde gelöscht.

Die folgende Behandlungsroutine für *OnUpdateData* weist beispielsweise jedem Datensatz, der neu in die Datenbank eingefügt wird, das aktuelle Datum zu:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
```

```
          FieldByName('DateCreated').AsDateTime := Date;
          Post;
        end;
        Next;
      end;
end;
```

Die Art der Aktualisierung steuern

Mit dem Ereignis *OnUpdateData* kann der Provider festlegen, wie die Datensätze des Delta-Pakets in die Datenbank geschrieben werden.

Standardmäßig werden Änderungen im Delta-Paket mit Hilfe automatisch generierter SQL-Anweisungen (UPDATE, INSERT oder DELETE) in die Datenbank geschrieben. Ein Beispiel:

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Wenn Sie nichts anderes festlegen, werden alle Felder der Datensätze des Delta-Pakets in die UPDATE- und WHERE-Klauseln aufgenommen. Sie können aber einzelne Felder ausschließen. Eine Möglichkeit besteht z. B. darin, der Eigenschaft *UpdateMode* des Providers einen der folgenden Werte zuzuweisen:

Tabelle 15.3 Werte für UpdateMode

Wert	Bedeutung
upWhereAll	In der WHERE-Klausel werden alle Felder verwendet.
upWhereChanged	Nur Schlüsselfelder und Felder, die geändert wurden, werden verwendet.
upWhereOnly	Nur Schlüsselfelder werden verwendet.

In manchen Fällen ist noch mehr Einflußnahme erforderlich. In der obigen Anweisung können Sie beispielsweise verhindern, daß das Feld EMPNO geändert wird, indem Sie es aus der UPDATE-Klausel ausschließen. Die Felder TITLE und DEPT könnten aus der WHERE-Klausel ausgeschlossen werden, um Aktualisierungskonflikte zu vermeiden, wenn andere Anwendungen die Daten geändert haben. Mit Hilfe der Eigenschaft *ProviderFlags* können Sie festlegen, in welchen Klauseln ein bestimmtes Feld berücksichtigt werden soll. *ProviderFlags* ist eine Menge, die beliebige Werte aus der folgenden Tabelle enthalten kann.

Tabelle 15.4 Werte für ProviderFlags

Wert	Bedeutung
pfInWhere	Das Feld erscheint nicht in der WHERE-Klausel der generierten INSERT-, DELETE- und UPDATE-Anweisungen.
pfInUpdate	Das Feld erscheint nicht in der UPDATE-Klausel der generierten UPDATE-Anweisungen.

Tabelle 15.4 Werte für ProviderFlags (Fortsetzung)

Wert	Bedeutung
pfInKey	Das Feld wird in der WHERE-Klausel einer generierten SELECT-Anweisung verwendet, die bei Aktualisierungsfehlern ausgeführt wird. Diese SELECT-Anweisung versucht, den aktuellen Wert eines geänderten, gelöschten oder eine Schlüsselverletzung verursachenden Datensatzes zu ermitteln, wenn eine Einfügung fehlgeschlagen ist.
pfHidden	Das Feld ist in Datensätzen enthalten, um die Eindeutigkeit sicherzustellen. Der Client kann es jedoch weder anzeigen noch verwenden.

Die folgende Behandlungsroutine für *OnUpdateData* schließt das Feld EMPNO aus der UPDATE-Klausel und die Felder TITLE und DEPT aus der WHERE-Klausel aus:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TClientDataSet);
begin
   with DataSet do
   begin
    FieldByName('EMPNO').UpdateFlags := [ufInUpdate];
    FieldByName('TITLE').UpdateFlags := [ufInWhere];
    FieldByName('DEPT').UpdateFlags := [ufInWhere];
   end;
end;
```

Hinweis Mit der Eigenschaft *UpdateFlags* können Sie Aktualisierungen auch dann steuern, wenn Sie eine Datenmenge aktualisieren, ohne dynamisch generiertes SQL einzusetzen. Auch in diesem Fall legen die Flags fest, über welche Felder Datensätze gesucht und welche Felder aktualisiert werden.

Bestimmte Aktualisierungen überwachen

Unmittelbar vor dem Eintragen einer Aktualisierung empfängt der Provider das Ereignis *BeforeUpdateRecord*. In der Behandlungsroutine für dieses Ereignis können Sie Datensätze bearbeiten, bevor sie eingetragen werden. In ähnlicher Weise wird auch das Ereignis *OnUpdateData* verwendet, mit dem Sie Delta-Pakete bearbeiten können. Der Provider prüft beispielsweise keine BLOB-Felder wie Memos auf Aktualisierungskonflikte. Wenn Sie auch BLOB-Felder untersuchen wollen, verwenden Sie das Ereignis *BeforeUpdateRecord*.

Das Ereignis eignet sich auch dazu, Aktualisierungen einzutragen, zu überwachen oder zurückzuweisen. In der Behandlungsroutine können Sie dem Resolver beispielsweise signalisieren, daß eine Aktualisierung bereits bearbeitet wurde und nicht mehr eingetragen werden soll. Der Resolver übergeht diesen Datensatz, ohne ihn als Aktualisierungsfehler zu werten. Dieses Verfahren bietet z. B. die Möglichkeit, Stored Procedures zu aktualisieren (die nicht automatisch aktualisiert werden können), wobei der Provider alle automatischen Bearbeitungsschritte übergeht, wenn ein Datensatz bereits innerhalb der Behandlungsroutine aktualisiert wurde.

Aktualisierungsfehler beheben

Wenn der Anwendungsserver versucht, einen Datensatz im Delta-Paket zu speichern, und dabei eine Fehlerbedingung auftritt, wird das Ereignis *OnUpdateError* generiert. Kann der Anwendungsserver einen Aktualisierungsfehler nicht beheben, legt er eine temporäre Kopie des betreffenden Datensatzes an. Ist die Bearbeitung der Datensätze beendet, gibt der Anwendungsserver die Anzahl der aufgetretenen Fehler an die Client-Datenmenge zurück und kopiert die nicht bereinigten Datensätze in ein Ergebnis-Datenpaket, das zur weiteren Bearbeitung an den Client zurückgegeben wird.

Durch diesen Mechanismus können Sie alle Aktualisierungsfehler behandeln, die auf dem Anwendungsserver behoben werden können, während in der Client-Anwendung nach wie vor der Benutzer eingreifen kann, um Fehlerbedingungen zu beseitigen.

Die Behandlungsroutine für *OnUpdateError* erhält eine Kopie des Datensatzes, der nicht geändert werden konnte, den Fehlercode der Datenbank und einen Hinweis darauf, ob der Resolver versucht hat, den Datensatz einzufügen, zu löschen oder zu aktualisieren. Der verursachende Datensatz wird in einer Client-Datenmenge zurückgegeben. Verwenden Sie in dieser Datenmenge niemals die Navigationsmethoden. Sie können für jedes Feld der Datenmenge die Eigenschaften *NewValue*, *OldValue* und *CurValue* verwenden, um die Ursache des Problems zu ermitteln, und dann die nötigen Änderungen vornehmen, um den Fehler zu beseitigen. Wenn die Behandlungsroutine für *OnUpdateError* das Problem beheben kann, setzt sie den Parameter *Response* so, daß der berichtigte Datensatz eingetragen wird.

Aktualisierungen in Datenmengen mit mehreren Tabellen

Wenn eine Resolver-Komponente SQL-Anweisungen generiert, die Aktualisierungen direkt auf einem Datenbankserver eintragen, benötigt sie den Namen der Datenbanktabelle, die die entsprechenden Datensätze enthält. Bei Datenmengen wie z. B. *TTable*- oder *TQuery*-Komponenten ist dieser automatisch verfügbar.

Automatische Aktualisierungen werfen jedoch ein Problem auf, wenn der Resolver sie an Daten vornehmen muß, denen eine Stored Procedure mit einer Ergebnismenge oder eine Abfrage über mehrere Tabellen zugrundeliegt. Es fehlt nämlich an einer direkten Möglichkeit, den Namen der Tabelle zu ermitteln, in der die Aktualisierungen eingetragen werden sollen.

Wenn es sich bei der Abfrage oder Stored Procedure um eine BDE-Datenmenge handelt (*TQuery* oder *TStoredProc*), der ein Aktualisierungsobjekt zugeordnet ist, verwendet der Provider das Aktualisierungsobjekt. Gibt es kein Aktualisierungsobjekt, können Sie den Tabellennamen in einer Ereignisbehandlungsroutine für *OnGetTableName* bereitstellen. Sobald der Tabellenname zur Verfügung steht, kann die Resolver-Komponente die erforderlichen SQL-Anweisungen generieren.

Hinweis Die Bereitstellung des Tabellennamens ist nur möglich, wenn das Ziel der Aktualisierung eine einzelne Datenbanktabelle ist (wenn also nur in einer einzigen Tabelle Datensätze aktualisiert werden müssen). Sollte die Aktualisierung Änderungen an meh-

reren Tabellen erforderlich machen, muß sie in einer Behandlungsroutine für das Ereignis *BeforeUpdateRecord* des Providers explizit eingetragen werden. Sobald eine Aktualisierung eingetragen wurde, setzen Sie den Parameter *Applied* der Behandlungsroutine auf *True*, damit der Resolver keinen Fehler generiert.

Auf Client-Ereignisse reagieren

Provider-Komponenten implementieren das allgemeine Ereignis *OnDataRequest*, mit dem Sie eigene Aufrufe generieren können, die der Client an den Provider sendet.

Das Ereignis *OnDataRequest* spielt für die normale Tätigkeit des Providers keine Rolle. Es stellt lediglich eine Möglichkeit für die Clients dar, direkt mit Providern auf dem Anwendungsserver zu kommunizieren. Die Behandlungsroutine erwartet einen Parameter des Typs *OleVariant* und gibt einen Wert des gleichen Typs zurück. Die Verwendung von OleVariant-Werten erlaubt den Austausch nahezu beliebiger Informationen mit dem Provider.

Um ein *OnDataRequest*-Ereignis zu generieren, ruft die Client-Anwendung die Methode *DataRequest* der Client-Datenmenge auf.

Server-Beschränkungen

Die meisten relationalen Datenbanksysteme implementieren für ihre Tabellen Beschränkungen, um die Integrität der Daten zu gewährleisten. Eine Beschränkung ist eine Art Vorschrift, die für Datenwerte in Tabellen und Spalten oder für Beziehungen zwischen Spalten in unterschiedlichen Tabellen gilt. Die meisten relationalen SQL-92-Datenbanken unterstützen beispielsweise folgende Beschränkungen:

- NOT NULL garantiert, daß eine Spalte einen Wert enthält.
- NOT NULL UNIQUE garantiert, daß eine Spalte einen Wert enthält und dieser in der Spalte nur einmal vorkommt.
- CHECK garantiert, daß ein Wert in einem bestimmten Bereich liegt oder zu einer begrenzten Menge möglicher Werte gehört.
- CONSTRAINT ist eine tabellenweite Prüfung, die sich auf mehrere Spalten bezieht.
- PRIMARY KEY kennzeichnet eine oder mehrere Spalten als Primärschlüssel.
- FOREIGN KEY kennzeichnet eine oder mehrere Spalten, die eine andere Tabelle referenzieren.

Hinweis Diese Liste erhebt keinen Anspruch auf Vollständigkeit. Ihr Datenbankserver unterstützt möglicherweise alle oder einige dieser Beschränkungen vollständig oder teilweise. Auf einigen Datenbankservern stehen auch andere Beschränkungen zur Verfügung. Weitere Informationen zu diesem Thema finden Sie in der Server-Dokumentation.

Beschränkungen auf Datenbankservern entsprechen in vieler Hinsicht den Datenprüfungen, die Sie von herkömmlichen Desktop-Datenbankanwendungen kennen. Sie können diese Beschränkungen in mehrschichtigen Datenbankanwendungen nutzen, ohne Sie im Quelltext des Anwendungsservers oder der Client-Anwendung wiederholen zu müssen.

Wenn der Provider mit einer BDE-Datenmenge arbeitet, können Sie mit Hilfe seiner Eigenschaft *Constraints* Server-Beschränkungen auf Daten anwenden, die von Client-Anwendungen empfangen und an diese gesendet werden. Wenn die Eigenschaft *True* ist (Voreinstellung), sind die Beschränkungen auch für die Clients wirksam und beeinflussen alle Versuche, Daten zu aktualisieren.

Wichtig Bevor der Anwendungsserver Informationen über Beschränkungen an die Client-Anwendung weitergibt, muß er sie erst vom Datenbankserver abrufen. Verwenden Sie den SQL-Explorer, um die Beschränkungen und Standardausdrücke des Datenbankservers in das Daten-Dictionary zu importieren. Dort stehen sie automatisch allen BDE-Datenmengen im Anwendungsserver zur Verfügung.

Manchmal sollen die Daten, die an eine Client-Anwendung gesendet werden, keinen Server-Beschränkungen unterliegen. Eine Anwendung, die Daten in Paketen empfängt und lokale Aktualisierungen durchführt, bevor weitere Datensätze abgerufen werden, kann beispielsweise einige Beschränkungen deaktivieren, die wegen der zeitweise unvollständigen Datenmenge Probleme verursachen würden. Um die Weitergabe der Beschränkungen vom Anwendungsserver an eine Client-Datenmenge zu unterbinden, setzen Sie *Constraints* auf *False*. Beachten Sie auch, daß Client-Datenmengen Beschränkungen mit den Methoden *EnableConstraints* und *DisableConstraints* aktivieren und deaktivieren können. Weitere Informationen zu diesem Thema finden Sie unter »Beschränkungen verarbeiten« auf Seite 24-21.

Kapitel

16

Datenbanksitzungen

Sowohl eigenständige Client-Datenbankanwendungen als auch Datenbank-Server können über die Borland Database Engine (BDE) mit Datenbanken kommunizieren. Datenbankverbindungen, Treiber, Cursor, Abfragen usw. einer Anwendung werden im Kontext einer oder mehrerer BDE-Sitzungen verwaltet. Sitzungen isolieren Datenbank-Zugriffsoperationen (z.B. Datenbankverbindungen), ohne daß eine weitere Instanz der Anwendung gestartet werden muß.

In einer Anwendung können BDE-Sitzungen über *TSession*- und *TSessionList*-Komponenten verwaltet werden. Innerhalb einer Anwendung kapselt jede *TSession*-Komponente eine einzelne BDE-Sitzung. Alle Sitzungen einer Anwendung werden von einer einzelnen *TSessionList*-Komponente verwaltet.

Allen Datenbankanwendungen wird automatisch eine Sitzungskomponente mit dem Namen *Session* zugewiesen, welche die Standard-BDE-Sitzung kapselt. Anwendungen können, falls erforderlich, zusätzliche Sitzungskomponenten deklarieren, erzeugen und bearbeiten.

Allen Datenbankanwendungen wird automatisch eine Sitzungslisten-Komponente mit dem Namen *Sessions* zugewiesen, mit der Anwendungen ihre Sitzungskomponenten verwalten können.

Dieses Kapitel beschreibt die Sitzungs- und Sitzungslisten-Komponenten und erklärt, wie mit ihrer Hilfe BDE-Sitzungen in Client-Datenbankanwendungen und Datenbank-Servern gesteuert werden können.

Hinweis Die Standardkomponenten für Sitzungen und Sitzungslisten verfügen über eine Vielzahl von Vorgaben, die von den meisten Anwendungen verwendet werden können. Anwendungen, die mehrere Sitzungen benötigen (z.B. für gleichzeitige Abfragen in einer Datenbank), müssen ihre Sitzungs- und Sitzungslisten-Komponenten möglicherweise ändern.

Mit einer Sitzungskomponente arbeiten

Eine Sitzungskomponente ermöglicht die globale Kontrolle über eine Gruppe von Datenbankverbindungen einer Anwendung. Beim Erzeugen einer Client-Anwendung oder eines Anwendungsservers wird Ihrer Anwendung automatisch eine Sitzungskomponente namens *Session* zugewiesen. Datenbank- und Datenmengen-Komponenten, die Sie Ihrer Anwendung hinzufügen, werden automatisch mit dieser Standardsitzung verbunden. Sie kontrolliert den Zugriff auf kennwortgeschützte Paradox-Dateien und gibt die Verzeichnisse an, die für die gemeinsame Nutzung von Paradox-Dateien in einem Netzwerk benötigt werden. Anwendungen können mit den Eigenschaften, Ereignissen und Methoden einer Sitzung Datenbankverbindungen kontrollieren und auf Paradox-Dateien zugreifen.

Sie können alle Datenbankverbindungen einer Anwendung mit der Standardsitzung überwachen. Sie können aber auch zusätzliche Sitzungskomponenten zur Entwurfszeit hinzufügen oder dynamisch zur Laufzeit erzeugen, um nur einen Teil der Datenbankverbindungen einer Anwendung zu kontrollieren.

Manchmal benötigen Anwendungen zusätzliche Sitzungskomponenten (wenn sie beispielsweise gleichzeitig mehrere Abfragen in derselben Datenbank durchführen). In diesem Fall muß jede der Abfragen in einer eigenen Sitzung ausgeführt werden. Multithread-Datenbankanwendungen benötigen mehrere Sitzungen. Anwendungen, die mehrere Sitzungen verwenden, müssen diese über die Sitzungslisten-Komponente *Sessions* verwalten. Weitere Informationen über die Verwaltung mehrerer Sitzungen finden Sie im Abschnitt »Mehrere Sitzungen verwalten« auf Seite 16-17.

Die Standardsitzung

Alle Datenbankanwendungen enthalten automatisch eine Standardsitzung. Delphi erzeugt bei jedem Start einer Datenbankanwendung eine Standard-Sitzungskomponente mit dem Namen *Session* (deren Eigenschaft *SessionName* den Wert *Default* hat). Die Standardsitzung gibt der Anwendung globale Kontrolle über alle Datenbank-Komponenten, die nicht mit einer anderen Sitzung verbunden sind. Dabei ist es gleichgültig, ob es sich um temporäre Komponenten (die von einer Sitzung zur Laufzeit erzeugt werden, wenn eine Datenmenge geöffnet wird, die nicht mit einer von Ihnen erzeugten Datenbank-Komponente verbunden ist) oder um dauerhafte Komponenten handelt (die von Ihrer Anwendung explizit erzeugt werden). Die Standardsitzung ist zur Entwurfszeit nicht in Ihrem Datenmodul oder Formular sichtbar. Sie können jedoch zur Laufzeit in Ihrem Quelltext auf ihre Eigenschaften und Methoden zugreifen.

Wenn Sie eine Datenbank-Komponente erzeugen, wird sie automatisch mit der Standardsitzung verbunden. Sie brauchen Ihre Datenbank-Komponente nur dann mit einer eigenen Sitzung zu verbinden, wenn die Komponente simultan eine Abfrage in einer bereits von der Standardsitzung geöffneten Datenbank durchführt. Wenn Sie eine Multithread-Datenbankanwendung erstellen, müssen Sie eine zusätzliche Sitzung für jeden weiteren Thread erzeugen.

Für die Standardsitzung brauchen Sie keinerlei Quelltext zu schreiben, es sei denn, Ihre Anwendung muß

- Sitzungseigenschaften ändern, um beispielsweise festzulegen, wann Datenbankverbindungen von automatisch erzeugten Datenbank-Komponenten aufrechterhalten oder beendet werden sollen.
- auf Sitzungsereignisse reagieren, z.B. beim Versuch einer Anwendung, auf eine kennwortgeschützte Paradox-Datei zuzugreifen.
- Sitzungsmethoden ausführen, z.B. zum Öffnen oder Schließen einer Datenbank aufgrund einer Benutzeraktion.
- die Eigenschaft *NetFileDir* setzen, um auf Paradox-Tabellen im Netzwerk zuzugreifen, oder der Eigenschaft *PrivateDir* eine lokale Festplatte zuweisen, um den Durchsatz zu erhöhen.

Unabhängig davon, ob Sie einer Anwendung während des Entwurfs Datenbank-Komponenten hinzufügen oder diese zur Laufzeit dynamisch erzeugen, werden sie automatisch mit der Standardsitzung verbunden, wenn sie nicht explizit einer anderen Sitzung zugewiesen werden. Wenn Ihre Anwendung eine Datenmenge öffnet, die nicht mit einer Datenbank-Komponente verbunden ist, erzeugt Delphi automatisch

- zur Laufzeit eine Datenbank-Komponente,
- verbindet diese mit der Standardsitzung, und
- weist einigen ihrer Schlüsseleigenschaften Anfangswerte zu, die auf den Eigenschaften der Standardsitzung basieren.

Eine der wichtigsten Eigenschaften ist *KeepConnections*. Sie gibt an, wann Datenbankverbindungen von einer Anwendung aufrechterhalten oder beendet werden. Weitere Informationen über *KeepConnections* finden Sie im Abschnitt »Das Standardverhalten von Datenbankverbindungen festlegen« auf Seite 16-6. In diesem Kapitel werden außerdem weitere Eigenschaften, Ereignisse und Methoden der Komponente *TSession* beschrieben, die für die Standardsitzung und für die zusätzlichen Sitzungen verwendet werden, die Sie in Ihrer Anwendung erzeugen.

Zusätzliche Sitzungen erzeugen

Anstatt die Standardsitzung zu verwenden, können Sie auch Sitzungen erzeugen. Zur Entwurfszeit können Sie einem Datenmodul (oder Formular) zusätzliche Sitzungen hinzufügen, ihre Eigenschaften im Objektinspektor setzen, ihnen Ereignisbehandlungsroutinen zuweisen und den Quelltext schreiben, der ihre Methoden aufruft. Sie können auch zur Laufzeit Sitzungen erzeugen, ihre Eigenschaften setzen und ihre Methoden aufrufen. Dieser Abschnitt beschreibt, wie Sitzungen zur Laufzeit erzeugt und gelöscht werden.

Hinweis Beachten Sie, daß zusätzliche Sitzungen nur benötigt werden, wenn eine Anwendung gleichzeitig mehrere Abfragen über eine Datenbank durchführt oder wenn es sich um eine Multithread-Anwendung handelt.

So können Sie eine Sitzungskomponente zur Laufzeit dynamisch erzeugen:

1 Deklarieren Sie einen Zeiger auf eine *TSession*-Variable.

2 Instantiieren Sie eine neue Sitzung durch Aufrufen des Konstruktors *Create*. Der Konstruktor erzeugt für die Sitzung eine leere Liste von Datenbank-Komponenten und eine leere Liste von BDE-Callbacks, setzt die Eigenschaft *KeepConnections* auf *True* und fügt die Sitzung der Liste hinzu, die von der Sitzungslisten-Komponente der Anwendung verwaltet wird.

3 Setzen Sie die Eigenschaft *SessionName* der neuen Sitzung auf einen eindeutigen Namen. Über diese Eigenschaft werden Datenbank-Komponenten mit der Sitzung verbunden. Weitere Informationen über *SessionName* finden Sie im Abschnitt »Sitzungsnamen zuweisen« auf Seite 16-4.

4 Aktivieren Sie die Sitzung, und passen Sie gegebenenfalls die Eigenschaften an.

Hinweis Entfernen Sie niemals die Standardsitzung.

Sitzungen können auch mit der Methode *OpenSession* von *TSessionList* erzeugt und geöffnet werden. Die Verwendung von *OpenSession* ist sicherer als das Aufrufen von *Create*, da *OpenSession* eine Sitzung nur dann erzeugt, wenn sie nicht bereits vorhanden ist. Weitere Informationen über *OpenSession* finden Sie im Abschnitt »Mehrere Sitzungen verwalten« auf Seite 16-17.

Der folgende Quelltext erzeugt eine neue Sitzungskomponente, weist ihr einen Namen zu und öffnet die Sitzung für nachfolgende Datenbankoperationen (im Beispiel nicht aufgeführt). Am Ende wird sie durch einen Aufruf der Methode *Free* freigegeben.

```
var
   SecondSession: TSession;
begin
   SecondSession := TSession.Create;
   with SecondSession do
    try
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
end;
```

Sitzungsnamen zuweisen

Über die Eigenschaft *SessionName* können Datenbanken und Datenmengen mit einer Sitzung verbunden werden. Bei der Standardsitzung hat *SessionName* den Wert *Default*. Bei jeder weiteren Sitzungskomponente, die Sie erzeugen, müssen Sie der Eigenschaft *SessionName* einen eindeutigen Wert zuweisen.

Datenbank- und Datenmengenkomponenten verwenden für die Eigenschaft *SessionName* den entsprechenden Wert der Sitzungskomponente. Wenn Sie für die Eigenschaft *SessionName* keinen Wert angeben, wird ihr automatisch der entsprechende Wert der Standardsitzung zugewiesen. Sie können der Eigenschaft auch einen Wert

zuweisen, der mit der Eigenschaft *SessionName* einer von Ihnen erzeugten Sitzungskomponente übereinstimmt.

Details über die Komponente *TSessionList* (und über die Standardsitzung *Sessions*) finden Sie unter »Mehrere Sitzungen verwalten« auf Seite 16-17.

Das folgende Quelltextbeispiel erzeugt mit der Methode *OpenSession* der Standardsitzung *Sessions* eine neue Sitzungskomponente, setzt *SessionName* auf *InterBaseSession*, aktiviert die Sitzung und weist sie der Datenbank-Komponente *Database1* zu:

```
var
   IBSession: TSession;
   ...
begin
   IBSession := Sessions.OpenSession('InterBaseSession');
   Database1.SessionName := 'InterBaseSession';
end;
```

Eine Sitzung aktivieren

Active ist eine Boolesche Eigenschaft, die angibt, ob die mit einer Sitzung verbundenen Datenbank- und Datenmengenkomponenten geöffnet sind. Mit *Active* können Sie den aktuellen Status der Datenbank- und Datenmengenverbindungen einer Sitzung ermitteln oder ändern.

Überprüfen Sie *Active*, um den aktuellen Status einer Sitzung zu ermitteln. Wenn *Active False* ist (Standardwert), sind alle mit der Sitzung verbundenen Datenbanken und Datenmengen geschlossen, andernfalls geöffnet.

Wenn Sie *Active* auf *True* setzen, wird das Ereignis *OnStartup* der Sitzung ausgelöst. Außerdem werden die Eigenschaften *NetFileDir* und *PrivateDir* (falls sie einen Wert enthalten) sowie die Eigenschaft *ConfigMode* gesetzt. Sie können eine Ereignisbehandlungsroutine für *OnStartup* schreiben, um beim Start der Sitzung bestimmte Aktionen durchzuführen. Details über *OnStartup* finden Sie im Abschnitt »Kennwortgeschützte Paradox- und dBase-Tabellen« auf Seite 16-15. Die Eigenschaften *NetFileDir* und *PrivateDir* werden nur bei Paradox-Tabellen verwendet. Details dazu finden Sie im Abschnitt »Das Verzeichnis der Steuerdatei festlegen« auf Seite 16-14 und »Ein Verzeichnis für temporäre Dateien festlegen« auf Seite 16-14. *ConfigMode* gibt an, wie die BDE Aliase verwaltet, die im Kontext der Sitzung erzeugt wurden. Details über *ConfigMode* finden Sie im Abschnitt »Die Sichtbarkeit von Aliasen festlegen« auf Seite 16-11. Details über das Öffnen von Datenbank-Komponenten in einer Sitzung finden Sie im Abschnitt »Datenbankverbindungen erzeugen, öffnen und schließen« auf Seite 16-7.

Mit der Methode *OpenDatabase* können Sie die Datenbankverbindungen öffnen, nachdem Sie eine Sitzung aktiviert haben.

Wenn Sie bei Sitzungskomponenten, die Sie in ein Datenmodul oder Formular einfügen, *Active* auf *False* setzen, werden eventuell geöffnete Datenbanken oder Datenmengen geschlossen. Zur Laufzeit können bestimmte Ereignisse ausgelöst werden, wenn Datenbanken und Datenmengen geschlossen werden.

Hinweis Zur Entwurfszeit können Sie *Active* für die Standardsitzung nicht auf *False* setzen. Die Standardsitzung zur Laufzeit zu schließen, ist zwar möglich, aber nicht ratsam.

Bei von Ihnen erzeugten Sitzungskomponenten können Sie den Objektinspektor verwenden, um *Active* zur Entwurfszeit auf *False* zu setzen. Dadurch werden die Datenbankverbindungen deaktiviert, und die Eigenschaften der Sitzung können geändert werden. Auf diese Weise werden zur Entwurfszeit keine störenden Exceptions ausgelöst, wenn eine Remote-Datenbank kurzzeitig nicht verfügbar ist.

Mit den Methoden *Open* und *Close* können Sitzungen mit Ausnahme der Standardsitzung zur Laufzeit aktiviert oder deaktiviert werden. Die folgende Anweisung schließt beispielsweise alle offenen Datenbanken und Datenmengen einer Sitzung:

```
Session1.Close;
```

Diese Anweisung setzt die Eigenschaft *Active* von *Session1* auf *False*. Wenn die Eigenschaft *Active* einer Sitzung *False* ist, setzt ein nachfolgender Versuch der Anwendung, eine Datenbank oder Datenmenge zu öffnen, sie wieder auf *True* und ruft die Ereignisbehandlungsroutine für *OnStartup* auf, falls sie existiert. Sie können eine Sitzung in Ihrem Quelltext auch zur Laufzeit aktivieren. Die folgende Anweisung aktiviert *Session1*:

```
Session1.Open;
```

Hinweis Wenn eine Sitzung aktiv ist, können Sie einzelne Datenbankverbindungen öffnen und schließen. Details dazu finden Sie im Abschnitt »Eine einzelne Datenbankverbindung schließen« auf Seite 16-7.

Den Start der Sitzung anpassen

Sie können das Verhalten beim Start einer Sitzung anpassen, indem Sie eine Ereignisbehandlungsroutine für *OnStartup* schreiben. *OnStartup* wird ausgelöst, wenn eine Sitzung aktiviert wird. Eine Sitzung wird aktiviert, wenn sie erzeugt wird oder sobald ihre Eigenschaft *Active* von *False* auf *True* geändert wird (wenn beispielsweise eine Datenbank oder Datenmenge, die mit einer Sitzung verbunden ist, geöffnet wird und keine anderen Datenbanken oder Datenmengen geöffnet sind).

Das Standardverhalten von Datenbankverbindungen festlegen

KeepConnections enthält den Standardwert der Eigenschaft *KeepConnection* von temporären Datenbank-Komponenten, die zur Laufzeit erzeugt werden. *KeepConnection* legt fest, was mit der Datenbankverbindung einer Datenbank-Komponente geschehen soll, wenn alle Datenmengen geschlossen werden. Bei *True* (Vorgabe) bleibt eine konstante oder dauerhafte Datenbankverbindung auch dann bestehen, wenn keine Datenmenge aktiv ist. Bei *False* wird eine Datenbankverbindung geschlossen, sobald alle ihre Datenmengen geschlossen werden.

Hinweis Die Verbindungsdauer von Datenbank-Komponenten, die Sie explizit in ein Datenmodul oder Formular einfügen, wird von der Eigenschaft *KeepConnection* der Datenbank-Komponente bestimmt. Wenn die Einstellungen divergieren, hat die Eigenschaft *KeepConnection* einer Datenbank-Komponente Vorrang vor der Eigenschaft

KeepConnections der Sitzung. Weitere Informationen über mehrere Datenbankverbindungen innerhalb einer Sitzung finden Sie im Abschnitt »Datenbankverbindungen erzeugen, öffnen und schließen« auf Seite 16-7.

Bei Anwendungen, welche die mit einer Datenbank auf einem Remote-Server verbundenen Datenmengen häufig öffnen und schließen, sollte *KeepConnections* immer *True* sein. Dies reduziert den Netzwerkverkehr und beschleunigt den Datenzugriff, da jede Verbindung nur einmal pro Sitzung geöffnet und geschlossen werden muß. Andernfalls muß sich die Anwendung immer wieder bei der Datenbank an- oder abmelden, wenn eine Verbindung geöffnet oder geschlossen wird.

Hinweis Auch wenn *KeepConnections True* ist, können Sie alle inaktiven Datenbankverbindungen jederzeit mit der Methode *DropConnections* schließen. Weitere Informationen über *DropConnections* finden Sie im Abschnitt »Temporäre Datenbankverbindungen trennen« auf Seite 16-8.

Die folgende Anweisung beendet beispielsweise die inaktiven Verbindungen der Standardsitzung:

```
Session.DropConnections;
```

Datenbankverbindungen erzeugen, öffnen und schließen

Mit der Methode *OpenDatabase* können Sie in einer Sitzung eine Datenbankverbindung öffnen. *OpenDatabase* erwartet als Parameter den Namen der zu öffnenden Datenbank. Der Name ist entweder ein BDE-Alias oder der Name einer Datenbank-Komponente. Bei Paradox- oder dBASE-Tabellen kann der Name auch ein Pfadname sein. Die folgende Anweisung verwendet beispielsweise die Standardsitzung und versucht, eine Datenbankverbindung zu öffnen, auf die der Alias DBDEMOS zeigt:

```
var
 DBDemosDatabase: TDatabase;
begin
 DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
 ...
```

OpenDatabase aktiviert seine eigene Sitzung, wenn diese nicht bereits aktiv ist, und stellt dann fest, ob der angegebene Datenbankname mit der Eigenschaft *DatabaseName* einer Datenbank-Komponente der Sitzung übereinstimmt. Wenn keine Übereinstimmung gefunden wird, erzeugt *OpenDatabase* eine temporäre Datenbank-Komponente mit dem angegebenen Namen. Mit jedem Aufruf von *OpenDatabase* erhöht sich ein Referenzzähler für die Datenbank um 1. Die Datenbank ist geöffnet, solange dieser Wert größer als 0 ist. Anschließend ruft *OpenDatabase* die Methode *Open* der Datenbank-Komponente auf, um eine Verbindung zum Server herzustellen.

Eine einzelne Datenbankverbindung schließen

Mit der Methode *CloseDatabase* können Sie einzelne Datenbankverbindungen schließen. Mit der Methode *Close* können Sie alle Verbindungen einer Sitzung schließen. Wenn Sie *CloseDatabase* aufrufen, verringert sich der Referenzzähler für die Datenbank um 1. Bei einem Wert von 0 gibt es in der Sitzung keine gegenwärtig aktiven Datenbank-Komponenten, d.h., die Sitzung ist geschlossen. *CloseDatabase* erwartet

als Parameter die zu schließende Datenbank. Die folgende Anweisung schließt beispielsweise die Datenbankverbindung, die im Beispiel des vorangegangenen Abschnitts geöffnet wurde:

```
Session.CloseDatabase(DBDemosDatabase);
```

Wenn der angegebene Datenbankname mit einer temporären Datenbank-Komponente verbunden und die Eigenschaft *KeepConnections* der Sitzung *False* ist, wird die temporäre Datenbank-Komponente freigegeben und die Verbindung getrennt.

Hinweis Wenn *KeepConnections False* ist, werden temporäre Datenbank-Komponenten automatisch geschlossen und freigegeben, sobald die letzte mit der Datenbank-Komponente verbundene Datenmenge geschlossen wird. Eine Anwendung kann jederzeit *CloseDatabase* aufrufen, um das Schließen der Datenbank zu erzwingen. Mit der Methode *Close* der Datenbank-Komponente können Sie temporäre Datenbank-Komponenten auch dann freigeben, wenn *KeepConnections True* ist. Rufen Sie anschließend die Methode *DropConnections* auf.

Wenn die Datenbank-Komponente dauerhaft ist (was bedeutet, daß sie von der Anwendung deklariert und instantiiert wurde) und die Eigenschaft *KeepConnections* der Sitzung *False* ist, ruft *CloseDatabase* die Methode *Close* der Datenbank-Komponente auf, um die Verbindung zu schließen.

Hinweis Mit dem Aufruf von *CloseDatabase* für eine dauerhafte Datenbank-Komponente wird die Verbindung nicht wirklich geschlossen. Rufen Sie die Methode *Close* der Datenbank-Komponente direkt auf, um die Verbindung zu schließen.

Alle Datenbankverbindungen schließen

Es gibt zwei Möglichkeiten, alle Datenbankverbindungen zu schließen:

- Setzen Sie die Eigenschaft *Active* der Sitzung auf *False*.
- Rufen Sie die Methode *Close* der Sitzung auf.

Wenn Sie Active auf *False* setzen, ruft Delphi automatisch *Close* auf. Die Methode trennt die Verbindung mit allen aktiven Datenbanken, indem sie temporäre Datenbank-Komponenten freigibt und die Methode *Close* jeder dauerhaften Datenbank-Komponente aufruft. Anschließend setzt *Close* das BDE-Handle auf **nil**.

Temporäre Datenbankverbindungen trennen

Wenn die Eigenschaft *KeepConnections* einer Sitzung den Wert *True* hat (Vorgabe), werden Datenbankverbindungen für temporäre Datenbank-Komponenten auch dann aufrechterhalten, wenn alle von der Komponente verwendeten Datenmengen geschlossen werden. Sie können diese Verbindungen löschen sowie alle inaktiven temporären Datenbank-Komponenten einer Sitzung freigeben, indem Sie die Methode *DropConnections* aufrufen. Die folgende Anweisung beendet beispielsweise die inaktiven temporären Verbindungen der Standardsitzung:

```
Session.DropConnections;
```

Temporäre Datenbank-Komponenten mit einer oder mehreren aktiven Datenmengen werden durch diesen Aufruf nicht freigegeben. Um diese Komponenten freizugeben, müssen Sie *Close* aufrufen.

Nach einer Datenbankverbindung suchen

Mit der Methode *FindDatabase* können Sie ermitteln, ob eine bestimmte Datenbank-Komponente bereits mit einer Sitzung verbunden ist. *FindDatabase* erwartet als Parameter den Namen der gesuchten Datenbank. Dies ist entweder ein BDE-Alias oder der Name einer Datenbank-Komponente. Bei Paradox- oder dBASE-Tabellen kann der Name auch ein Pfadname sein.

FindDatabase gibt im Erfolgsfall die Datenbank-Komponente zurück, andernfalls **nil**.

Im folgenden Quelltext wird die Standardsitzung nach einer Datenbank-Komponente mit dem Alias DBDEMOS durchsucht. Wird die Komponente nicht gefunden, wird sie erzeugt und geöffnet:

```
var
   DB: TDatabase;
begin
   DB := Session.FindDatabase('DBDEMOS');
   if (DB = nil) then { Wenn die Datenbank in der Sitzung nicht gefunden wird, }
    DB := Session.OpenDatabase('DBDEMOS'); { wird sie erzeugt und geöffnet }
   if Assigned(DB) and DB.Active then begin
    DB.StartTransaction;
    ...
   end;
end;
```

Informationen über eine Sitzung abrufen

Mit einigen Methoden können Sie Informationen über eine Sitzung und ihre Datenbank-Komponenten abrufen. Eine Methode sucht beispielsweise nach den Namen aller bekannten Aliase der Sitzung, eine andere nach Namen von Tabellen, die mit einer bestimmten Datenbank-Komponente verbunden sind und von der Sitzung verwendet werden. Tabelle 16.1 enthält Methoden, mit denen eine Anwendung Informationen über die Datenbank abfragen kann.

Tabelle 16.1 Methoden, die Datenbankinformationen abfragen

Methode	Verwendungszweck
GetAliasDriverName	Gibt den BDE-Treiber für den angegebenen Alias zurück.
GetAliasNames	Gibt die Liste der BDE-Aliase einer Datenbank zurück.
GetAliasParams	Gibt die Parameterliste des angegebenen BDE-Alias zurück.
GetConfigParams	Gibt bestimmte Informationen aus der BDE-Konfigurationsdatei zurück.
GetDatabaseNames	Gibt die Liste der BDE-Aliase und die Namen aller gegenwärtig verwendeten *TDatabase*-Komponenten zurück
GetDriverNames	Gibt die Namen aller gegenwärtig installierten BDE-Treiber zurück.

Tabelle 16.1 Methoden, die Datenbankinformationen abfragen (Fortsetzung)

Methode	Verwendungszweck
GetDriverParams	Gibt die Parameterliste für den angegebenen BDE-Treiber zurück.
GetStoredProcNames	Gibt die Namen aller Stored Procedures der angegebenen Datenbank zurück.
GetTableNames	Gibt die Namen aller Tabellen einer Datenbank zurück, die mit dem angegebenen Muster übereinstimmen.

Mit Ausnahme von *GetAliasDriverName* übergeben diese Methoden eine Menge von Werten an eine von Ihrer Anwendung deklarierte und verwaltete String-Liste. *GetAliasDriverName* gibt den Namen des aktuellen BDE-Treibers einer bestimmten Datenbank-Komponente, die von der Sitzung verwendet wird, in einem String zurück.

Der folgende Quelltext ermittelt die Namen aller bekannten Datenbank-Komponenten und Aliase der Sitzung:

```
var
   List: TStringList;
begin
   List := TStringList.Create;
   try
    Session.GetDatabaseNames(List);
    ...
   finally
    List.Free;
   end;
end;
```

Eine vollständige Beschreibung der Methoden zum Ermitteln von Informationen über eine Sitzung finden Sie unter »TSession« in der VCL-Referenz.

BDE-Aliase

Da eine Sitzung normalerweise eine Reihe von Datenbankverbindungen kapselt, arbeiten einige der Eigenschaften und viele der Methoden einer Sitzungskomponente mit BDE-Aliasen. Jede Datenbank-Komponente, die mit einer Sitzung verbunden ist, verfügt über einen BDE-Alias (bei Paradox- und dBASE-Tabellen können Sie anstelle des Alias einen Pfadnamen angeben). BDE-Aliase und die zugehörigen *TSession*-Methoden werden in erster Linie für die folgenden Zwecke eingesetzt:

- Die Sichtbarkeit festlegen.
- Alias- und Treiberinformationen anzeigen.
- Aliase erzeugen, ändern und löschen.

Die Sichtbarkeit von Aliasen festlegen

Die Eigenschaft *ConfigMode* gibt an, welche BDE-Aliase in einer Sitzung sichtbar sind. *ConfigMode* ist eine Menge, die beschreibt, welche Sitzungstypen sichtbar sind. Die Standardeinstellung ist *cmAll*, was der Menge [*cfmVirtual, cfmPersistent*] entspricht. Wenn *ConfigMode cmAll* ist, sind alle in der Sitzung erzeugten Aliase, alle Aliase in der BDE-Konfigurationsdatei des Systems und alle Aliase, die sich im Speicher der BDE befinden, sichtbar.

ConfigMode ermöglicht einer Anwendung, die Sichtbarkeit von Aliasen in Sitzungen festzulegen und einzuschränken. Wenn Sie z.B. *ConfigMode* auf *cfmSession* setzen, sind für die Sitzung nur die in ihr erzeugten Aliase sichtbar, nicht aber die Aliase der BDE-Konfigurationsdatei und die Aliase im Speicher.

In der VCL-Referenz finden Sie eine Beschreibung von *ConfigMode* zusammen mit den entsprechenden Einstellungen.

Sitzungs-Aliase für andere Sitzungen und Anwendungen sichtbar machen

Wenn ein Alias während einer Sitzung erzeugt wird, hinterlegt die BDE eine Kopie im Speicher. Standardmäßig ist diese Kopie nur in der Sitzung sichtbar, in der sie erzeugt wird. Eine andere Sitzung derselben Anwendung kann den Alias nur dann sehen, wenn ihre Eigenschaft *ConfigMode* auf *cmAll* oder *cfmPersistent* gesetzt ist.

Mit der Methode *SaveConfigFile* der Sitzung können Sie den Alias für alle anderen Sitzungen und Anwendungen sichtbar machen. *SaveConfigFile* schreibt die Aliase im Speicher in die BDE-Konfigurationsdatei, wo sie von anderen BDE-Anwendungen gelesen und verwendet werden können.

Bekannte Aliase, Treiber und Parameter festlegen

Die folgenden fünf Methoden für Sitzungskomponenten ermöglichen einer Anwendung, Informationen über BDE-Aliase einschließlich der Parameter- und Treiberinformationen abzurufen:

- *GetAliasNames* zeigt die Aliase an, auf welche die Sitzung zugreifen kann.
- *GetAliasParams* zeigt die Parameter für einen bestimmten Alias an.
- *GetAliasDriverName* übergibt einen String, der die Namen des vom Alias verwendeten BDE-Treibers enthält.
- *GetDriverNames* gibt eine Liste aller BDE-Treiber zurück, die für die Sitzung verfügbar sind.
- *GetDriverParams* gibt die Parameter für einen bestimmten Treiber zurück.

Weitere Informationen über diese Methoden finden Sie im Abschnitt »Informationen über eine Sitzung abrufen« auf Seite 16-9. Weitere Informationen über BDE-Aliase, Parameter und Treiber finden Sie in der BDE-Hilfedatei BDE32.HLP.

Aliase erzeugen, ändern und löschen

Eine Sitzung kann Aliase erzeugen, ändern und löschen. Die Methode *AddAlias* erzeugt einen neuen BDE-Alias für einen SQL-Datenbankserver. *AddStandardAlias* erzeugt einen neuen BDE-Alias für Paradox-, dBASE- oder ASCII-Tabellen.

AddAlias erwartet drei Parameter: Einen String, der den Alias-Namen enthält, einen String, der den verwendeten SQL-Links-Treiber angibt, und eine Stringliste mit Parametern für den Alias. Weitere Informationen zu *AddAlias* finden Sie in der VCL-Referenz. Weitere Informationen über BDE-Aliase und SQL-Links-Treiber finden Sie in der BDE-Hilfedatei BDE32.HLP.

AddStandardAlias erwartet drei String-Parameter: den Namen des Alias, den vollständigen Pfad der Paradox- oder dBASE-Tabellen und den Namen des Standardtreibers für Tabellen ohne Erweiterung. Weitere Informationen zu *AddStandardAlias* finden Sie in der Online-Referenz der Objekt- und Komponentenbibliotheken. Weitere Informationen über BDE-Aliase finden Sie in der BDE-Hilfedatei BDE32.HLP.

Hinweis Wenn einer Sitzung ein Alias hinzugefügt wird, ist er nur für diese Sitzung und für diejenigen Sitzungen verfügbar, deren Eigenschaft *ConfigMode* das Flag *cfmPersistent* enthält. Rufen Sie nach dem Erzeugen des Alias *SaveConfigFile* auf, um ihn für alle Anwendungen verfügbar zu machen. Weitere Informationen über *ConfigMode* finden Sie im Abschnitt »BDE-Aliase« auf Seite 16-10.

Nachdem Sie einen Alias erzeugt haben, können Sie seine Parameter durch einen Aufruf von *ModifyAlias* ändern. *ModifyAlias* erwartet zwei Parameter: den Namen des Alias, der geändert werden soll, und eine Stringliste, die die zu ändernden Parameter und die neuen Werte enthält.

Mit der Methode *DeleteAlias* können Sie einen Alias löschen. *DeleteAlias* erwartet als Parameter den Namen des zu löschenden Alias. Nach dem Aufruf ist der Alias für die Sitzung nicht mehr verfügbar.

Hinweis *DeleteAlias* entfernt den Alias nicht aus der BDE-Konfigurationsdatei, wenn er durch einen vorangegangenen Aufruf von *SaveConfigFile* in die Datei geschrieben wurde. Rufen Sie nach dem Aufruf von *DeleteAlias* auch *SaveConfigFile* auf, um den Alias aus der Konfigurationsdatei zu entfernen.

In der folgenden Anweisung wird *AddAlias* verwendet, um für die Standardsitzung einen neuen Alias zum Zugriff auf einen InterBase-Server zu erzeugen:

```
var
   AliasParams: TStringList;
begin
   AliasParams := TStringList.Create;
   try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
   finally
    AliasParams.Free;
```

```
    end;
end;
```

In der folgenden Anweisung wird *AddStandardAlias* verwendet, um einen neuen Alias für den Zugriff auf eine Paradox-Tabelle zu erzeugen:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

In den folgenden Anweisungen wird *ModifyAlias* verwendet, um in der Standardsitzung den Parameter *OPEN MODE* für den Alias *CATS* auf *READ/WRITE* zu setzen:

```
var
   List: TStringList;
begin
   List := TStringList.Create;
   with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
   end;
   Session.ModifyAlias('CATS', List);
   List.Free;
   ...
```

Durch die Datenbank-Komponenten einer Sitzung iterieren

Mit den Komponenteneigenschaften *Databases* und *DatabaseCount* können Sie Aktionen mit allen aktiven Datenbank-Komponenten durchführen.

Die Eigenschaft *Databases* enthält ein Array aller gegenwärtig aktiven Datenbank-Komponenten, die mit der Sitzung verbunden sind. In Verbindung mit der Eigenschaft *DatabaseCount* kann *Databases* dazu verwendet werden, mit einigen oder allen Datenbank-Komponenten bestimmte Aktionen durchzuführen.

Die Eigenschaft *DatabaseCount* enthält einen Integerwert, der die Anzahl der gegenwärtig aktiven Datenbanken angibt, die mit der Sitzung verbunden sind. Dieser Wert kann sich im Lauf einer Sitzung ändern, wenn Verbindungen geöffnet und geschlossen werden. Wenn beispielsweise die Eigenschaft *KeepConnections* einer Sitzung *False* ist und alle Datenbank-Komponenten zur Laufzeit erzeugt werden, erhöht sich der Wert von *DatabaseCount* jedesmal um 1, wenn eine neue Datenbank geöffnet wird. Umgekehrt wird der Wert um 1 verringert, wenn eine Datenbank geschlossen wird. Wenn *DatabaseCount* den Wert 0 hat, gibt es in der Sitzung keine gegenwärtig aktiven Datenbank-Komponenten.

DatabaseCount wird gewöhnlich zusammen mit der Eigenschaft *Databases* verwendet, um Aktionen mit allen aktiven Datenbank-Komponenten durchzuführen.

Das folgende Beispiel setzt die Eigenschaft *KeepConnection* für jede aktive Datenbank der Standardsitzung auf *True*:

```
var
   MaxDbCount: Integer;
begin
   with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
```

```
        Databases[MaxDbCount].KeepConnection := True;
end;
```

Paradox-Verzeichnisse angeben

Die Eigenschaften *NetFileDir* und *PrivateDir* von Sitzungskomponenten werden für Anwendungen benötigt, die mit Paradox-Tabellen arbeiten. *NetFileDir* gibt das Verzeichnis an, das die Netzwerk-Steuerdatei von Paradox (PDOXUSRS.NET) enthält. Diese Datei regelt die gleichzeitige Verwendung von Paradox-Tabellen auf Netzlaufwerken. Alle Anwendungen, die Paradox-Tabellen gemeinsam verwenden, müssen für die Netzwerk-Steuerdatei dasselbe Verzeichnis angeben (normalerweise ein Verzeichnis auf einem Datei-Server im Netzwerk).

PrivateDir gibt das Verzeichnis an, in dem temporäre Dateien gespeichert werden, wie sie z.B. von der BDE bei der Bearbeitung lokaler SQL-Anweisungen erzeugt werden.

Das Verzeichnis der Steuerdatei festlegen

Delphi entnimmt den Wert für *NetFileDir* der Konfigurationsdatei der Borland Database Engine (BDE). Seien Sie vorsichtig, wenn Sie *NetFileDir* selbst einen Wert zuweisen, da dieser Wert Vorrang vor dem Wert der BDE-Konfiguration hat.

Zur Entwurfszeit können Sie den Wert für *NetFileDir* im Objektinspektor festlegen. Zur Laufzeit können Sie *NetFileDir* in Ihrem Quelltext ändern. Die folgende Anweisung setzt *NetFileDir* für die Standardsitzung auf das Verzeichnis, in dem die Anwendung gestartet wurde:

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

Hinweis *NetFileDir* kann nur geändert werden, wenn eine Anwendung keinerlei Paradox-Dateien verwendet. Wenn Sie die Eigenschaft *NetFileDir* zur Laufzeit ändern, stellen Sie sicher, daß sie auf ein gültiges Netzwerkverzeichnis zeigt, auf das die Netzwerkbenutzer gemeinsam zugreifen können.

Ein Verzeichnis für temporäre Dateien festlegen

Wenn kein Wert für *PrivateDir* angegeben ist, verwendet die BDE automatisch das aktuelle Verzeichnis zur Zeit ihrer Initialisierung. Wenn Ihre Anwendung direkt auf einem Datei-Server im Netzwerk läuft, sollte *PrivateDir* auf ein lokales Laufwerk zeigen, bevor Sie die Datenbank öffnen, da dadurch die Leistung des Servers zur Laufzeit gesteigert wird.

Hinweis Vermeiden Sie es, *PrivateDir* zur Entwurfszeit zu setzen und dann in der IDE die Sitzung zu öffnen. Bei der Ausführung Ihrer Anwendung in der IDE erschiene sonst eine Fehlermeldung.

Die folgende Anweisung weist der Eigenschaft *PrivateDir* der Standardsitzung das Verzeichnis C:\TEMP des Benutzers zu:

```
Session.PrivateDir := 'C:\TEMP';
```

Wichtiger Hinweis Setzen Sie *PrivateDir* nicht auf das Stammverzeichnis eines Laufwerks. Geben Sie immer ein Unterverzeichnis an.

Kennwortgeschützte Paradox- und dBase-Tabellen

Eine Sitzungskomponente verfügt über vier Methoden und ein Ereignis, die ausschließlich für den Zugriff auf kennwortgeschützte Paradox- und dBase-Dateien verwendet werden. Es handelt sich um die Methoden *AddPassword, GetPassword, RemoveAllPasswords* und *RemovePassword* und um das Ereignis *OnPassword*. Zum Hinzufügen und Entfernen von Kennwörtern einer Sitzung steht auch die Funktion *PasswordDialog* zur Verfügung.

Die Methode AddPassword verwenden

Die Methode *TSession.AddPassword* stellt eine zusätzliche Möglichkeit zur Verfügung, um einer Sitzung vor dem Öffnen einer verschlüsselten Paradox- oder dBase-Tabelle, für die ein Kennwort erforderlich ist, ein solches zuzuweisen. *AddPassword* erwartet einen String-Parameter, der das Kennwort enthält. Durch wiederholte Aufrufe von *AddPassword* können Sie beliebig viele Kennwörter hinzufügen.

```
var
   Passwrd: String;
begin
   Passwrd := InputBox('Kennwort eingeben', 'Kennwort: ', '');
   Session.AddPassword(Passwrd);
   try
    Table1.Open
   except
    ShowMessage('Kann Tabelle nicht öffnen!');
    Application.Terminate;
   end;
end;
```

Die Verwendung der Funktion *InputBox* im vorhergehenden Quelltext dient nur als Beispiel. In einer echten Anwendung würden Sie ein benutzerdefiniertes Formular oder Funktionen verwenden, die das Anzeigen des Kennworts während der Eingabe verhindern, wie beispielsweise die Funktion *PasswordDialog*. In einem benutzerdefinierten Formular zur Eingabe von Kennwörtern verwenden Sie *TEdit* mit *PasswordChar*, um bei der Eingabe Sternchen (*) anzuzeigen.

Die Schaltfläche *Hinzufügen* im Dialogfeld der Funktion *PasswordDialog* hat denselben Effekt wie die Methode *AddPassword*.

```
if PasswordDialog(Session) then
   Table1.Open
else
   ShowMessage('Kein Kennwort angegeben. Kann Tabelle nicht öffnen!');
end;
```

Hinweis Wenn Ihre Anwendung versucht, eine kennwortgeschützte Tabelle zu öffnen, und Sie zuvor weder mit *AddPassword* ein Kennwort festgelegt noch eine Ereignisbehandlungsroutine für *OnPassword* definiert haben, wird ein Dialogfeld angezeigt, in dem Sie zur Eingabe eines Kennwortes aufgefordert werden.

Die Methoden RemovePassword und RemoveAllPasswords verwenden

TSession.RemovePassword löscht ein zuvor festgelegtes Kennwort aus dem Speicher. Die Methode erwartet einen String-Parameter, der das zu löschende Kennwort angibt.

```
Session.RemovePassword('secret');
```

TSession.RemoveAllPasswords löscht alle Kennwörter aus dem Speicher.

```
Session.RemoveAllPasswords;
```

Die Methode GetPassword und das Ereignis OnPassword verwenden

TSession.GetPassword löst das Ereignis *TSession.OnPassword* einer Sitzung aus. Dies geschieht nur dann, wenn eine Anwendung versucht, eine Paradox- oder dBase-Tabelle zum ersten Mal zu öffnen, und die BDE meldet, daß keine ausreichende Zugriffsberechtigung vorliegt. Sie können eine Ereignisbehandlungsroutine für *OnPassword* schreiben und ein Kennwort für die BDE zur Verfügung stellen. Andernfalls wird ein Standard-Dialogfeld angezeigt, das den Benutzer zur Eingabe eines Kennwortes auffordert.

Im folgenden Beispiel wird die Prozedur *Password* als Behandlungsroutine für das Ereignis *OnPassword* definiert, indem der Name der Prozedur der Eigenschaft *TSession.OnPassword* zugewiesen wird.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Session.OnPassword := Password;
end;
```

In der Prozedur *Password* wird die Funktion *InputBox* verwendet, um den Benutzer zur Eingabe eines Kennworts aufzufordern. Die Methode *TSession.AddPassword* stellt der Sitzung das eingegebene Kennwort zur Verfügung.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
   Passwrd: String;
begin
   Passwrd := InputBox('Enter password', 'Password:', '');
   Continue := (Passwrd > '');
   Session.AddPassword(Passwrd);
end;
```

Im folgenden Beispiel wird das Ereignis *OnPassword* durch den Versuch ausgelöst, eine kennwortgeschützte Tabelle zu öffnen. Die Exception-Behandlung kann verwendet werden, um einen fehlgeschlagenen Öffnungsversuch abzufangen. Obwohl der Benutzer in der Ereignisbehandlungsroutine für *OnPassword* zur Eingabe eines Kennworts aufgefordert wird, kann der Versuch, die Tabelle zu öffnen, durch Eingabe eines ungültigen Kennworts oder Auftreten eines anderen Fehlers fehlschlagen.

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
  CRLF = #13 + #10;
begin
   try
```

```
    Table1.Open; { Diese Zeile löst das Ereignis OnPassword aus }
   except
    on E:Exception do begin { Exception, wenn die Tabelle nicht geöffnet werden
                              kann }
      ShowMessage('Fehler!' + CRLF + { Fehlermeldung mit Erklärung }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate; { Anwendung beenden }
    end;
   end;
end;
```

Mehrere Sitzungen verwalten

Wenn Sie eine Anwendung schreiben, die mehrere Threads für Datenbankoperationen verwendet, müssen Sie für jeden Thread eine zusätzliche Sitzung erzeugen. Die Seite *Datenzugriff* der Komponentenpalette enthält eine Sitzungskomponenete, die Sie zur Entwurfszeit in ein Datenmodul oder Formular einfügen können.

Wichtiger Hinweis

Der Wert der Eigenschaft *SessionName* einer neuen Sitzungskomponente muß so gewählt werden, daß er nicht mit der Eigenschaft *SessionName* der Standardsitzung konfligiert.

Wenn Sie zur Entwurfszeit eine Sitzungskomponente hinzufügen, gehen Sie gewöhnlich davon aus, daß die von der Anwendung zur Laufzeit benötigte Anzahl von Threads (und daher Sitzungen) statisch ist. Es ist jedoch eher wahrscheinlich, daß eine Anwendung Sitzungen dynamisch erzeugen muß. Mit der globalen Funktion *Sessions.OpenSession* können Sitzungen zur Laufzeit dynamisch erzeugt werden.

Sessions.OpenSession erwartet nur einen Parameter, nämlich einen Sitzungsnamen, der anwendungsweit eindeutig ist. Der folgende Quelltext erzeugt und aktiviert eine Sitzung dynamisch und weist ihr einen eindeutigen Namen zu:

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

Diese Anweisung erzeugt einen eindeutigen Namen für die neue Sitzung, indem sie die aktuelle Anzahl der Sitzungen ermittelt und diesen Wert um 1 erhöht. Beachten Sie, daß dieses Beispiel nicht wie erwartet funktioniert, wenn Sie Sitzungen zur Laufzeit dynamisch erzeugen und wieder freigeben. Es soll nur verdeutlichen, wie mit den Methoden und Eigenschaften von *Sessions* mehrere Sitzungen verwaltet werden können.

Sessions ist vom Typ *TSessionList* und wird für Datenbankanwendungen automatisch instantiiert. Mit den Eigenschaften und Methoden von *Sessions* können mehrere Sit-

zungen in einer Multithread-Datenbankanwendung verwaltet werden. Tabelle 16.2 enthält die Eigenschaften und Methoden der Komponente *TSessionList*:

Tabelle 16.2 Eigenschaften und Methoden von *Sessions*

Eigenschaft oder Methode	Verwendungszweck
Count	Gibt die Anzahl der aktiven und inaktiven Sitzungen der Sitzungsliste zurück
FindSession	Durchsucht die Sitzungsliste nach der Sitzung mit dem angegebenen Namen und gibt einen Zeiger auf die Sitzungskomponente zurück. Wenn die angegebene Sitzung nicht existiert, wird **nil** zurückgegeben. Wenn der Methode ein leerer String übergeben wird, gibt sie einen Zeiger auf die Standardsitzung Session zurück.
GetSessionNames	Gibt eine Stringliste mit den Namen aller gegenwärtig instantiierten Sitzungskomponenten zurück. Diese Prozedur fügt immer mindestens den String *Default* für die Standardsitzung hinzu (beachten Sie, daß der Name der Standardsitzung eigentlich ein leerer String ist).
List	Gibt die Sitzungskomponente für den angegebenen Sitzungsnamen zurück. Wenn keine Sitzung mit dem angegebenen Namen existiert, wird eine Exception ausgelöst.
OpenSession	Erzeugt und aktiviert eine neue Sitzung mit dem angegebenen Namen oder aktiviert die bestehende Sitzung mit diesem Namen.
Sessions	Greift über einen Ordinalwert auf die Sitzungsliste zu.

Ein Beispiel soll verdeutlichen, wie die Eigenschaften und Methoden von *Sessions* in einer Multithread-Anwendung eingesetzt werden, wenn Sie beispielsweise eine Datenbankverbindung öffnen wollen. Mit der Eigenschaft *Sessions* können Sie herausfinden, ob die gewünschte Verbindung bereits besteht, indem Sie, beginnend bei der Standardsitzung, jede Sitzung in der Liste untersuchen. Die Eigenschaft *Database* einer jeden Sitzungskomponente gibt Auskunft darüber, ob die fragliche Datenbank bereits geöffnet ist. Wenn Sie feststellen, daß die Datenbank bereits von einem anderen Thread verwendet wird, überprüfen Sie die nächste Sitzung in der Liste.

Wenn ein bestehender Thread die Datenbank nicht verwendet, kann die Verbindung innerhalb dieser Sitzung geöffnet werden.

Wenn die Datenbank dagegen von allen bestehenden Threads verwendet wird, müssen Sie eine neue Sitzung und in dieser Sitzung eine neue Datenbankverbindung öffnen.

Wenn Sie ein Datenmodul replizieren, das eine Sitzung in einer Multithread-Anwendung enthält, in der wiederum jeder Thread seine eigene Kopie des Datenmoduls enthält, können Sie die Eigenschaft *AutoSessionName* verwenden, um sicherzustellen, daß alle Datenmengen im Datenmodul die korrekte Sitzung verwenden. Wenn *AutoSessionName* auf *True* gesetzt wird, generiert die Sitzung bei ihrer Erzeugung zur Laufzeit dynamisch einen eigenen eindeutigen Namen. Dieser Name wird dann allen Datenmengen im Datenmodul zugewiesen. Alle explizit festgelegten Sitzungsnamen werden dabei überschrieben. Jeder Thread erhält dadurch seine eigene Sitzung, und jede Datenmenge verwendet die Sitzung in ihrem eigenen Datenmodul.

Sitzungskomponenten in Datenmodulen verwenden

TSession- und *TDatabase*-Komponenten können problemlos in Datenmodulen plaziert werden. Wenn Sie jedoch ein Datenmodul mit *TSession*- oder *TDatabase*-Komponenten in der Objektablage ablegen, müssen Sie sicherstellen, daß *AutoSessionName* auf *True* gesetzt ist, damit keine Namespace-Konflikte auftreten, wenn Benutzer Komponenten davon ableiten.

Kapitel

17

Datenbankverbindungen

Die Verbindung zwischen einer Delphi-Anwendung und einer Datenbank über die BDE (Borland Database Engine) wird mit Hilfe der Komponente *TDatabase* gekapselt. Dieses Objekt kapselt in einer Anwendung die Verbindung zu einer bestimmten Datenbank im Kontext einer BDE-Sitzung. Dieses Kapitel enthält Informationen zu den Datenbank-Komponenten und beschreibt, wie Sie Datenbankverbindungen steuern können.

Datenbank-Komponenten werden auch zur Verwaltung von Transaktionen in BDE-basierten Anwendungen eingesetzt. Einzelheiten hierzu finden Sie unter »Transaktionen« auf Seite 13-5.

Ein weiteres Einsatzgebiet für Datenbank-Komponenten ist das Eintragen zwischengespeicherter Aktualisierungen in verknüpfte Tabellen. Informationen zu diesem Thema finden Sie unter »Aktualisierungen mit Datenbank-Komponenten eintragen« auf Seite 25-5.

Persistente und temporäre Datenbank-Komponenten

Jede auf BDE basierende Datenbankverbindung in einer Anwendung wird durch eine Datenbank-Komponente gekapselt, unabhängig davon, ob Sie die Komponente explizit bereitstellen oder nicht. Sobald eine Anwendung eine Verbindung mit der Datenbank herstellen will, verwendet sie entweder eine explizit instantiierte (*persistente*) Datenbank-Komponente oder generiert eine temporäre Komponente, die nur vorhanden ist, solange die Verbindung besteht.

Temporäre Datenbank-Komponenten für eine Datenmenge werden bei Bedarf automatisch in einem Datenmodul bzw. Formular erstellt. Diese temporären Objekte stellen eine umfassende Unterstützung für viele typische Desktop-Datenbankanwendungen bereit, ohne daß Sie sich mit den Einzelheiten der Datenbankverbindung befassen müssen. Bei den meisten Client/Server-Anwendungen sollten Sie jedoch eigene Datenbank-Komponenten erstellen. Sie erhalten dadurch umfangreiche Steue-

rungsmöglichkeiten für die Datenbank und können zusätzlich folgende Aktionen durchführen:

- Persistente Datenbankverbindungen erstellen.
- Datenbankserver-Logins anpassen.
- Transaktionen steuern und Isolationsstufen festlegen.
- In Ihrer Anwendung lokale BDE-Aliasnamen erstellen.

Temporäre Datenbank-Komponenten

Temporäre Datenbank-Komponenten werden bei Bedarf automatisch erstellt. Wenn Sie beispielsweise eine *TTable*-Komponente in einem Formular plazieren, ihre Eigenschaften festlegen und dann die Tabelle öffnen, ohne zuerst eine *TDatabase*-Komponente hinzuzufügen, einzurichten und der Tabelle zuzuordnen, wird automatisch eine temporäre Datenbank-Komponente erzeugt.

Einige Schlüsseleigenschaften temporärer Datenbank-Komponenten werden durch die Sitzung bestimmt, zu der sie gehören. So legt beispielsweise die Eigenschaft *KeepConnections* der steuernden Sitzung fest, ob eine Datenbankverbindung auch dann aufrechterhalten wird, wenn die zugehörigen Datenmengen geschlossen werden (Standardeinstellung), oder ob die Verbindung beim Schließen der Datenmengen beendet wird. Entsprechend stellt das *OnPassword*-Standardereignis einer Sitzung sicher, daß eine Standard-Kennwortabfrage angezeigt wird, wenn eine Anwendung die Verbindung mit einer Datenbank auf einem Server herstellen will, der ein Kennwort erfordert. Andere Eigenschaften temporärer Datenbank-Komponenten stellen die Standardbehandlung für Logins und Transaktionen bereit. Weitere Informationen über Sitzungen und die Sitzungskontrolle über temporäre Datenbankverbindungen finden Sie unter »Mit einer Sitzungskomponente arbeiten« auf Seite 16-2.

Die Standardeigenschaften der temporären Datenbank-Komponente stellen allgemeine Funktionsweisen für eine Vielzahl verschiedener Situationen zur Verfügung. Bei komplexen Client/Server-Anwendungen mit vielen Benutzern und unterschiedlichen Anforderungen an die Datenbankverbindungen sollten Sie jedoch eigene Datenbank-Komponenten erstellen, um die einzelnen Verbindungen an die Bedürfnisse der Anwendung anzupassen.

Datenbank-Komponenten zur Entwurfszeit erstellen

Die Registerkarte *Datenzugriff* der Komponentenpalette enthält eine Datenbank-Komponente, die Sie in einem Datenmodul oder Formular plazieren können. Der Hauptvorteil des Erstellens einer Datenbank-Komponente zur Entwurfszeit besteht darin, daß Sie ihre Anfangseigenschaften festlegen und *OnLogin*-Ereignisbehandlungsroutinen für sie schreiben können. Mit *OnLogin* können Sie die Behandlung der Sicherheitsmaßnahmen auf einem Datenbankserver anpassen, wenn eine Datenbank-Komponente erstmals eine Verbindung zum Server herstellt. Weitere Informationen über das Verwalten der Verbindungseigenschaften finden Sie unter »Mit einem Datenbankserver verbinden« auf Seite 17-7. Informationen über die Server-Sicherheit finden Sie im Abschnitt »Server-Login steuern« auf Seite 17-6.

Datenbank-Komponenten zur Laufzeit erstellen

Datenbank-Komponenten können zur Laufzeit dynamisch erstellt werden. Diese Vorgehensweise ist notwendig, wenn die Anzahl der zur Laufzeit benötigten Datenbank-Komponenten nicht bekannt ist und Ihre Anwendung die explizite Steuerung der Datenbankverbindung erforderlich macht. Delphi richtet die temporären Datenbank-Komponenten zur Laufzeit nach Bedarf ein. Wenn Sie eine Datenbank-Komponente zur Laufzeit erstellen, müssen Sie ihr einen eindeutigen Namen zuweisen und sie einer Sitzung zuordnen.

Eine Datenbank-Komponente wird durch den Aufruf des Kontruktors *TDatabase.Create* erstellt. Der folgende Quelltext erstellt auf Grundlage eines Datenbank- und Sitzungsnamens zur Laufzeit eine Datenbank-Komponente, verknüpft sie mit der angegebenen Sitzung (die gegebenenfalls erzeugt wird) und legt die Anfangswerte wichtiger Eigenschaften der Datenbank-Komponente fest:

```
function TDataForm.RunTimeDbCreate(const DatabaseName, SessionName: String): TDatabase;
var
  TempDatabase: TDatabase;
begin
  TempDatabase := nil;
  try
    { Wenn die Sitzung existiert, diese aktivieren; sonst neue serstellen}
    Sessions.OpenSession(SessionName);
    with Sessions do
      with FindSession(SessionName) do begin
        Result := FindDatabase(DatabaseName);
        if (Result = nil) then begin
          { Create a new database component }
          TempDatabase := TDatabase.Create(Self);
          TempDatabase.DatabaseName := DatabaseName;
          TempDatabase.SessionName := SessionName;
          TempDatabase.KeepConnection := True;
        end;
        Result := OpenDatabase(DatabaseName);
      end;
  except
    TempDatabase.Free;
    raise;
  end;
end;
```

Das folgende Beispiel zeigt, wie Sie mit dieser Funktion zur Laufzeit eine Datenbank-Komponente für die Standardsitzung erstellen können:

```
var
  MyDatabase: array [1..10] of  TDatabase;
  MyDbCount: Integer;
begin
  { Initialize MyDbCount early on }
  MyDbCount := 1;
  ⋮
  { Später eine Datenbank-Komponente zur Laufzeit erstellen }
  begin
```

```
      MyDatabase[MyDbCount] := RunTimeDbCreate('MyDb' + IntToStr(MyDbCount), '');
      Inc(MyDbCount);
    end;
    :
end;
```

Verbindungen steuern

Unabhängig davon, ob Sie eine Datenbank-Komponente zur Entwurfs- oder zur Laufzeit erstellen, können Sie ihre Funktionsweise mit den Eigenschaften, Ereignissen und Methoden von *TDatabase* in Ihren Anwendungen steuern. In den folgenden Abschnitten finden Sie Informationen, wie Sie mit Datenbank-Komponenten arbeiten können. Einzelheiten zu den verschiedenen Eigenschaften, Ereignissen und Methoden der Klasse *TDatabase* finden Sie in der Online-Hilfe.

Eine Datenbank-Komponente mit einer Sitzung verbinden

Jede Datenbank-Komponente muß mit einer BDE-Sitzung verbunden werden. Diese Verbindung wird mit den beiden Eigenschaften *Session* und *SessionName* der Komponente eingerichtet.

SessionName bezeichnet den Sitzungs-Alias, dem die Komponente zugeordnet werden soll. Wird eine Datenbank-Komponente während des Entwurfs erstellt, hat *SessionName* den Standardwert *Default*. Programme mit mehreren Threads oder reentrante BDE-Anwendungen können auch mehrere Sitzungen verwalten. Während des Entwurfs können Sie einen gültigen SessionName-Wert aus der Dropdown-Liste im Objektinspektor wählen. Die Sitzungsnamen in der Liste entsprechen dem Wert der Eigenschaft *SessionName* der einzelnen Sitzungskomponenten der Anwendung.

Die Nur-Lesen-Eigenschaft *Session* ist nur zur Laufzeit verfügbar und zeigt auf die von der Eigenschaft *SessionName* angegebene Sitzungskomponente. Ist *SessionName* beispielsweise leer oder hat den Wert *Default*, nimmt die Eigenschaft *Session* auf die von der globalen Variable *Session* referenzierte *TSession*-Instanz Bezug. *Session* ermöglicht einer Anwendung, auf die Eigenschaften, Methoden und Ereignisse der übergeordneten Sitzungskomponente einer Datenbank-Komponente zuzugreifen, ohne daß der eigentliche Name der Sitzung bekannt ist. Dies ist hilfreich, wenn eine Datenbank-Komponente zur Laufzeit einer anderen Sitzung zugewiesen werden soll.

Weitere Informationen über BDE-Sitzungen finden Sie in Kapitel 16, »Datenbanksitzungen«.

Einen BDE-Alias angeben

Die BDE-spezifischen Eigenschaften *AliasName* und *DriverName* schließen sich gegenseitig aus. *AliasName* gibt den Namen eines vorhandenen BDE-Alias an, der für die Datenbank-Komponente verwendet werden soll. Der Alias erscheint dann in den Dropdown-Listen für Datenmengenkomponenten, so daß sie mit einer bestimmten Datenbank-Komponente verbunden werden können. Wenn Sie *AliasName* für eine

Datenbank-Komponente angeben, wird der Wert der Eigenschaft *DriverName* gelöscht, da zu jedem BDE-Alias ein eigener Treibername gehört.

Hinweis BDE-Aliase können mit dem Datenbank-Explorer oder mit der BDE-Konfiguration erstellt und bearbeitet werden. Weitere Informationen zu BDE-Aliasnamen finden Sie in der Online-Hilfe der beiden Dienstprogramme.

Mit der Eigenschaft *DatabaseName* können Sie einen anwendungsspezifischen Namen für eine Datenbank-Komponente angeben. Dieser Name ist in der Anwendung lokal und wird zusätzlich zu *AliasName* oder *DriverName* verwendet. *DatabaseName* kann ein BDE-Alias oder (bei Paradox- und dBASE-Tabellen) ein vollständiger Pfadname sein. Wie *AliasName* erscheint auch *DatabaseName* anschließend in den Dropdown-Listen der Datenmengenkomponenten, so daß Sie sie mit einer Datenbank-Komponente verbinden können.

DriverName ist der Name eines BDE-Treibers. Der Treibername ist ein Parameter in einem BDE-Alias. Sie können aber einen Treiber anstelle eines Alias angeben, wenn Sie mit Hilfe der Eigenschaft *DatabaseName* einen lokalen BDE-Alias für eine Datenbank-Komponente erstellen. Geben Sie *DriverName* an, wird der Wert von *AliasName* automatisch gelöscht, damit keine Konflikte zwischen dem von Ihnen festgelegten Treibernamen und dem in *AliasName* als Teil des BDE-Alias gespeicherten Treibernamen auftreten können.

Um während des Entwurfs einen BDE-Alias festzulegen, einen BDE-Treiber zuzuweisen oder einen lokalen BDE-Alias zu erstellen, doppelklicken Sie auf eine Datenbank-Komponente, um den Datenbankeditor zu öffnen.

Im Eingabefeld *Name* können Sie einen Datenbanknamen eingeben. Im Kombinationsfeld Aliasname geben Sie einen vorhandenen BDE-Alias ein oder wählen den gewünschten Namen aus der Dropdown-Liste. Im Kombinationsfeld *Treibername* können Sie den Namen eines vorhandenen BDE-Treibers eingeben oder den gewünschten Treibernamen aus der Dropdown-Liste wählen.

Hinweis Im Datenbankeditor können Sie auch die BDE-Verbindungsparameter anzeigen oder festlegen und den Status der Eigenschaften *LoginPrompt* und *KeepConnection* setzen. Informationen über die Arbeit mit Verbindungsparametern finden Sie unter »BDE-Aliasparameter festlegen« auf Seite 17-5. Informationen zur Eigenschaft *LoginPrompt* finden Sie unter »Server-Login steuern« auf Seite 17-6. Informationen über *KeepConnection* finden Sie unter »Mit einem Datenbankserver verbinden« auf Seite 17-7.

Um *DatabaseName, AliasName* oder *DriverName* zur Laufzeit zu setzen, nehmen Sie die entsprechenden Zuweisungen im Quelltext vor. Im folgenden Beispiel wird der Inhalt eines Eingabefeldes verwendet, um einen lokalen Alias für die Datenbank-Komponente Database1 zu erzeugen:

```
Database1.DatabaseName := Edit1.Text;
```

BDE-Aliasparameter festlegen

Während des Entwurfs können Sie Verbindungsparameter auf die folgenden drei Arten erstellen oder bearbeiten:

- Mit dem Datenbank-Explorer oder der BDE-Konfiguration. Weitere Informationen zu diesen Anwendungen finden Sie in der Online-Hilfe.
- Doppelklicken Sie im Objektinspektor auf die Eigenschaft *Params*, um den Stringlisten-Editor zu öffnen. Weitere Informationen zu diesem Editor finden Sie unter»Stringlisten« in der Online-Hilfe.
- Doppelklicken Sie in einem Datenmodul oder Formular auf eine Datenbank-Komponente, um den Datenbankeditor zu öffnen.

Bei diesen drei Vorgehensweisen wird die Eigenschaft *Params* der Datenbank-Komponente bearbeitet. Die Stringliste *Params* enthält die Verbindungsparameter für den BDE-Alias einer Datenbank-Komponente. Zu den Parametern gehören beispielsweise Pfadanweisung, Server-Name, Cache-Größe, Sprachtreiber und SQL-Abfragemodus.

Wenn der Datenbankeditor zum ersten Mal aufgerufen wird, sind die Parameter für den BDE-Alias nicht sichtbar. Um die aktuellen Einstellungen anzuzeigen, klicken Sie auf *Vorgaben*. Die aktuellen Parameter sind dann im Memofeld *Parameter* überschreibt zu sehen. Sie können die vorhandenen Einträge bearbeiten oder neue hinzufügen. Mit der Schaltfläche *Löschen* können die bestehenden Parameter entfernt werden. Änderungen werden erst wirksam, wenn Sie auf *OK* klicken.

Zur Laufzeit können die Aliasparameter nur geändert werden, indem die Eigenschaft *Params* direkt bearbeitet wird. Informationen über die spezifischen Parameter für SQL Links-Treiber finden Sie in der Online-Hilfe.

Server-Login steuern

Die meisten Remote-Datenbankserver verfügen über Sicherheitsvorkehrungen, um unberechtigte Zugriffe auszuschließen. Normalerweise fordert der Server zur Eingabe eines Benutzernamens und eines Kennworts auf, bevor er den Zugriff auf die Datenbank erlaubt.

Während des Entwurfs wird bei einem Server, der ein Login erfordert, ein Standarddialogfeld angezeigt, in dem Sie beim ersten Verbindungsversuch mit der Datenbank zur Eingabe eines Benutzernamens und eines Kennworts aufgefordert werden.

Zur Laufzeit können Sie die Login-Anforderung eines Servers auf drei Arten behandeln:

- Setzen Sie die Eigenschaft *LoginPrompt* der Datenbank-Komponente auf *True* (Standardeinstellung). Ihre Anwendung zeigt dann das Login-Standarddialogfeld an, wenn der Server einen Benutzernamen und ein Kennwort anfordert.
- Setzen Sie die Eigenschaft *LoginPrompt* auf *False*, und fügen Sie der Eigenschaft *Params* der Datenbank-Komponente die Parameter USER NAME und PASSWORD mit den entsprechenden Werten hinzu. Ein Beispiel:

```
USER NAME=SYSDBA
PASSWORD=masterkey
```

Wichtig Beachten Sie, daß der Wert der Eigenschaft *Params* sehr einfach angezeigt werden kann. Soll die Server-Sicherheit gewährleistet sein, ist diese Vorgehensweise daher nicht zu empfehlen.

- Schreiben Sie eine Behandlungsroutine für das Ereignis *OnLogin* der Datenbank-Komponente, und setzen Sie hier die Login-Parameter zur Laufzeit. *OnLogin* erhält eine Kopie der Eigenschaft *Params* der Datenbank-Komponente, die Sie beliebig ändern können. Der Name der Kopie in *OnLogin* ist *LoginParams*. Folgendermaßen können Sie die Login-Parameter mit der Eigenschaft *Values* setzen oder ändern:

```
LoginParams.Values['USER NAME'] := UserName;
LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
```

Beim Beenden gibt *OnLogin* seine *LoginParams*-Werte an *Params* zurück, wo sie dann zum Einrichten der Verbindung verwendet werden.

Mit einem Datenbankserver verbinden

Es gibt zwei Möglichkeiten, mit einer Datenbank-Komponente eine Verbindung zu einem Datenbankserver herzustellen:

- Rufen Sie die Methode *Open* auf.
- Setzen Sie die Eigenschaft *Connected* auf *True*.

Wenn Sie *Connected* den Wert *True* zuweisen, wird die Methode *Open* aufgerufen. *Open* prüft, ob die mit der Eigenschaft *DatabaseName* oder *Directory* angegebene Datenbank vorhanden ist, und ruft die Behandlungsroutine für das Ereignis *OnLogin* der Datenbank-Komponente auf, falls eine solche definiert wurde. Andernfalls wird das Login-Standarddialogfeld angezeigt.

Hinweis Will eine Anwendung eine Datenmenge öffnen, die einer nicht mit einem Server verbundenen Datenbank-Komponente zugeordnet ist, wird zuerst die Methode *Open* der Komponente aufgerufen, um die Verbindung einzurichten. Ist die Datenmenge nicht mit einer vorhandenen Datenbank-Komponente verbunden, wird eine temporäre Komponente erzeugt und zum Einrichten der Verbindung verwendet.

Eine eingerichtete Datenbankverbindung wird aufrechterhalten, solange mindestens eine aktive Datenmenge vorhanden ist. Ist keine Datenmenge mehr aktiv, wird die Verbindung abhängig von der Einstellung der Eigenschaft *KeepConnection* der Datenbank-Komponente geschlossen oder beibehalten.

KeepConnection bestimmt, ob die Verbindung mit einer Datenbank auch dann aufrechterhalten wird, wenn alle zugehörigen Datenmengen geschlossen sind. Hat die Eigenschaft den Wert *True*, bleibt die Verbindung bestehen. Bei Verbindungen mit Remote-Servern oder bei Anwendungen, in denen Datenmengen häufig geöffnet und geschlossen werden, sollten Sie *KeepConnection* auf *True* setzen. Dadurch kann das Datenaufkommen im Netzwerk verringert und die Anwendung beschleunigt werden. Hat *KeepConnection* den Wert *False*, wird die Verbindung geschlossen, sobald keine aktive Datenmenge mehr vorhanden ist. Wird später eine Datenmenge geöffnet, die die Datenbank verwendet, muß die Verbindung erneut eingerichtet und initialisiert werden.

Besonderheiten beim Verbinden mit einem Remote-Server

Wenn Sie in einer Anwendung eine Verbindung zu einem Remote-Server herstellen, werden die BDE und der Borland SQL Links-Treiber zum Einrichten der Verbindung verwendet (die BDE kann auch mit einem ODBC-Treiber kommunizieren). Bevor die Verbindung hergestellt werden kann, muß jedoch SQL Links oder der ODBC-Treiber für die Anwendung konfiguriert werden. Die Parameter für SQL Links und ODBC werden in der Eigenschaft *Params* der Datenbank-Komponente gespeichert. Informationen über die verschiedenen SQL Links-Parameter finden Sie in der Online-Hilfe zu SQL Links. Informationen über die Eigenschaft *Params* finden Sie unter »BDE-Aliasparameter festlegen« auf Seite 17-5.

Mit Netzwerkprotokollen arbeiten

Beim Konfigurieren des SQL Links- oder ODBC-Treibers müssen Sie möglicherweise (je nach den Konfigurationsoptionen des Treibers) auch das vom Server verwendete Netzwerkprotokoll angeben (z. B. SPX / IPX oder TCP / IP). Meistens wird das Netzwerkprotokoll mit einem Client-Konfigurationsprogramm des Servers eingerichtet. Bei ODBC müssen Sie möglicherweise auch die Treiberkonfiguration mit dem ODBC Driver Manager überprüfen.

Das erstmalige Einrichten einer Verbindung zwischen Client und Server kann verschiedene Probleme mit sich bringen. Sollten Schwierigkeiten auftreten, überprüfen Sie die folgenden Punkte:

- Ist die clientseitige Verbindung Ihres Servers richtig konfiguriert?
- Falls Sie TCP / IP verwenden, ist folgendes zu klären:
  - Haben Sie Ihre TCP / IP-Kommunikationssoftware installiert? Ist die richtige WINSOCK.DLL installiert?
  - Ist die IP-Adresse des Servers in der HOSTS-Datei des Client eingetragen?
  - Sind die Domain Name Services (DNS) richtig konfiguriert?
  - Können Sie den Server mit dem Hilfsprogramm PING erreichen?
- Befinden sich die DLLs für Ihre Verbindung und die Datenbanktreiber im Suchpfad?

Weitere Informationen zur Fehlersuche finden Sie in der Online-Hilfe zu SQL Links und in der Dokumentation Ihres Servers.

ODBC

In einer Anwendung können ODBC-Datenquellen (z. B. Btrieve) verwendet werden. Für eine Verbindung über einen ODBC-Treiber benötigen Sie

- einen mitgelieferten ODBC-Treiber,
- den Microsoft ODBC Driver Manager und
- das Hilfsprogramm BDE-Konfiguration.

Um einen BDE-Alias für eine Verbindung über einen ODBC-Treiber einzurichten, verwenden Sie die BDE-Konfiguration. Weitere Informationen finden Sie in der Online-Hilfe.

Eine Verbindung zu einem Datenbankserver trennen

Es gibt zwei Möglichkeiten, die Verbindung einer Datenbank-Komponente mit einem Server zu trennen:

- Setzen Sie die Eigenschaft *Connected* auf *False*.
- Rufen Sie die Methode *Close* auf.

Wenn Sie *Connected* auf *False* setzen, wird die Methode *Close* aufgerufen. *Close* schließt alle geöffneten Datenmengen und trennt die Verbindung zum Server. Ein Beispiel:

```
Database1.Connected := False;
```

Hinweis *Close* trennt die Verbindung auch dann, wenn *KeepConnection* auf *True* gesetzt ist.

Datenmengen ohne Trennen der Server-Verbindung schließen

Sollen alle geöffneten Datenmengen geschlossen werden, ohne die Verbindung mit einem Datenbankserver zu trennen, gehen Sie folgendermaßen vor:

1. Setzen Sie die Eigenschaft *KeepConnection* der Datenbank-Komponente auf *True*.
2. Rufen Sie die Methode *CloseDataSets* der Datenbank-Komponente auf.

Durch die Datenmengen einer Datenbank-Komponente iterieren

Eine Datenbank-Komponente verfügt über zwei Eigenschaften, mit denen eine Anwendung durch die zugeordneten Datenmengen iterieren kann, nämlich *DataSets* und *DataSetCount*.

DataSets ist ein indiziertes Array aller aktiven Datenmengen (*TTable*, *TQuery* und *TStoredProc*). Eine Datenmenge ist aktiv, wenn sie aktuell geöffnet ist. Die Nur-Lesen-Eigenschaft *DataSetCount* ist ein Integer-Wert, der die Anzahl der aktiven Datenmengen angibt.

Sie können *DataSets* zusammen mit *DataSetCount* verwenden, um die aktiven Datenmengen in einer Schleife zu verarbeiten. Im folgenden Beispiel wird bei allen Datenmengen des Typs *TTable* die Eigenschaft *CachedUpdates* auf *True* gesetzt:

```
var
      I: Integer;
begin
      for I := 0 to DataSetCount - 1 do
         if DataSets[I] is TTable then
         DataSets[I].CachedUpdates := True;
end;
```

Interaktionen zwischen Datenbank- und Sitzungskomponenten

Im allgemeinen stellen die Eigenschaften einer Sitzungskomponente (z. B. *KeepConnection*) globale Standardfunktionen für alle zur Laufzeit erzeugten temporären Datenbank-Komponenten bereit.

Bei den Methoden einer Sitzungskomponente ist dies jedoch anders. Die Methoden von *TSession* wirken sich auf alle Datenbank-Komponenten unabhängig von ihrem Status aus. So schließt beispielsweise die Sitzungsmethode *DropConnections* alle Datenmengen der Datenbank-Komponenten einer Sitzung und beendet danach alle Datenbankverbindungen, selbst wenn die Eigenschaft *KeepConnection* einzelner Datenbanken *True* ist.

Die Methoden einer Datenbank-Komponente wirken sich nur auf ihre eigenen Datenmengen aus. Angenommen, die Datenbank-Komponente *Database1* ist mit der Standardsitzung verbunden. Die Methode *Database1.CloseDataSets()* schließt dann nur die Datenmengen, die Database1 zugeordnet sind. Geöffnete Datenmengen, die zu anderen Datenbank-Komponenten innerhalb der Standardsitzung gehören, bleiben geöffnet.

Datenbank-Komponenten in Datenmodulen

Datenbank-Komponenten können jederzeit in einem Datenmodul plaziert werden. Wenn Sie jedoch ein Datenmodul mit einer Datenbank-Komponente in die Objektablage aufnehmen, damit es von anderen Benutzern wiederverwendet werden kann, müssen Sie die Eigenschaft *HandleShared* der Komponente auf *True* setzen, um globale Namespace-Konflikte zu vermeiden.

SQL-Anweisungen mit einer TDatabase-Komponente ausführen

Einfache SQL-Anweisungen können direkt aus einer *TDatabase*-Komponente mit Hilfe ihrer Methode Execute ausgeführt werden. Die Routine wird hauptsächlich zum Ausführen von DDL-Anweisungen verwendet, mit denen die Datendefinition (z. B. Tabellenstruktur) geändert werden kann. Diese SQL-Anweisungen geben keine Ergebnismenge zurück und arbeiten nur mit den Metadaten einer Datenbank. Mit *Execute* können aber auch DML-Anweisungen ausgeführt werden, die Ergebnismengen zurückgeben und mit den Daten in der Datenbank arbeiten.

Mit jedem Aufruf der Methode *Execute* kann jeweils nur eine SQL-Anweisung ausgeführt werden. Um mehrere Anweisungen auszuführen, gehen Sie folgendermaßen vor: ersetzen Sie den Inhalt des Parameters *SQL* durch eine neue SQL-Anweisung, und rufen Sie *Execute* erneut auf. Wiederholen Sie diese Schritte für jede weitere Anweisung. Möglicherweise müssen dabei der jeweiligen Situation entsprechend noch andere Änderungen vorgenommen werden, wie beispielsweise Ändern der Parameterwerte zwischen den Aufrufen.

SQL-Anweisungen ohne Ergebnismengen ausführen

Alle DDL- (Data Definition Language = Datendefinitionssprache) und manche DML- (Data Manipulation Language = Datenbearbeitungssprache) SQL-Anweisungen geben keine Ergebnismenge zurück. Sie werden aufgerufen, führen Operationen mit den Metadaten (DDL-Anweisungen) oder Daten (DML-Anweisungen) durch und beenden dann die Interaktion mit der Datenbank. Zu den DDL-Anweisungen gehören CREATE INDEX, ALTER TABLE und DROP DOMAIN. Beispiele für DML-Anweisungen, die keine Ergebnismengen zurückgeben, sind INSERT, DELETE und UPDATE.

Diese Anweisungen können durch einen Aufruf der Methode *TDatabase.Execute* ausgeführt werden. Dabei wird die SQL-Anweisung in einem Wert des Typs String (als Variable oder Literal) mit dem Parameter SQL an *Execute* übergeben. Werden keine Parameter in der SQL-Anweisung verwendet, übergeben Sie den Wert **nil** im Parameter Params. Informationen über das Verwenden von Parametern mit der Methode *Execute* finden Sie im Abschnitt »SQL-Anweisungen mit Parametern ausführen« auf Seite 17-13. Da die Anweisungen keine Ergebnismenge zurückgeben, übergeben Sie den Wert **nil** im Parameter *Cursor*.

Im folgenden Beispiel wird die SQL-Anweisung CREATE TABLE (DDL) ohne Parameter mit der Methode *Execute* ausgeführt.

```
procedure TDataForm.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  Database1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  Database1.Execute(SQLstmt, nil, False, nil);
end;
```

Im nächsten Beispiel wird die SQL-Anweisung INSERT (DML) ohne Parameter ausgeführt.

```
procedure TDataForm.InsertRecordButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  Database1.Connected := True;
  SQLstmt := 'INSERT INTO Customer ' +
    '(Custno, Company, State) ' +
    'VALUES (9999, "John Doe Rentals", "CA")';
  Database1.Execute(SQLstmt, nil, False, nil);
end;
```

SQL-Anweisungen mit Ergebnismengen ausführen

Nur DML-Anweisungen, in denen die Anweisung SELECT verwendet wird, geben Ergebnismengen zurück. Die Anweisung SELECT kann sich auf eine Tabelle oder eine Stored Procedure beziehen, die eine Ergebnismenge zurückgibt.

Um eine dieser Anweisungen auszuführen, rufen Sie die Methode *TDatabase.Execute* auf und stellen eine Datenmengenkomponente für die Ergebnismenge bereit. Dabei wird die SQL-Anweisung in einem Wert des Typs String (als Variable oder Literal) mit dem Parameter SQL an *Execute* übergeben. Werden keine Parameter in der Anweisung verwendet, übergeben Sie den Wert **nil** im Parameter *Params*. Informationen über das Verwenden von Parametern mit der Methode *Execute* finden Sie im Abschnitt »SQL-Anweisungen mit Parametern ausführen« auf Seite 17-13.

Um die Daten der Ergebnismenge verfügbar zu machen, die durch den Aufruf der Methode *Execute* von der SQL-Anweisung zurückgegeben wurde, stellen Sie eine bereits vorhandene Datenmengenkomponente (z. B. *TTable*) bereit. Es wird auch eine Variable des Typs *hDBICur* (Handle für einen BDE-Cursor) benötigt. Sie müssen in die **uses**-Klausel der Quelltextdatei mit dem *Execute*-Aufruf die Unit „BDE“ aufnehmen, damit Sie diesen Typ verwenden können. Übergeben Sie dann die dereferenzierte *hDBICur*-Variable im Parameter *Cursor*. Weisen Sie nach dem Aufruf von *Execute* das Handle des BDE-Cursors der Datenmengeneigenschaft *Handle* zu. Dazu muß aber der Typ der Datenmenge explizit in *TDBDataSet* umgewandelt werden, da die Eigenschaft *Handle* nur in dieser Klasse geändert werden kann.

Im folgenden Beispiel wird eine SELECT-Anweisung mit *Execute* ausgeführt. Anschließend wird die zurückgegebene Ergebnismenge mit Hilfe der *TTable*-Komponente *Table1* bereitgestellt:

```
procedure TDataForm.SELECT_NoParamsButtonClick(Sender: TObject);
var
   SQLstmt: String;
   Cursor: hDBICur;
begin
   Database1.Connected := True;
   SQLstmt := 'SELECT Company, State ' +
    'FROM Customer ' +
    'ORDER BY State, Company';
   Database1.Execute(SQLstmt, nil, False, @Cursor);
   Table1.Close;
   (Table1 as TDBDataSet).Handle := Cursor;
end;
```

Wenn die Datenmenge zuvor für eine andere Ergebnismenge verwendet wurde, muß diese vor der neuen Zuweisung mit *Close* geschlossen werden.

Die im Parameter *Cursor* zurückgegebene Ergebnismenge besitzt immer den Status »Nur lesen«.

SQL-Anweisungen mit Parametern ausführen

Einigen SQL-Anweisungen können Datenwerte als Parameter übergeben werden.

Übergeben Sie dazu beim Aufruf von *TDatabase.Execute* ein Objekt des Typs *TParams* im Parameter *Params*. Geben Sie die SQL-Anweisung in einem Wert des Typs *String* (als Variable oder Literal) mit dem Parameter SQL an. Wenn die Anweisung keine Ergebnismenge zurückgibt, übergeben Sie im Parameter Cursor den Wert **nil**. Wird eine Ergebnismenge zurückgegeben, finden Sie die entsprechenden Informationen im Abschnitt »SQL-Anweisungen mit Ergebnismengen ausführen« auf Seite 17-12

Die Parameter der SQL-Anweisung werden als *TParams*-Objekt im Parameter Params an *Execute* übergeben. Dabei wird jeder Parameter in einem eigenen Objekt des Typs *TParam* in Params repräsentiert. Mit der Methode *TParams.CreateParam* kann jeweils ein Parameter (*TParam*-Objekt) in das *TParams*-Objekt aufgenommen werden. Sie können dann mit Hilfe der Eigenschaften und Methoden von *TParam* den Wert des Objekts ändern, um den gewünschten SQL-Parameter an *Execute* zu übergeben.

Im folgenden Beispiel wird eine SQL-Anweisung mit einem Parameter (*StateParam*) ausgeführt. In der Routine wird zunächst ein TParams-Objekt (*stmtParams*) erstellt. Dann wird durch einen Aufruf von *TParams.CreateParam* das *TParam*-Objekt (der Parameter) *stmtParams* zugewiesen. Dem *TParam*-Objekt wird anschließend der Wert „CA“ zugewiesen. Abschließend wird im Parameter *Params* die Methode *Execute* mit dem Objekt *stmtParams* aufgerufen.

```
procedure TDataForm.SELECT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
  Cursor: hDBICur;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'SELECT Company, State '+
      'FROM "Customer.db" ' +
      'WHERE (State = :StateParam) ' +
      'ORDER BY State, Company';
    Database1.Execute(SQLstmt, stmtParams, False, @Cursor);
    Table1.Close;
    TDBDataSet(Table1).Handle := Cursor;
  finally
    stmtParams.Free;
  end;
end;
```

Eine SQL-Anweisung mit Parametern kann mit *Execute* nur erfolgreich ausgeführt werden, wenn folgende Bedingungen erfüllt sind:

1 Die Parameter müssen in der SQL-Anweisung vorhanden sein (diesen Tokens wird ein Doppelpunkt vorangestellt).

2 Ein *TParams*-Objekt muß für die *TParam*-Objekte erstellt werden.

3 Für jeden Parameter der SQL-Anweisung muß im *TParams*-Objekt ein Objekt des Typs *TParam* erstellt werden.

4 Das *TParams*-Objekt muß im Parameter *Params* an *Execute* übergeben werden.

Wird für einen Parameter der SQL-Anweisung kein *TParam*-Objekt erstellt, kann kein Wert übergeben werden, und die Anweisung führt möglicherweise zu einem Fehler (abhängig von der Datenbank). Wenn Sie ein *TParam*-Objekt ohne entsprechenden Parameter in der SQL-Anweisung erstellen, wird eine Exception ausgelöst.

Kapitel

18

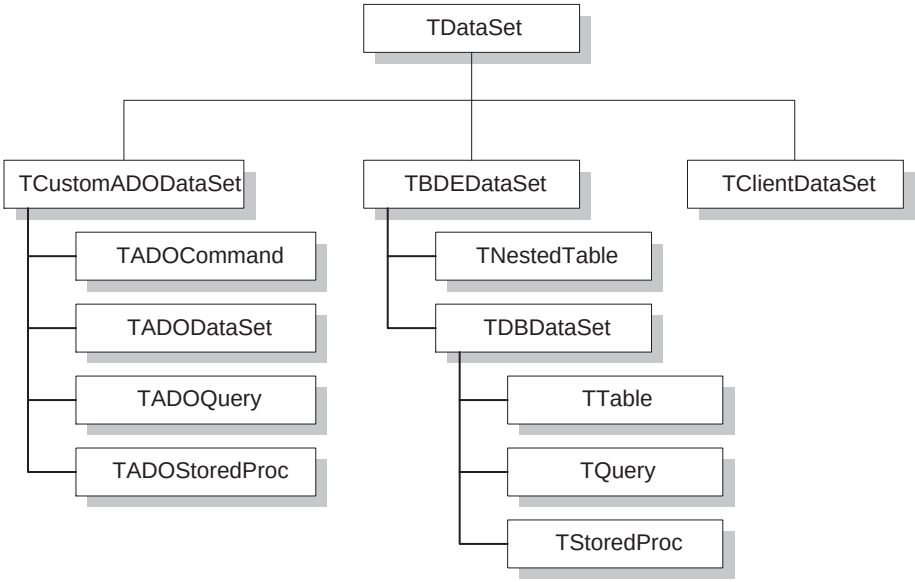
Datenmengen

In Delphi bildet die Familie der Datenmengenobjekte die Basis für den Zugriff auf Daten. Ihre Anwendungen nutzen Datenmengen für jeden Datenbankzugriff. Im allgemeinen repräsentiert ein Datenmengenobjekt eine spezielle Datenbanktabelle, eine Abfrage oder eine gespeicherte Prozedur, die auf eine Datenbank zugreift.

Alle Datenmengenobjekte, die Sie in Ihren Datenbankanwendungen einsetzen, sind Nachkommen des Datenmengenobjekts *TDataSet*, dessen Datenfelder, Eigenschaften, Ereignisse und Methoden sie erben. In diesem Kapitel werden die Merkmale von *TDataSet* erläutert, die alle Datenmengenobjekte erben. Sie müssen die Bedeutung dieser Merkmale kennen, um mit Datenmengenobjekten arbeiten zu können.

Abbildung 18.1 illustriert die hierarchischen Beziehungen zwischen Datenmengenkomponenten.

Abbildung 18.1 Die Datenmengenhierarchie in Delphi



Die Komponente TDataSet

TDataSet ist der Vorfahr aller Datenmengenobjekte, die Sie in Ihren Anwendungen einsetzen. Die Komponente definiert eine Menge von Datenfeldern, Eigenschaften, Ereignissen und Methoden, die von allen Datenmengenobjekten gemeinsam verwendet werden. *TDataSet* ist eine virtualisierte Datenmenge, weil viele ihrer Eigenschaften und Methoden als **virtual** oder **abstract** deklariert sind. Eine *virtuelle Methode* ist eine Funktions- oder Prozedurdeklaration, deren Implementierung in abgeleiteten Objekten überschrieben werden kann und normalerweise auch tatsächlich überschrieben wird. Eine *abstrakte Methode* ist eine Funktions- oder Prozedurdeklaration ohne tatsächliche Implementierung. Die Deklaration stellt einen Prototyp dar, der eine Methode (und ihre Parameter und gegebenenfalls den Rückgabetyp) beschreibt. Die Methode muß dann in allen abstammenden Datenmengenobjekten implementiert werden, wobei sich die Implementierungen voneinander unterscheiden können.

Da *TDataSet* **abstract**-Methoden enthält, läßt sich die Komponente nicht direkt in einer Anwendung einsetzen, ohne einen Laufzeitfehler zu verursachen. Erstellen Sie statt dessen Instanzen von Nachkommen von *TDataSet*, z.B. *TTable*, *TQuery*, *TStoredProc* und *TClientDataSet*, und verwenden Sie diese in Ihrer Anwendung. Sie können aber auch eigene Datenmengenobjekte von der Klasse *TDataSet* oder ihren Nachkommen ableiten und Implementationen für deren **abstract**-Methoden schreiben.

TDataSet definiert viele Features, die alle Datenmengenobjekte gemeinsam haben. Dazu gehören beispielsweise die grundlegende Struktur aller Datenmengen, ein Array mit *TField*-Komponenten, die den tatsächlichen Spalten in einer oder mehreren Datenbanktabellen entsprechen, Lookup-Felder und berechnete Felder, die von der

Anwendung bereitgestellt werden. Informationen über *TField*-Komponenten finden Sie in Kapitel 19, »Felder«.

In diesem Kapitel werden folgende Themen behandelt:

- Arten von Datenmengen
- Datenmengen öffnen und schließen
- Den Status von Datenmengen bestimmen und einstellen
- Durch Datenmengen navigieren
- Datenmengen durchsuchen
- Teilmengen von Daten mit Hilfe von Filtern anzeigen und bearbeiten
- Daten bearbeiten
- Ereignisse für Datenmengen behandeln
- BDE-Datenmengen

Arten von Datenmengen

Dieses Kapitel befaßt sich mit dem Konzept, das allen Datenmengenobjekten zugrundeliegt. Die Erläuterungen richten sich an Programmierer, die eigene, benutzerdefinierte Datenmengenobjekte entwickeln wollen, ohne mit der Borland Database Engine (BDE) oder mit ActiveX Data Objects (ADO) zu arbeiten.

Informationen zur Entwicklung konventioneller zweischichtiger Client / Server-Datenbankanwendungen, welche die Borland Database Engine (BDE) nutzen, finden Sie weiter hinten in diesem Kapitel unter »BDE-Datenmengen im Überblick«. Sie finden dort neben einer Erläuterung der Objekte *TBDEDataSet* und *TDBDataSet* auch eine Beschreibung der Funktionen, die von den am häufigsten in Datenbankanwendungen verwendeten Datenmengenkomponenten *TQuery*, *TStoredProc* und *TTable* für die gemeinsame Nutzung bereitgestellt werden.

Mit einigen Delphi-Versionen können Sie mehrschichtige Datenbankanwendungen entwickeln, die verteilte Datenmengen nutzen. Informationen zur Verwendung von Client-Datenmengen in mehrschichtigen Anwendungen finden Sie in Kapitel 14, »Mehrschichtige Anwendungen erstellen«. Dieses Kapitel enthält Einzelheiten zur Verwendung von *TClientDataSet* und zur Verbindung eines Client mit einem Anwendungsserver.

Datenmengen öffnen und schließen

Damit Daten aus einer Tabelle gelesen bzw. in eine Tabelle geschrieben oder über eine Abfrage ermittelt werden können, muß eine Anwendung zuerst eine Datenmenge öffnen. Dazu gibt es zwei Möglichkeiten:

- Setzen Sie die Eigenschaft *Active* der Datenmenge entweder zur Entwurfszeit im Objektinspektor oder zur Laufzeit im Quelltext auf *True*:

```
CustTable.Active := True;
```

- Rufen Sie zur Laufzeit die Methode *Open* der Datenmenge auf:

```
CustQuery.Open;
```

Zum Schließen einer Datenmenge gibt es ebenfalls zwei Möglichkeiten:

- Setzen Sie die Eigenschaft *Active* der Datenmenge entweder zur Entwurfszeit im Objektinspektor oder zur Laufzeit im Quelltext auf *False*:

```
CustQuery.Active := False;
```

- Rufen Sie zur Laufzeit die Methode *Close* der Datenmenge auf:

```
CustTable.Close;
```

Sie müssen eine Datenmenge unter Umständen erst schließen, wenn Sie bestimmte Eigenschaften der Menge ändern wollen, wie beispielsweise *TableName* in einer *TTable*-Komponente. Außerdem kann es sein, daß Sie eine Datenmenge aus anwendungsspezifischen Gründen zur Laufzeit schließen müssen.

Den Status von Datenmengen bestimmen und einstellen

Über den Status – oder Modus – einer Datenmenge wird festgelegt, was mit den Daten der Menge geschehen kann. Wenn eine Datenmenge beispielsweise geschlossen ist, hat sie den Status *dsInactive*. In diesem Fall ist kein Zugriff auf die Daten möglich. Zur Laufzeit kann die Nur-Lesen-Eigenschaft *State* der Datenmenge überprüft und dadurch deren aktueller Status festgestellt werden. In der folgenden Tabelle sind die möglichen Werte für die Eigenschaft *State* und ihre Bedeutung zusammengestellt:

Tabelle 18.1 Werte für die Eigenschaft *State* einer Datenmenge

Wert	Status	Bedeutung
dsInactive	*Inactive*	Die Datenmenge ist geschlossen. Auf die Daten ist kein Zugriff möglich.
dsBrowse	*Browse*	Die Datenmenge ist geöffnet. Ihre Daten können angezeigt, aber nicht geändert werden. Dies ist der voreingestellte Status einer geöffneten Datenmenge
dsEdit	*Edit*	Die Datenmenge ist geöffnet. Der aktuelle Datensatz kann geändert werden.
dsInsert	*Insert*	Die Datenmenge ist geöffnet. Ein neuer Datensatz kann ein- oder angefügt werden.
dsSetKey	*SetKey*	Nur bei *TTable* und *TClientDataSet*. Die Datenmenge ist geöffnet. Für Bereiche und *GotoKey*-Operationen können Bereiche und Schlüsselwerte gesetzt werden.
dsCalcFields	*CalcFields*	Die Datenmenge ist geöffnet. Dieser Status zeigt an, daß das Ereignis *OnCalcFields* »unterwegs« ist. Änderungen an Feldern, die nicht berechnet sind, werden verhindert.
dsCurValue	*CurValue*	Nur für den internen Gebrauch.
dsNewValue	*NewValue*	Nur für den internen Gebrauch.

Tabelle 18.1 Werte für die Eigenschaft *State* einer Datenmenge (Fortsetzung)

Wert	Status	Bedeutung
dsOldValue	*OldValue*	Nur für den internen Gebrauch.
dsFilter	*Filter*	Die Datenmenge ist geöffnet. Dieser Status zeigt an, daß eine Filteroperation abläuft. Eine bestimmte Teilmenge von Daten kann zwar angezeigt, aber nicht geändert werden.

Wenn eine Anwendung eine Datenmenge öffnet, befindet sich diese per Voreinstellung im Modus *dsBrowse*. Der Status einer Datenmenge ändert sich, wenn in der Anwendung Daten bearbeitet werden. Eine geöffnete Datenmenge wechselt ihren Status in Abhängigkeit

- vom Quelltext der Anwendung oder
- vom integrierten Verhalten datensensitiver Komponenten.

Um eine Datenmenge in den Status *dsBrowse*, *dsEdit*, *dsInsert* oder *dsSetKey* zu versetzen, rufen Sie die dem Statusnamen entsprechende Methode auf. Der folgende Quelltext versetzt beispielsweise *CustTable* in den Status *dsInsert*, akzeptiert Benutzereingaben für einen neuen Datensatz und schreibt den neuen Datensatz in die Datenbank:

```
CustTable.Insert; { Die Anwendung setzt den Datenmengenstatus explizit auf Insert. }
AddressPromptDialog.ShowModal;
if AddressPromptDialog.ModalResult := mrOK then
   CustTable.Post; { Delphi setzt den Status nach erfolgreicher Ausführung auf dsBrowse }
else
   CustTable.Cancel; { Delphi setzt den Status nach einem Abbruch auf dsBrowse }
```

Das Beispiel zeigt auch, daß die Datenmenge automatisch in den Status *dsBrowse* versetzt wird, wenn folgende Bedingungen zutreffen:

- Die Methode *Post* trägt einen Datensatz erfolgreich in die Datenbank ein (wenn *Post* fehlschlägt, bleibt der Status unverändert).
- Die Methode *Cancel* wird aufgerufen.

Einige Statusarten können nicht direkt festgelegt werden. Um eine Datenmenge beispielsweise in den Status *dsInactive* zu versetzen, müssen Sie ihrer Eigenschaft *Active* den Wert *False* zuweisen oder die Methode *Close* der Datenmenge aufrufen. Die folgenden Anweisungen sind gleichbedeutend:

```
CustTable.Active := False;

CustTable.Close;
```

Die anderen Statusarten (*dsCalcFields*, *dsCurValue*, *dsNewValue*, *dsOldValue* und *dsFilter*) können nicht von der Anwendung zugewiesen werden. Statt dessen wird die Änderung bei Bedarf automatisch vorgenommen. Beispielsweise wird *dsCalcFields* gesetzt, wenn das Ereignis *OnCalcFields* der Datenmenge aufgerufen wird. Sobald das Ereignis ausgeführt ist, wird die Datenmenge wieder in den vorherigen Status zurückversetzt.

Hinweis Sobald sich der Status einer Datenmenge ändert, wird das Ereignis *OnStateChange* für alle Datenquellenkomponenten aufgerufen, die mit der Datenmenge verbunden

sind. Informationen über die Datenquellenkomponenten und das Ereignis *OnStateChange* finden Sie unter »TDataSource-Ereignisse verwenden« auf Seite 26-8.

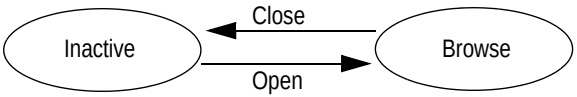
In den folgenden Abschnitten finden Sie Erläuterungen zu jedem Status. Sie erfahren, wie und wann ein Status gesetzt wird, wie die verschiedenen Statusarten miteinander in Beziehung stehen und wo Sie bei Bedarf weitere Informationen erhalten können.

Eine Datenmenge deaktivieren

Eine Datenmenge ist deaktiviert, wenn sie geschlossen ist. Sie können dann nicht auf ihre Datensätze zugreifen. Zur Entwurfszeit bleibt eine Datenmenge geschlossen, bis Sie ihre Eigenschaft *Active* auf *True* setzen. Zur Laufzeit ist eine Datenmenge anfänglich geschlossen, bis eine Anwendung sie über einen Aufruf der Methode *Open* öffnet oder ihre Eigenschaft *Active* auf *True* setzt.

Wenn eine deaktivierte Datenmenge geöffnet wird, erhält sie automatisch den Status *dsBrowse*. Das folgende Diagramm zeigt die Beziehung zwischen diesen Statusarten und den Methoden, mit denen sie festgelegt werden.

Abbildung 18.2 Die Beziehung zwischen dem Status *Inactive* und dem Status *Browse*.



Zur Deaktivierung einer Datenmenge rufen Sie ihre Methode *Close* auf. Sie können Behandlungsroutinen für die Ereignisse *BeforeClose* und *AfterClose* schreiben, die auf die Methode *Close* einer Datenmenge reagieren. Wenn eine Datenmenge beispielsweise beim Aufruf der Methode *Close* in der Anwendung den Status *dsEdit* oder *dsInsert* hat, sollten Sie den Benutzer dazu auffordern, die anstehenden Änderungen zu speichern oder zu verwerfen, bevor die Datenmenge geschlossen wird. Der folgende Quelltext zeigt eine derartige Behandlungsroutine:

```
procedure CustTable.VerifyBeforeClose(DataSet: TDataSet)
begin
   if (CustTable.State = dsEdit) or (CustTable.State = dsInsert) then
   begin
    if MessageDlg('Änderungen vor dem Beenden speichern?', mtConfirmation, mbYesNo, 0) =
mrYes then
      CustTable.Post;
    else
      CustTable.Cancel;
   end;
end;
```

Um eine Prozedur zur Entwurfszeit mit dem Ereignis *BeforeClose* einer Datenmenge zu verknüpfen, führen Sie folgende Schritte aus:

1. Wählen Sie die Tabelle im Datenmodul (oder im Formular) aus.
2. Klicken Sie auf die Registerkarte *Ereignisse* im Objektinspektor.

3 Geben Sie den Namen der Prozedur für das Ereignis *BeforeClose* ein (oder wählen Sie ihn aus der Dropdown-Liste).

Datenmengen durchsuchen

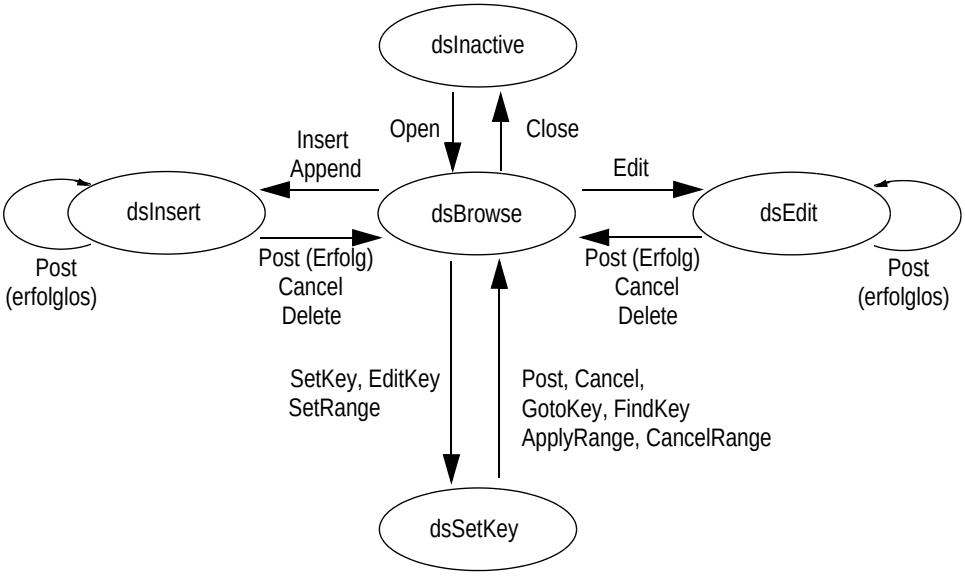
Sobald eine Anwendung eine Datenmenge öffnet, wird sie automatisch in den Status *dsBrowse* versetzt. In diesem Status können die Datensätze einer Datenmenge angezeigt, aber nicht geändert oder neu eingefügt werden. In erster Linie wird dieser Status dazu verwendet, um in einer Datenmenge durch die Datensätze zu blättern. Informationen hierzu finden Sie im Abschnitt »Durch Datenmengen navigieren« auf Seite 18-11.

Ausgehend vom Status *dsBrowse* können alle anderen Statusarten für die Datenmenge gesetzt werden. Wenn beispielsweise die Methoden *Insert* oder *Append* für eine Datenmenge aufgerufen werden, ändert sich ihr Status von *dsBrowse* zu *dsInsert* (beachten Sie aber, daß andere Faktoren oder Eigenschaften der Datenmenge, wie z.B. *CanModify*, diese Änderung unter Umständen verhindern). Der Aufruf von *SetKey* zur Suche nach Datensätzen versetzt die Datenmenge in den Status *dsSetKey*. Informationen über das Einfügen und Anhängen von Datensätzen an die Datenmenge finden Sie im Abschnitt »Daten bearbeiten« auf Seite 18-24.

Mit Hilfe von zwei Methoden, die mit allen Datenmengen verknüpft sind, kann die Datenmenge wieder in den Status *dsBrowse* zurückversetzt werden. Die Methode *Cancel* beendet die aktuelle Bearbeiten-, Einfügen- oder Suchen-Operation und bringt die Datenmenge wieder in den Status *dsBrowse*. Die Methode *Post* versucht, die Änderung in die Datenbank einzutragen, und versetzt die Datenmenge im Erfolgsfall wieder in den Status *dsBrowse*. Wenn *Post* fehlschlägt, bleibt der aktuelle Status erhalten.

Die folgende Abbildung zeigt die Beziehung zwischen dem Status *dsBrowse* und den anderen Statusarten der Datenmenge, die in Ihren Anwendungen gesetzt werden können, und zu den Methoden, mit denen diese Modi festgelegt werden.

Abbildung 18.3 Die Beziehung zwischen dsBrowse und anderen Modi



Die Bearbeitung von Datenmengen ermöglichen

Eine Datenmenge muß sich im Status *dsEdit* befinden, damit Datensätze in einer Anwendung geändert werden können. In Ihrem Quelltext läßt sich eine Datenmenge mit der Methode *Edit* in diesen Modus versetzen, wenn die Nur-Lesen-Eigenschaft *CanModify* für die Datenmenge *True* ist. *CanModify* ist *True*, wenn für die der Datenmenge zugrundeliegende Datenbank Lese- und Schreibberechtigung besteht.

In Formularen der Anwendung können bestimmte datensensitive Steuerelemente eine Datenmenge unter folgenden Bedingungen automatisch in den Status *dsEdit* versetzen:

- Die Eigenschaft *ReadOnly* des Steuerelements hat den Wert *False* (Voreinstellung).
- Die Eigenschaft *AutoEdit* der Datenquelle für das Steuerelement hat den Wert *True*.
- *CanModify* hat für die Datenmenge den Wert *True*.

Wichtiger Hinweis Wenn die Eigenschaft *ReadOnly* bei *TTable*-Komponenten (und *TQuery*-Komponenten, falls *RequestLive False* ist) den Wert *True* hat, ist *CanModify False*, wodurch die Bearbeitung von Datensätzen verhindert wird.

Hinweis Auch wenn eine Datenmenge den Status *dsEdit* hat, lassen sich die Datensätze in SQL-Datenbanken unter Umständen nicht bearbeiten, weil der Benutzer nicht über die entsprechenden SQL-Zugriffsrechte verfügt.

Sie können eine Datenmenge im Quelltext vom Status *dsEdit* wieder in den Status *dsBrowse* versetzen, indem Sie eine der Methoden *Cancel*, *Post* oder *Delete* aufrufen. *Cancel* verwirft die Änderungen im aktuellen Feld oder Datensatz. *Post* versucht, ei-

nen geänderten Datensatz in die Datenmenge zu schreiben. Die Methode versetzt im Erfolgsfall die Datenmenge wieder in den Status *dsBrowse*. Wenn *Post* die Änderungen nicht eintragen kann, bleibt die Datenmenge im Status *dsEdit*. *Delete* versucht, den aktuellen Datensatz aus der Datenmenge zu entfernen. Die Methode versetzt die Datenmenge im Erfolgsfall wieder in den Status *dsBrowse*. Wenn *Delete* fehlschlägt, bleibt die Datenmenge im Status *dsEdit*.

Im Bearbeitungsmodus rufen datensensitive Steuerelemente automatisch die Methode *Post* auf, sobald ein Benutzer eine Aktion ausführt, durch die der aktuelle Datensatz geändert wird (beispielsweise, wenn zu einem anderen Datensatz in einem Gitter gewechselt wird). Der Aufruf von *Post* erfolgt auch dann, wenn das Steuerelement den Fokus abgibt (beispielsweise beim Wechsel zu einem anderen Steuerelement im Formular).

Eine vollständige Erläuterung zur Bearbeitung von Feldern und Datensätzen in Datenmengen finden Sie im Abschnitt »Daten bearbeiten« auf Seite 18-24.

Das Einfügen neuer Datensätze ermöglichen

Eine Datenmenge muß in den Status *dsInsert* versetzt werden, bevor neue Datensätze eingefügt werden können. Sie können im Quelltext die Methoden *Insert* oder *Append* verwenden, um eine Datenmenge in den Status *dsInsert* zu versetzen, falls die Nur-Lesen-Eigenschaft *CanModify* der Datenmenge den Wert *True* hat. *CanModify* ist *True*, wenn für die der Datenmenge zugrundeliegende Datenbank Lese- und Schreibberechtigung besteht.

In Formularen Ihrer Anwendung können datensensitive Gitter und Navigatoren eine Datenmenge unter folgenden Bedingungen in den Status *dsInsert* versetzen:

- Die Eigenschaft *ReadOnly* des Steuerelements hat den Wert *False* (Voreinstellung).
- Die Eigenschaft *AutoEdit* der Datenquelle für das Steuerelement hat den Wert *True*.
- *CanModify* hat für die Datenmenge den Wert *True*.

Wichtig Wenn die Eigenschaft *ReadOnly* bei *TTable*-Komponenten (und *TQuery*-Komponenten, falls *RequestLive False* ist) den Wert *True* hat, ist *CanModify False*, wodurch die Bearbeitung von Datensätzen verhindert wird.

Hinweis Auch wenn eine Datenmenge den Status *dsInsert* hat, lassen sich die Datensätze in SQL-Datenbanken unter Umständen nicht bearbeiten, weil der Benutzer nicht über die entsprechenden SQL-Zugriffsrechte verfügt.

Sie können eine Datenmenge im Quelltext vom Status *dsInsert* wieder in den Status *dsBrowse* versetzen, indem Sie eine der Methoden *Cancel*, *Post* oder*Delete* aufrufen. *Delete* und *Cancel* verwerfen den neuen Datensatz. Die Methode *Post* versucht, den neuen Datensatz in die Datenmenge zu schreiben. Die Methode versetzt die Datenmenge im Erfolgsfall wieder in den Status *dsBrowse*. Wenn *Post* den Datensatz nicht eintragen kann, bleibt die Datenmenge im Status *dsInsert*.

Im Einfügemodus rufen datensensitive Steuerelemente automatisch die Methode *Post* auf, sobald ein Benutzer eine Aktion ausführt, durch die der aktuelle Datensatz

geändert wird (beispielsweise, wenn zu einem anderen Datensatz in einem Gitter gewechselt wird).

Eine vollständige Erläuterung zum Einfügen und Anhängen von Datensätzen finden Sie im Abschnitt »Daten bearbeiten« auf Seite 18-24.

Auf Indizes basierende Suchvorgänge und Bereiche in Tabellen

Mit Hilfe der Methoden *Locate* und *Lookup* von *TDataSet* können Sie in jeder Datenmenge Suchvorgänge durchführen. *TTable*-Komponenten stellen eine zusätzliche Familie von *GotoKey* und *FindKey*-Methoden bereit, mit denen nach Datensätzen gesucht werden kann. Die Suche basiert auf einem Index. Zur Verwendung dieser Methoden für Tabellenkomponenten muß sich die Komponente im Status *dsSetKey* befinden. Diesen Status können nur *TTable*-Komponenten annehmen. Zur Laufzeit können Sie eine Datenmenge mit der Methode *SetKey* in den Status *dsSetKey* versetzen. Die Methoden *GotoKey*, *GotoNearest*, *FindKey* und *FindNearest*, welche die Suchläufe ausführen, versetzen die Datenmenge nach Beendigung des Suchvorgangs wieder in den Status *dsBrowse*. Informationen über Suchvorgänge in einer Tabelle, die auf dem Index der Tabelle beruhen, finden Sie unter »Datensätze über indizierte Felder suchen« auf Seite 20-6.

Mit Hilfe von Filtern können Sie vorübergehend nur einen Teil einer Datenmenge anzeigen und bearbeiten. Informationen hierzu finden Sie unter »Teilmengen von Daten mit Hilfe von Filtern anzeigen und bearbeiten« auf Seite 18-19. *TTable*-Komponenten unterstützen außerdem zusätzliche Zugriffsmöglichkeiten auf eine Teilmenge der verfügbaren Datensätze. Um einen Bereich für eine Tabelle zu erzeugen und anzuwenden, muß sich die Tabelle im Status *dsSetKey* befinden. Informationen über die Verwendung von Bereichen finden Sie unter »Mit Teilmengen der Daten arbeiten« auf Seite 20-12.

Felder berechnen

Delphi versetzt eine Datenmenge immer dann in den Status *dsCalcFields*, wenn eine Anwendung die Ereignisbehandlungsroutine für *OnCalcFields* der Datenmenge aufruft. Dieser Status verhindert Änderungen an einer Datenmenge. Ausgenommen sind berechnete Felder, zu deren Änderung die Behandlungsroutine dient. Der Grund für die Verhinderung aller anderen Änderungen liegt darin, daß *OnCalcFields* die Werte anderer Felder zur Ableitung der Werte verwendet, die den berechneten Feldern zugewiesen werden. Änderungen in den anderen Feldern könnten deshalb die den berechneten Feldern zugewiesenen Werte ungültig machen.

Nach Beendigung der Behandlungsroutine für *OnCalcFields* wird die Datenmenge wieder in den Status *dsBrowse* versetzt.

Informationen über berechnete Felder und die Ereignisbehandlungsroutinen für *OnCalcFields* finden Sie unter »Das Ereignis OnCalcFields« auf Seite 18-30.

Datensätze filtern

Delphi versetzt eine Datenmenge in den Modus *dsFilter*, sobald eine Anwendung die Behandlungsroutine für das Ereignis *OnFilterRecord* aufruft. In diesem Status werden Änderungen an einer Datenmenge während des Filtervorgangs verhindert, so daß die Filterbedingung nicht ungültig werden kann. Informationen über Filter finden Sie unter »Teilmengen von Daten mit Hilfe von Filtern anzeigen und bearbeiten« auf Seite 18-19.

Nach Beendigung der Ereignisbehandlungsroutine für *OnFilterRecord* wird die Datenmenge wieder in den Status *dsBrowse* versetzt.

Datensätze aktualisieren

Bei der Ausführung zwischengespeicherter Aktualisierungen kann Delphi eine Datenmenge vorübergehend in den Status *dsNewValue, dsOldValue* oder *dsCurValue* versetzen. Diese Statusarten bedeuten, daß gerade auf die entsprechenden Eigenschaften einer Feldkomponente *(NewValue, OldValue* bzw. *CurValue)* zugegriffen wird, und zwar normalerweise in einer Behandlungsroutine für das Ereignis *OnUpdateError*. Ihre Anwendungen können diese Modi weder erkennen noch zuweisen.

Durch Datenmengen navigieren

In jeder aktiven Datenmenge gibt es einen Cursor oder Zeiger auf den aktuellen Datensatz der Datenmenge. Der aktuelle Datensatz (oder die aktuelle Zeile) einer Datenmenge ist der Datensatz, dessen Werte durch Bearbeiten-, Einfügen- und Löschen-Methoden geändert werden können und dessen Feldwerte aktuell von den datensensitiven Steuerelementen in einem Formular angezeigt werden, wie beispielsweise *TDBEdit, TDBLabel* oder *TDBMemo*.

Sie können den aktuellen Datensatz mit Hilfe des Cursors wechseln. Die folgende Tabelle enthält die Methoden, die Sie im Quelltext dafür verwenden können.

Tabelle 18.2 Navigationsmethoden für Datenmengen

Methode	Beschreibung
First	Setzt den Cursor auf die erste Zeile einer Datenmenge.
Last	Setzt den Cursor auf die letzte Zeile einer Datenmenge.
Next	Setzt den Cursor auf die nächste Zeile einer Datenmenge.
Prior	Setzt den Cursor auf die vorherige Zeile einer Datenmenge.
MoveBy	Bewegt den Cursor um die angegebene Anzahl von Zeilen vorwärts oder rückwärts.

Die datensensitive visuelle Komponente *TDBNavigator* kapselt diese Methoden als Schaltflächen, auf die der Benutzer klicken kann, um zur Laufzeit zwischen den Datensätzen zu navigieren. Informationen über den Navigator finden Sie in Kapitel 26, »Datensensitive Steuerelemente«.

Zusätzlich zu diesen Methoden sind in der folgenden Tabelle zwei Boolesche Eigenschaften beschrieben, die nützliche Informationen bereitstellen, wenn durch die Datensätze einer Datenmenge iteriert wird.

Tabelle 18.3 Navigationseigenschaften von Datenmengen

Eigenschaft	Beschreibung
BOF (Dateianfang)	*True:* Der Cursor befindet sich in der ersten Zeile der Datenmenge. *False:* Der Cursor befindet sich nicht in der ersten Zeile der Datenmenge.
EOF (Dateiende)	*True:* Der Cursor befindet sich in der letzten Zeile der Datenmenge. *False:* Der Cursor befindet sich nicht in der letzten Zeile der Datenmenge.

Die Methoden First und Last

Die Methode *First* verschiebt den Cursor in die erste Zeile einer Datenmenge und setzt die Eigenschaft *BOF* auf *True*. Wenn sich der Cursor bereits in der ersten Zeile befindet, führt die Methode keine Aktion aus.

Die folgende Anweisung verschiebt den Cursor zum ersten Datensatz von *CustTable*:

```
CustTable.First;
```

Die Methode *Last* verschiebt den Cursor in die letzte Zeile einer Datenmenge und setzt die Eigenschaft *EOF* auf *True*. Wenn sich der Cursor bereits in der letzten Zeile befindet, führt die Methode keine Aktion aus.

Die folgende Anweisung verschiebt den Cursor zum letzten Datensatz von *CustTable*:

```
CustTable.Last;
```

Hinweis Es kann unter Umständen erforderlich sein, den Cursor ohne Zutun des Benutzers in die erste oder letzte Zeile der Datenmenge zu verschieben. Sie sollten den Benutzern die Komponente *TDBNavigator* aber trotzdem nicht vorenthalten. Der Navigator enthält Schaltflächen, die (falls sie aktiviert und sichtbar sind) vom Benutzer dazu verwendet werden, in die erste oder letzte Zeile einer aktiven Datenmenge zu gelangen. Die Ereignisse *OnClick* dieser Schaltflächen rufen die Methoden *First* und *Last* auf. Informationen über die effiziente Verwendung des Navigators finden Sie in Kapitel 26, »Datensensitive Steuerelemente«.

Die Methoden Next und Prior

Die Methode *Next* bewegt den Cursor in der Datenmenge um eine Zeile nach vorne und setzt die Eigenschaft *BOF* auf *False*, wenn die Datenmenge nicht leer ist. Befindet sich der Cursor beim Aufruf der Methode *Next* bereits in der letzten Zeile der Datenmenge, wird keine Aktion ausgeführt.

Die folgende Anweisung bewegt den Cursor zum nächsten Datensatz von *CustTable*:

```
CustTable.Next;
```

Die Methode *Prior* bewegt den Cursor in einer Datenmenge um eine Zeile zurück und setzt die Eigenschaft *EOF* auf *False*, wenn die Datenmenge nicht leer ist. Befindet sich der Cursor beim Aufruf der Methode *Prior* bereits in der ersten Zeile der Datenmenge, wird keine Aktion ausgeführt.

Die folgende Anweisung bewegt den Cursor zum vorhergehenden Datensatz von *CustTable*:

```
CustTable.Prior;
```

Die Methode MoveBy

Mit der Methode *MoveBy* können Sie angeben, um wie viele Zeilen der Cursor in einer Datenmenge vorwärts oder rückwärts bewegt werden soll. Die Bewegung erfolgt ausgehend vom aktuellen Datensatz in dem Augenblick, in dem diese Methode aufgerufen wird. *MoveBy* setzt außerdem die Eigenschaften *BOF* und *EOF* der Datenmenge in entsprechender Weise.

Diese Funktion erwartet einen Parameter vom Typ *Integer*, der die Distanz angibt. Positive Zahlen kennzeichnen eine Vorwärts-, negative Zahlen eine Rückwärtsbewegung.

MoveBy liefert die Anzahl der Zeilen zurück, um die der Cursor bewegt wurde. Wenn Sie versuchen, ihn vor den Anfang oder hinter das Ende der Datenmenge zu positionieren, unterscheidet sich die von *MoveBy* zurückgelieferte Distanz von der gewünschten Anzahl der Zeilen. Der Grund dafür liegt darin, daß *MoveBy* stoppt, wenn der erste oder der letzte Datensatz in einer Datenmenge erreicht wird.

Die folgende Anweisung bewegt den Cursor in *CustTable* um zwei Datensätze zurück:

```
CustTable.MoveBy(-2);
```

Hinweis Wenn Sie die Methode *MoveBy* in einer Mehrbenutzer-Datenbankumgebung einsetzen, müssen Sie bedenken, daß Datenmengen dynamisch sind. Ein Datensatz, der sich im Augenblick fünf Datensätze hinter dem aktuellen Datensatz befindet, kann sich im nächsten Augenblick vier, sechs oder eine nicht bekannte Anzahl von Datensätzen weiter hinten befinden. Dies liegt daran, daß mehrere Benutzer gleichzeitig auf die Datenbank zugreifen und Daten ändern können.

Die Eigenschaften EOF und BOF

Die Laufzeit-Eigenschaften *EOF* (Dateiende) und *BOF* (Dateianfang) sind Nur-Lesen-Eigenschaften, die für die Steuerung der Navigation von Nutzen sind. Dies gilt vor allem dann, wenn Sie durch alle Datensätze einer Datenmenge iterieren wollen.

EOF

Wenn *EOF True* ist, bedeutet das, daß sich der Cursor eindeutig in der letzten Zeile einer Datenmenge befindet. *EOF* ist *True*, wenn eine Anwendung

- eine leere Datenmenge öffnet;
- die Methode *Last* einer Datenmenge aufruft;
- die Methode *Next* einer Datenmenge aufruft und die Methode fehlschlägt (weil sich der Cursor in der letzten Zeile der Datenmenge befindet);
- die Methode *SetRange* für einen leeren Bereich oder eine leere Datenmenge aufruft.

In allen anderen Fällen können Sie davon ausgehen, daß *EOF False* ist.

EOF wird häufig in Schleifen abgefragt, um die iterative Bearbeitung aller Datensätze einer Datenmenge zu kontrollieren. Wenn Sie eine Datenmenge öffnen, die Datensätze enthält (oder wenn Sie die Methode *First* aufrufen), hat *EOF* den Wert *False*. Um datensatzweise durch die Datenmenge zu iterieren, erstellen Sie eine Schleife, die abgebrochen wird, sobald *EOF* den Wert *True* hat. In der Schleife rufen Sie für jeden Datensatz die Methode *Next* auf. *EOF* bleibt *False*, bis der Cursor bei einem Aufruf von *Next* bereits auf den letzten Datensatz zeigt.

Der folgende Quelltext zeigt eine Bearbeitungsschleife für eine Datenmenge namens *CustTable*:

```
CustTable.DisableControls;
try
   CustTable.First; { Auf ersten Datensatz positionieren; EOF ist False. }
   while not CustTable.EOF do { Schleife durchlaufen, bis EOF True ist. }
   begin
    { Hier die einzelnen Datensätze verarbeiten. }
    %
    CustTable.Next; { EOF wird True, wenn Next beim letzten Datensatz fehlschlägt }
   end;
finally
   CustTable.EnableControls;
end;
```

Tip Dieses Beispiel zeigt auch, wie datensensitive visuelle Steuerelemente, die mit einer Datenmenge verbunden sind, aktiviert und deaktiviert werden. Wenn Sie während der Iteration durch die Datenmenge die visuellen Steuerelemente deaktivieren, erhöht sich die Bearbeitungsgeschwindigkeit. Delphi muß in diesem Fall nämlich nicht den Inhalt des Steuerelements aktualisieren, wenn sich der aktuelle Datensatz ändert. Nach Beendigung der Iteration sollten die Steuerelemente wieder aktiviert werden, damit sie die Werte des neuen aktuellen Datensatzes anzeigen können. Beachten Sie, daß das Aktivieren der visuellen Steuerelemente in der **finally**-Klausel einer **try...finally**-Anweisung stattfindet. Dies garantiert, daß die Steuerelemente auch dann wieder aktiviert werden, wenn die Schleifenbearbeitung durch eine Exception abgebrochen wird.

BOF

Wenn *BOF* den Wert *True* hat, bedeutet dies, daß der Cursor eindeutig auf den ersten Datensatz einer Datenmenge zeigt. *BOF* wird auf *True* gesetzt, wenn eine Anwendung

- eine Datenmenge öffnet;
- die Methode *First* einer Datenmenge aufruft;
- die Methode *Prior* einer Datenmenge aufruft und die Methode fehlschlägt (weil der Cursor bereits auf den ersten Datensatz zeigt);
- die Methode *SetRange* für einen leeren Bereich oder eine leere Datenmenge aufruft.

In allen anderen Fällen können Sie davon ausgehen, daß *BOF False* ist.

Ebenso wie mit *EOF~* kann auch mit *BOF* in einer Schleife die iterative Bearbeitung von Datensätzen gesteuert werden. Der folgende Quelltext zeigt eine Bearbeitungsschleife für eine Datenmenge namens *CustTable*:

```
CustTable.DisableControls; { Verarbeitungsgeschwindigkeit erhöhen und
                             Bildschirmflackern verhindern. }
try
   while not CustTable.BOF do { Schleife durchlaufen, bis BOF True ist. }
   begin
    { Hier jeden Datensatz verarbeiten. }
    %
    CustTable.Prior; { BOF wird auf True gesetzt, wenn Prior für ersten Datensatz
                       fehlschlägt. }
   end;
finally
   CustTable.EnableControls; { Neuen aktuellen Datensatz in Steuerelementen
                               anzeigen. }
end;
```

Datensätze markieren und dorthin zurückkehren

Oft sollen die Datensätze einer Datenmenge nicht einfach sequentiell durchlaufen werden, sondern man möchte eine bestimmte Position in einer Datenmenge markieren und bei Bedarf schnell dorthin zurückkehren. *TDataSet* und ihre Nachkommen implementieren eine Positionsmarke, mit deren Hilfe Sie Datensätze markieren und später dorthin zurückkehren können. Die Positionsmarke wird durch die Eigenschaft *Bookmark* und fünf zugehörige Methoden implementiert.

Die Eigenschaft *Bookmark* gibt an, welche der in Ihrer Anwendung vorhandenen Positionsmarken aktuell ist. Die Eigenschaft enthält einen String, der die aktuelle Positionsmarke angibt. Jedesmal, wenn Sie eine weitere Positionsmarke hinzufügen, wird diese zur aktuellen Marke.

TDataSet implementiert virtuelle Positionsmarken-Methoden. Obwohl diese Methoden garantieren, daß bei ihrem Aufruf jedes von *TDataSet* abgeleitete Datenmengenobjekt einen Wert zurückliefert, sind die Rückgabewerte lediglich Standardwerte, die

keine Auskunft über die aktuelle Position geben. Nachkommen von *TDataSet*, z.B. *TBDEDataSet*, implementieren die Methoden neu, damit sie sinnvolle Werte zurückgeben. Folgende Methoden stehen zur Verfügung:

- *BookmarkValid* stellt fest, ob eine bestimmte Positionsmarke gerade verwendet wird.
- *CompareBookmarks* überprüft zwei Positionsmarken, um festzustellen, ob es sich um dieselbe Marke handelt.
- *GetBookmark* weist der aktuellen Position in der Datenmenge eine Positionsmarke zu.
- *GotoBookmark* kehrt zu einer zuvor mit *GetBookmark* erzeugten Positionsmarke zurück.
- *FreeBookmark* gibt eine zuvor mit *GetBookmark* erzeugte Positionsmarke frei.

Um eine Positionsmarke zu erzeugen, müssen Sie in der Anwendung eine Variable vom Typ *TBookmark* deklarieren, danach die Methode *GetBookmark* zur Zuweisung von Speicherplatz für die Variable aufrufen und ihr dann eine bestimmte Position in der Datenmenge zuweisen.

Bevor Sie mit *GotoBookmark* die Positionsmarke auf einen bestimmten Datensatz positionieren, können Sie die Methode *BookmarkValid* aufrufen, um festzustellen, ob die Positionsmarke auf einen Datensatz zeigt. *BookmarkValid* liefert *True* zurück, wenn die angegebene Positionsmarke auf einen Datensatz zeigt. In *TDataSet* ist *BookmarkValid* eine virtuelle Methode, die immer *False* zurückgibt, was bedeutet, daß die Positionsmarke nicht gültig ist. Nachkommen von *TDataSet* implementieren diese Methode neu, damit sie einen sinnvollen Wert zurückgibt.

Mit der Methode *CompareBookmarks* können Sie feststellen, ob die Positionsmarke, zu der Sie gelangen wollen, sich von einer anderen (oder der aktuellen) Marke unterscheidet. *TDataSet.CompareBookmarks* liefert immer 0 zurück, was bedeutet, daß die Positionsmarken identisch sind. Nachkommen von *TDataSet* implementieren diese Methode neu, damit sie einen sinnvollen Wert zurückgibt.

Sobald eine Positionsmarke übergeben wird, bewegt die Methode *GotoBookmark* den Cursor der Datenmenge zu der angegebenen Position. *TDataSet.GotoBookmark* ist eine interne, rein virtuelle Methode, die beim Aufruf einen Laufzeitfehler generiert. Nachkommen von *TDataSet* implementieren diese Methode neu, damit sie einen sinnvollen Wert zurückgibt.

FreeBookmark gibt den einer bestimmten Positionsmarke zugewiesenen Speicherplatz frei, sobald die Marke nicht mehr benötigt wird. Außerdem sollten Sie vor der Wiederverwendung einer vorhandenen Positionsmarke die Methode *FreeBookmark* aufrufen.

Der folgende Quelltext zeigt den Einsatz von Positionsmarken:

```
procedure DoSomething (const Tbl: TTable)
var
   Bookmark: TBookmark;
begin
   Bookmark := Tbl.GetBookmark; { Speicher reservieren und einen Wert zuweisen. }
   Tbl.DisableControls; { Anzeige von Datensätzen in Datensteuerelementen deaktivieren. }
```

```
    try
     Tbl.First; { Auf den ersten Datensatz in der Tabelle positionieren. }
     while not Tbl.EOF do {Durch alle Datensätze der Tabelle iterieren. }
     begin
       { Hier folgt die weitere Verarbeitung. }
       ...
      Tbl.Next;
     end;
    finally
     Tbl.GotoBookmark(Bookmark);
     Tbl.EnableControls; { Ggf. Anzeige von Datensätzen in Datensteuerelementen
                           wieder aktivieren. }
     Tbl.FreeBookmark(Bookmark); {Speicher für Positionsmarke freigeben. }
    end;
end;
```

Vor der Iteration durch die Datensätze sind alle Steuerelemente deaktiviert. Sollte während dieses Vorgangs ein Fehler auftreten, stellt die **finally**-Klausel sicher, daß die Steuerelemente wieder aktiviert werden und daß die Positionsmarke immer freigegeben wird, auch wenn die Schleife abgebrochen wird.

Datenmengen durchsuchen

Mit den generischen Suchmethoden *Locate* und *Lookup* können Sie jede Datenmenge nach bestimmten Datensätzen durchsuchen. Diese Methoden ermöglichen die Suche nach jedem beliebigen Feldtyp in jeder beliebigen Datenmenge.

Die Methode Locate

Die Methode *Locate* positioniert den Cursor auf den ersten Datensatz, der mit der angegebenen Menge von Suchkriterien übereinstimmt. In der einfachsten Form werden folgende Informationen an *Locate* übergeben: der Name des zu untersuchenden Feldes, der zu suchende Feldwert und ein Options-Flag, das angibt, ob bei der Suche die Schreibweise (groß / klein) berücksichtigt wird oder ob eine teilweise Übereinstimmung als Treffer betrachtet wird. Der folgende Quelltext positioniert den Cursor beispielsweise auf den ersten Datensatz von *CustTable*, dessen Wert im Feld *Company* »Professional Divers, Ltd«. lautet:

```
var
   LocateSuccess: Boolean;
   SearchOptions: TLocateOptions;
begin
   SearchOptions := [loPartialKey];
   LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
       SearchOptions);
end;
```

Wenn *Locate* eine Übereinstimmung findet, wird der entsprechende Datensatz zum aktuellen. *Locate* liefert in diesem Fall *True* zurück. Wird kein passender Datensatz gefunden, gibt *Locate False* zurück, und der aktuelle Datensatz wird nicht geändert.

Die Leistungsfähigkeit von *Locate* zeigt sich erst bei der Suche in mehreren Feldern unter Angabe verschiedener Suchwerte. Suchwerte sind Varianten. Sie können also in Suchkriterien unterschiedliche Datentypen verwenden. Wenn Sie in einem Such-String mehrere Felder angeben, müssen Sie die einzelnen Elemente im String durch Semikolons voneinander trennen.

Da Suchwerte Varianten sind, müssen Sie bei der Angabe mehrerer Werte entweder ein variantes Array als Argument verwenden (z.B. die Rückgabewerte der Methode *Lookup*), oder mit Hilfe der Funktion *VarArrayOf* ein Variant-Array erzeugen. Der folgende Quelltext zeigt eine Suche in mehreren Feldern mit mehreren Suchwerten und einer teilweisen Übereinstimmung:

```
with CustTable do
   Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

Locate verwendet immer die schnellste Methode zum Auffinden übereinstimmender Datensätze. Wenn die zu durchsuchenden Felder indiziert sind und der Index zu den angegebenen Suchoptionen kompatibel ist, wird er von *Locate* verwendet.

Die Methode Lookup

Die Methode *Lookup* sucht nach dem ersten Datensatz, der mit den angegebenen Suchkriterien übereinstimmt. Sobald ein passender Datensatz gefunden wird, erzwingt *Lookup* die Neuberechnung aller berechneten Felder und der mit der Datenmenge verbundenen Lookup-Felder. Anschließend werden ein oder mehrere Felder des übereinstimmenden Datensatzes zurückgegeben. *Lookup* positioniert nicht den Cursor neu, sondern liefert nur Werte zurück.

In der einfachsten Form werden folgende Informationen an *Lookup* übergeben: der Name des zu durchsuchenden Feldes, der zu suchende Feldwert und das Feld bzw. die Felder, deren Werte zurückgegeben werden sollen. Der folgende Quelltext sucht nach dem ersten Datensatz von *CustTable*, dessen Wert im Feld *Company* »Professional Divers, Ltd«. lautet, und gibt den Firmennamen, die Kontaktperson und die Telefonnummer der Firma zurück:

```
var
   LookupResults: Variant;
begin
with CustTable do
   LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company;
       Contact; Phone');
end;
```

Wenn *Lookup* einen übereinstimmenden Datensatz findet, gibt die Methode die Werte der angegebenen Felder als Varianten zurück. Wenn mehrere Ergebniswerte angefordert wurden, gibt *Lookup* ein variantes Array zurück. Existieren keine übereinstimmenden Datensätze, liefert *Lookup* eine Null-Variante zurück. Einzelheiten über variante Arrays finden Sie in der Online-Hilfe.

Die Leistungsfähigkeit von *Lookup* zeigt sich bei der Suche in mehreren Spalten unter Angabe verschiedener Suchwerte. Wenn Sie in einem Suchstring mehrere Felder oder Ergebnisfelder angeben, müssen Sie die einzelnen Felder in den String-Elementen durch Semikolons voneinander trennen.

Da Suchwerte Varianten sind, müssen Sie bei der Angabe mehrerer Werte entweder ein variantes Array als Argument übergeben (z.B. die Rückgabewerte der Methode *Lookup)*, oder mit Hilfe der Funktion *VarArrayOf* ein variantes Array erzeugen. Der folgende Quelltext zeigt eine Suche mit der Methode *Lookup* in mehreren Feldern:

```
var
   LookupResults: Variant;
begin
with CustTable do
   LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
       'Company; Addr1; Addr2; State; Zip');
end;
```

Lookup verwendet immer die schnellste Methode zum Auffinden übereinstimmender Datensätze. Wenn die zu durchsuchenden Spalten indiziert sind, wird der Index verwendet.

Teilmengen von Daten mit Hilfe von Filtern anzeigen und bearbeiten

Anwendungen sind häufig nur an einer Teilmenge aller Datensätze interessiert. So könnte beispielsweise eine Situation eintreten, in der Sie aus Ihrer Kundendatenbank nur Datensätze von Firmen ermitteln und anzeigen wollen, die ihren Sitz in einem bestimmten Bundesland haben. Sie könnten aber auch nach einem Datensatz suchen, der eine bestimmte Gruppe von Feldwerten enthält. In derartigen Fällen schränken Sie mit Hilfe eines Filters den Zugriff der Anwendung auf einen Teil der Datensätze ein.

Ein Filter legt die Bedingungen fest, die ein Datensatz erfüllen muß, damit er angezeigt wird. Filterbedingungen können in der Eigenschaft *Filter* einer Datenmenge gesetzt oder in die Ereignisbehandlungsroutine für *OnFilterRecord* aufgenommen werden. Sie basieren auf den Werten in einer beliebigen Anzahl von Feldern der Datenmenge. Dabei ist es gleichgültig, ob diese Felder indiziert sind oder nicht. Wenn beispielsweise nur die Datensätze von Firmen mit Sitz in Kalifornien angezeigt werden sollen, könnte die Filterbedingung fordern, daß die betreffenden Datensätze im Feld *State* den Wert »CA« enthalten müssen.

Hinweis Bei der Suche mit einer Filterbedingung wird jeder Datensatz der Datenmenge berücksichtigt. Bei sehr umfangreichen Datenmengen kann es effizienter sein, die Anzahl der Datensätze mit einer Abfrage einzuschränken oder einen Wertebereich für eine indizierte Tabelle zu setzen.

Filter aktivieren und deaktivieren

Zur Aktivierung eines Filters für eine Datenmenge sind drei Schritte erforderlich:

1. Erzeugen des Filters.
2. Festlegen von Filteroptionen für Filtertests, die auf Strings basieren (falls erforderlich).

3 Zuweisen von *True* an die Eigenschaft *Filter*.

Wenn ein Filter aktiviert ist, sind für die Anwendung nur diejenigen Datensätze verfügbar, welche die Filterkriterien erfüllen. Bei einem Filter handelt es sich immer um eine temporäre Erscheinung. Sie können die Filterung deaktivieren, indem Sie die Eigenschaft *Filtered* auf *False* setzen.

Filter erzeugen

Ein Filter für eine Datenmenge kann auf zwei Arten erzeugt werden:

- Durch Angabe der Filterbedingungen in der Eigenschaft *Filter* festlegen. Diese Vorgehensweise eignet sich vor allem dann, wenn *Filter* zur Laufzeit erzeugt und angewendet werden sollen.
- Durch Erstellung einer Behandlungsroutine für das Ereignis *OnFilterRecord*. Damit lassen sich sowohl einfache als auch sehr komplexe Filterbedingungen definieren. Mit *OnFilterRecord* werden die Filterbedingungen zur Entwurfszeit festgelegt. Im Gegensatz zur Eigenschaft *Filter*, deren Filterlogik auf einen einzelnen String beschränkt ist, können Sie mit dem Ereignis *OnFilterRecord* Verzweigungen und Schleifen in die Logik einbauen und so komplexe Filterbedingungen auf mehreren Ebenen definieren.

Die Definition eines Filters mit Hilfe der Eigenschaft *Filter* bietet den Vorteil, daß die Anwendung den Filter dynamisch (z.B. als Reaktion auf eine Benutzereingabe) erzeugen, ändern und anwenden kann. Der Nachteil dieser Vorgehensweise liegt darin, daß die Filterbedingungen innerhalb eines einzelnen Text-Strings ausgedrückt werden müssen und die Verwendung von Verzweigungs- und Schleifenkonstrukten nicht zulässig ist. Außerdem ist kein Vergleich der Filterwerte mit Werten möglich, die sich noch nicht in der Datenmenge befinden.

Die Filterdefinition mit dem Ereignis *OnFilterRecord* bietet folgende Vorteile: Der Filter kann sehr komplex sein und ist dabei variabel. Er kann mehrere Quelltextzeilen umfassen, in denen auch Verzweigungs- und Schleifenkonstrukte enthalten sein dürfen. Außerdem können Werte der Datenmenge mit externen Werten verglichen werden (z.B. mit dem Text in einem Eingabefeld). Der Hauptnachteil bei der Verwendung des Ereignisses *OnFilterRecord* liegt in der Tatsache, daß der Filter zur Entwurfszeit festgelegt wird und daher nicht als Reaktion auf eine Benutzereingabe geändert werden kann. Dafür können aber mehrere Filterbehandlungsroutinen erstellt werden, die sich dann in Abhängigkeit vom Status der Anwendung aufrufen lassen.

Die folgenden Abschnitte gehen detailliert auf die Generierung von Filtern mit der Eigenschaft *Filter* und der Ereignisbehandlungsroutine für *OnFilterRecord* ein.

Die Eigenschaft Filter festlegen

Zum Festlegen eines Filters mit Hilfe der Eigenschaft *Filter* weisen Sie der Eigenschaft einen String zu, der die Filterbedingungen enthält. Dieser String beinhaltet die sogenannte Testbedingung des Filters. Die folgende Anweisung erzeugt einen Filter,

der das Feld *State* einer Datenmenge überprüft und feststellt, ob es den Wert »CA« (für Kalifornien) enthält:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

Als Wert für die Eigenschaft *Filter* kann auch Text verwendet werden, der in ein Steuerelement eingegeben wird. Zum Beispiel weist die folgende Anweisung der Eigenschaft *Filter* den Text eines Eingabefeldes zu:

```
Dataset1.Filter := Edit1.Text;
```

Natürlich können diese beiden Möglichkeiten auch kombiniert werden:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

Nachdem Sie einen Wert für *Filter* festgelegt haben, wenden Sie den Filter auf die Datenmenge an. Dazu setzen Sie die Eigenschaft *Filtered* auf *True*.

Unter Verwendung der folgenden logischen Operatoren und Vergleichsoperatoren können Feldwerte auch mit Literalen und Konstanten verglichen werden:

Tabelle 18.4 Logische Operatoren und Vergleichsoperatoren für Filterbedingungen

Operator	Bedeutung
<	Kleiner als
>	Größer als
>=	Größer oder gleich
<=	Kleiner oder gleich
=	Gleich
<>	Ungleich
AND	Beide Anweisungen müssen *True* sein
NOT	Die folgende Anweisung darf nicht *True* sein
OR	Mindestens eine der beiden Anweisungen muß *True* sein

Durch eine Kombination dieser Operatoren lassen sich relativ komplexe Filterbedingungen erstellen. Die folgende Anweisung stellt beispielsweise sicher, daß zwei Testbedingungen erfüllt sind, bevor ein Datensatz angezeigt wird:

```
(Custno > 1400) AND (Custno < 1500);
```

Hinweis Wenn die Filterung aktiviert ist und der Benutzer einen Datensatz bearbeitet, kann dies dazu führen, daß der Datensatz die Filterbedingungen nicht mehr erfüllt. Der Datensatz »verschwindet« dann möglicherweise, wenn er das nächste Mal aus der Datenmenge abgerufen wird. In diesem Fall wird der nächste Datensatz, der die Filterbedingungen erfüllt, zum aktuellen Datensatz.

Eine Behandlungsroutine für das Ereignis OnFilterRecord schreiben

Ein Filter für eine Datenmenge ist eine Routine zur Behandlung von *OnFilterRecord*-Ereignissen. Das Ereignis *OnFilterRecord* wird von der Datenmenge für jeden abgerufenen Datensatz generiert. Den Kern jeder Filterbehandlungsroutine bildet ein Test,

über den ermittelt wird, ob ein Datensatz in die Menge der für die Anwendung sichtbaren Datensätze aufgenommen wird.

Die Filterbehandlungsroutine muß anzeigen, ob ein Datensatz die Filterbedingung erfüllt oder nicht. Dies erfolgt durch ein entsprechendes Setzen des Parameters *Accept*. Wenn *Accept* den Wert *True* hat, wird der Datensatz berücksichtigt, andernfalls nicht. Der folgende Filter zeigt nur die Datensätze an, die im Feld *State* den Wert »CA« haben:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
   Accept := DataSet['State'] = 'CA';
end;
```

Wenn die Filterung aktiviert ist, wird für jeden abgerufenen Datensatz das Ereignis *OnFilterRecord* ausgelöst. Die Ereignisbehandlungsroutine prüft jeden Datensatz. Es werden nur Datensätze angezeigt, welche die Bedingungen des Filters erfüllen. Da das Ereignis *OnFilterRecord* für jeden Datensatz der Datenmenge generiert wird, sollte der Quelltext der Ereignisbehandlungsroutine so kompakt wie möglich gehalten werden, damit die Ausführungsgeschwindigkeit der Anwendung nicht beeinträchtigt wird.

Filterbehandlungsroutinen zur Laufzeit wechseln

Sie können beliebig viele Ereignisbehandlungsroutinen für Filter erstellen und diese zur Laufzeit wechseln. Dazu weisen Sie dem Ereignis *OnFilterRecord* einfach die neue Ereignisbehandlungsroutine zu.

Der folgende Quelltext weist dem Ereignis *OnFilterRecord* beispielsweise eine Behandlungsroutine namens *NewYorkFilter* zu:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

Filteroptionen festlegen

Mit Hilfe der Eigenschaft *FilterOptions* können Sie bestimmen, ob ein Filter für String-Felder Datensätze akzeptiert, die nur einen Teil des angegebenen Strings enthalten. Außerdem legen Sie mit dieser Eigenschaft fest, ob beim Vergleich von Strings die Schreibweise (groß / klein) berücksichtigt wird. *FilterOptions* ist eine Mengeneigenschaft. Sie kann leer sein (Vorgabe) oder einen bzw. beide der folgenden Werte enthalten:

Tabelle 18.5 Werte für *FilterOptions*

Wert	Bedeutung
foCaseInsensitive	Die Schreibweise wird beim String-Vergleich ignoriert.
foPartialCompare	Die teilweise Übereinstimmung ist deaktiviert (d.h., Strings mit einem Sternchen (*) am Ende sind nicht erlaubt).

Die folgenden Anweisungen erzeugen einen Filter, der die Schreibweise beim Vergleich der Werte im Feld *State* ignoriert:

```
FilterOptions := [foCaseInsensitive];
Filter := '''State'' = ''CA''';
```

Durch Datensätze einer gefilterten Datenmenge navigieren

Für die Navigation durch die Datensätze einer gefilterten Datenmenge stehen vier Methoden zur Verfügung:

Tabelle 18.6 Methoden zur Navigation in gefilterten Datenmengen

Methode	Verwendungszweck
FindFirst	Positioniert den Cursor auf den ersten Datensatz der Datenmenge, der die aktuellen Filterkriterien erfüllt. Die Suche beginnt immer beim ersten Datensatz der ungefilterten Datenmenge.
FindLast	Positioniert den Cursor auf den letzten Datensatz der Datenmenge, der die aktuellen Filterkriterien erfüllt.
FindNext	Wechselt vom aktuellen Datensatz der gefilterten Datenmenge zum nächsten Datensatz.
FindPrior	Wechselt vom aktuellen Datensatz der gefilterten Datenmenge zum vorherigen Datensatz.

Die folgende Anweisung positioniert den Cursor auf den ersten gefilterten Datensatz einer Datenmenge:

```
DataSet1.FindFirst;
```

Wenn Sie die Eigenschaft *Filter* definiert oder eine Behandlungsroutine für das Ereignis *OnFilterRecord* geschrieben haben, positionieren diese Methoden den Cursor immer auf den angegebenen Datensatz, gleichgültig, ob für die Datenmenge die Filterung aktiviert ist oder nicht. Werden diese Methoden bei deaktivierter Filterung aufgerufen, finden folgende Aktionen statt:

1. Die Filterung wird temporär aktiviert.
2. Der Cursor wird auf einen Datensatz gesetzt, der die Bedingungen erfüllt (falls vorhanden).
3. Die Filterung wird deaktiviert.

Hinweis Wenn die Filterung deaktiviert ist und Sie die Eigenschaft *Filter* nicht setzen bzw. keine Ereignisbehandlungsroutine für *OnFilterRecord* implementieren, führen diese Methoden dieselben Schritte wie *First*, *Last*, *Next* und *Prior* aus.

Alle oben aufgeführten Methoden positionieren den Cursor auf einen übereinstimmenden Datensatz (falls einer gefunden wird), machen diesen zum aktuellen Datensatz und liefern *True* zurück. Ist kein passender Datensatz vorhanden, bleibt die Position des Cursors unverändert, und der Rückgabewert lautet *False*. Sie können beim Aufruf dieser Methoden den Status der Eigenschaft *Found* prüfen und eine Aktion nur dann veranlassen, wenn *Found True* ist. Befindet sich z.B. der Cursor bereits auf dem letzten übereinstimmenden Datensatz der Datenmenge, liefert ein Aufruf der

Methode *FindNext* den Wert *False* zurück, und der aktuelle Datensatz ändert sich nicht.

Daten bearbeiten

Mit den Datenmengenmethoden in der folgenden Tabelle können Sie Daten einfügen, aktualisieren und löschen:

Tabelle 18.7 Datenmengenmethoden zum Einfügen, Aktualisieren und Löschen von Daten

Methode	Beschreibung
Edit	Versetzt die Datenmenge in den Status *dsEdit*, wenn sie sich noch nicht in diesem Status oder im Status *dsInsert* befindet.
Append	Speichert alle anstehenden Änderungen, positioniert den Cursor auf das Ende der Datenmenge und versetzt diese in den Status *dsInsert*.
Insert	Speichert alle anstehenden Änderungen und versetzt die Datenmenge in den Status *dsInsert*.
Post	Versucht, den neuen oder geänderten Datensatz in die Datenbank einzutragen. Bei erfolgreicher Ausführung wird die Datenmenge in den Status *dsBrowse* versetzt, andernfalls bleibt der aktuelle Status erhalten.
Cancel	Bricht die aktuelle Operation ab und versetzt die Datenmenge in den Status *dsBrowse*.
Delete	Löscht den aktuellen Datensatz und versetzt die Datenmenge in den Status *dsBrowse*.

Datensätze bearbeiten

Damit eine Anwendung Datensätze bearbeiten kann, muß sich die Datenmenge im Modus *dsEdit* befinden. Wenn die Nur-Lesen-Eigenschaft *CanModify* der Datenmenge den Wert *True* hat, können Sie die Datenmenge im Quelltext mit Hilfe der Methode *Edit* in den Modus *dsEdit* versetzen. *CanModify* ist *True*, wenn die einer Datenmenge zugrundeliegende Tabelle Lese- und Schreibzugriffe gestattet.

In Formularen einer Anwendung besitzen bestimmte datensensitive Steuerelemente die Fähigkeit, eine Datenmenge automatisch in den *dsEdit*-Modus zu versetzen. Dies ist unter folgenden Voraussetzungen möglich:

- Die Eigenschaft *ReadOnly* des Steuerelements muß den Wert *False* haben (Vorgabe).
- Die Eigenschaft *AutoEdit* der Datenquelle des Steuerelements muß den Wert *True* haben.
- *CanModify* muß für die Datenmenge *True* sein.

Wichtig Für *TTable* hat *CanModify* den Wert *False*, wenn die Eigenschaft *ReadOnly True* ist. Die Bearbeitung von Datensätzen wird dadurch verhindert. Dasselbe gilt für *TQuery*-Komponenten, wenn zusätzlich die Eigenschaft *RequestLive False* ist.

Hinweis Selbst wenn eine Datenmenge sich im Status *dsEdit* befindet, kann das Bearbeiten von Datensätzen in SQL-Datenbanken unmöglich sein. Ob eine Änderung der Daten möglich ist, hängt davon ab, ob die Benutzer Ihrer Anwendung über die erforderlichen SQL-Zugriffsrechte verfügen.

Sobald sich eine Datenmenge im Modus *dsEdit* befindet, kann der Benutzer die Feldwerte des aktuellen Datensatzes ändern, der in einem datensensitiven Steuerelement eines Formulars angezeigt wird. Datensensitive Steuerelemente, für welche die Bearbeitung aktiviert ist, rufen automatisch die Methode *Post* auf, wenn ein Benutzer eine Aktion ausführt, die den aktuellen Datensatz wechselt. Dies ist z.B. der Fall, wenn eine Bewegung zu einem anderen Datensatz im Gitter erfolgt.

Wenn Ihre Formulare einen Navigator enthalten, können die Benutzer die Bearbeitung abbrechen, indem sie auf die Schaltfläche *Abbrechen* des Navigators klicken. Nach dem Abbruch der Bearbeitung befindet sich die Datenmenge wieder im Status *dsBrowse*.

Im Quelltext werden Bearbeitungen durch einen Aufruf der entsprechenden Methoden gespeichert oder abgebrochen. Zum Speichern der Änderungen rufen Sie *Post* auf. Durch den Aufruf von *Cancel* werden die Änderungen verworfen. Häufig werden *Edit* und *Post* gemeinsam eingesetzt:

```
with CustTable do
begin
   Edit;
   FieldValues['CustNo'] := 1234;
   Post;
end;
```

In diesem Beispiel versetzt die erste Anweisung die Datenmenge in den Modus *dsEdit*. Die nächste Zeile weist dem Feld *CustNo* des aktuellen Datensatzes die Zahl 1234 zu. Die letzte Zeile speichert den geänderten Datensatz in der Datenbank.

Hinweis Wenn die Eigenschaft *CachedUpdates* für die Datenmenge den Wert *True* hat, werden eingetragene Änderungen in einen temporären Puffer geschrieben. Um diese zwischengespeicherten Änderungen in die Datenbank zu übernehmen, rufen Sie die Methode *ApplyUpdates* der Datenmenge auf. Weitere Informationen über zwischengespeicherte Aktualisierungen finden Sie in Kapitel 25, »Zwischengespeicherte Aktualisierungen«.

Neue Datensätze hinzufügen

Damit eine Anwendung Datensätze hinzufügen kann, muß sich die Datenmenge im Modus *dsInsert* befinden. Wenn die Nur-Lesen-Eigenschaft *CanModify* der Datenmenge den Wert *True* hat, können Sie die Datenmenge im Quelltext mit Hilfe der Methoden *Insert* oder *Append* in den Modus *dsInsert* versetzen. *CanModify* ist *True*, wenn die einer Datenmenge zugrundeliegende Datenbank Lese- und Schreibzugriffe gestattet.

In Formularen einer Anwendung können datensensitive Gitter- und Navigator-Steuerelemente eine Datenmenge unter folgenden Voraussetzungen in den Status *dsInsert* versetzen:

- Die Eigenschaft *ReadOnly* des Steuerelements muß den Wert *False* haben (Vorgabe).
- *CanModify* muß für die Datenmenge *True* sein.

Wenn sich eine Datenmenge im Modus *dsInsert* befindet, kann ein Benutzer oder eine Anwendung Werte in die Felder des neuen Datensatzes eingeben. Außer bei Gittern und Navigatoren gibt es für den Benutzer keinen sichtbaren Unterschied zwischen *Insert* und *Append*. Bei einem Aufruf von *Insert* erscheint im Gitter über dem vorher aktuellen Datensatz eine leere Zeile. Ein Aufruf von *Append* hat einen Bildlauf zum letzten Datensatz im Gitter zur Folge. Am unteren Ende des Gitters wird eine leere Zeile eingefügt, und in allen mit der Datenmenge verbundenen Navigator-Komponenten werden die Schaltflächen *Next* und *Last* deaktiviert.

Im Einfügemodus rufen datensensitive Steuerelemente automatisch die Methode *Post* auf, wenn ein Benutzer eine Aktion ausführt, die den aktuellen Datensatz wechselt. Dies ist z.B. der Fall, wenn eine Bewegung zu einem anderen Datensatz im Gitter erfolgt. Ansonsten muß zum Eintragen der Änderungen *Post* aufgerufen werden.

Post schreibt den neuen Datensatz in die Datenbank. Wenn die Zwischenspeicherung von Aktualisierungen aktiviert ist, wird der Datensatz in einem Puffer abgelegt. Um zwischengespeicherte Ein- oder Anfügungen in die Datenbank zu übernehmen, rufen Sie die Methode *ApplyUpdates* der Datenmenge auf.

Datensätze einfügen

Die Methode *Insert* öffnet vor dem aktuellen Datensatz einen neuen, leeren Datensatz. Dieser neue Datensatz wird gleichzeitig zum aktuellen Datensatz, so daß die Eingabe von Feldwerten durch den Benutzer oder die Anwendung möglich ist.

Wenn eine Anwendung die Methode *Post* aufruft (bzw. bei aktivierter Zwischenspeicherung die Methode *ApplyUpdates*), wird ein neu eingefügter Datensatz auf eine der folgenden Arten in die Datenbank geschrieben:

- Bei indizierten Paradox- und dBASE-Tabellen wird der Datensatz an der seinem Index entsprechenden Position in die Datenmenge eingefügt.
- Wenn es sich um eine Tabelle ohne Index handelt, wird der Datensatz an der aktuellen Position in die Datenmenge eingefügt.
- Bei SQL-Datenbanken ist die physikalische Einfügeposition von der Implementierung abhängig. Wenn die Tabelle indiziert ist, wird der Index mit der neuen Datensatzinformation aktualisiert.

Datensätze anhängen

Die Methode *Append* öffnet am Ende der Datenmenge einen neuen leeren Datensatz und macht ihn zum aktuellen Datensatz. Der Benutzer oder eine Anwendung können dann Feldwerte in diesen Datensatz eingeben.

Wenn eine Anwendung die Methode *Post* aufruft (bzw. bei aktivierter Zwischenspeicherung die Methode *ApplyUpdates*), wird ein neu angefügter Datensatz auf eine der folgenden Arten in die Datenbank geschrieben:

- Bei indizierten Paradox- und dBASE-Tabellen wird der Datensatz an der seinem Index entsprechenden Position in die Datenmenge eingefügt.
- Wenn es sich um eine Tabelle ohne Index handelt, wird der Datensatz am Ende der Datenmenge angehängt.
- Bei SQL-Datenbanken ist die physikalische Einfügeposition von der Implementierung abhängig. Wenn die Tabelle indiziert ist, wird der Index mit der neuen Datensatzinformation aktualisiert.

Datensätze löschen

Damit eine Anwendung Datensätze löschen kann, muß die Datenmenge aktiv sein. Die Methode *Delete* löscht den aktuellen Datensatz einer Datenmenge und versetzt diese in den Modus *dsBrowse*. Der darauffolgende Datensatz wird dadurch zum aktuellen. Wenn für die Datenmenge die Zwischenspeicherung aktiviert ist, bleibt der gelöschte Datensatz bis zu einem Aufruf von *ApplyUpdates* in einem temporären Zwischenspeicher.

Wenn Ihre Formulare einen Navigator enthalten, können die Benutzer den aktuellen Datensatz löschen, indem sie auf die Schaltfläche *Löschen* des Navigators klicken. In einem Programm ist das Löschen des aktuellen Datensatzes nur durch einen expliziten Aufruf von *Delete* möglich.

Daten in die Datenbank eintragen

Die Methode *Post* bildet das Kernstück bei der Interaktion einer Delphi-Anwendung mit einer Datenbanktabelle. Sie schreibt die Änderungen am aktuellen Datensatz in die Datenbank. Die tatsächlichen Auswirkungen von *Post* sind aber vom Status der Datenmenge abhängig:

- Im Status *dsEdit* schreibt *Post* einen geänderten Datensatz in die Datenbank (bzw. in einen Puffer, wenn die Zwischenspeicherung aktiviert ist).
- Im Status *dsInsert* schreibt *Post* einen neuen Datensatz in die Datenbank (bzw. in einen Puffer, wenn die Zwischenspeicherung aktiviert ist).
- Im Status *dsSetKey* versetzt *Post* die Datenmenge wieder in den Status *dsBrowse*.

Das Speichern kann explizit oder implizit als Teil einer anderen Prozedur erfolgen. Wenn eine Anwendung den aktuellen Datensatz wechselt, wird *Post* implizit aufgerufen. Aufrufe von *First*, *Next*, *Prior* und *Last* führen automatisch zu einem Aufruf von *Post*, wenn sich die Tabelle im Modus *dsEdit* oder *dsInsert* befindet. Die Methoden *Append* und *Insert* speichern anstehende Änderungen ebenfalls implizit.

Hinweis Die Methode *Close* führt nicht zum impliziten Aufruf von *Post*. Verwenden Sie in diesem Fall das Ereignis *BeforeClose*, um alle anstehenden Änderungen explizit zu speichern.

Änderungen rückgängig machen

Eine Anwendung kann Änderungen am aktuellen Datensatz jederzeit rückgängig machen, wenn noch kein direkter oder indirekter Aufruf von *Post* erfolgt ist. Befindet sich beispielsweise eine Datenmenge im Modus *dsEdit* und wurden die Daten eines oder mehrerer Felder von einem Benutzer geändert, kann durch einen Aufruf der Methode *Cancel* der Datenmenge der ursprüngliche Zustand des Datensatzes wiederhergestellt werden. Der Aufruf von *Cancel* versetzt die Datenmenge wieder in den Status *dsBrowse*.

In Formularen kann den Benutzern die Möglichkeit gegeben werden, Bearbeiten-, Einfügen- und Anfügen-Operationen rückgängig zu machen. Stellen Sie dafür einen Navigator mit der Schaltfläche *Abbrechen* bereit, der mit der Datenmenge verknüpft ist, oder erzeugen Sie mit entsprechendem Quelltext eine eigene Schaltfläche *Abbrechen* im Formular.

Komplette Datensätze bearbeiten

In Formularen ermöglichen alle datensensitiven Steuerelemente mit Ausnahme von Gittern und Navigatoren den Zugriff auf einzelne Felder eines Datensatzes.

Im Quelltext können Sie dazu die Methoden in der folgenden Tabelle verwenden. Diese Methoden arbeiten mit kompletten Datensatzstrukturen (vorausgesetzt, die Struktur der zugrundeliegenden Datenbanktabellen ist stabil und ändert sich nicht). Die folgende Tabelle faßt die Methoden zusammen, die eine Bearbeitung kompletter Datensätze (und nicht nur einzelner Felder) ermöglichen.

Tabelle 18.8 Methoden zur Bearbeitung kompletter Datensätze

Methode	Beschreibung
*AppendRecord([*Array mit Werten])	Hängt einen Datensatz mit den angegebenen Spaltenwerten am Ende der Tabelle an. Entspricht der Methode *Append*. *AppendRecord* hat einen impliziten Aufruf von *Post* zur Folge.
*InsertRecord([*Array mit Werten])	Fügt die angegebenen Werte als Datensatz vor der aktuellen Cursor-Position in eine Tabelle ein. Entspricht der Methode *Insert*. *InsertRecord* hat einen impliziten Aufruf von *Post* zur Folge.
*SetFields([*Array mit Werten])	Setzt die Werte der entsprechenden Felder. Dies entspricht der Zuweisung von Werten an *TFields*. Die Anwendung muß die Werte explizit (durch einen Aufruf von *Post*) eintragen.

Diese Methoden nehmen ein Array mit *Tavern*-Werten als Argument entgegen. Jeder Wert entspricht dabei einem Feld der zugrundeliegenden Datenmenge. Solche Arrays werden mit dem Makro ARRAYOFCONST erzeugt. Diese Werte können Literale, Variablen oder nil sein. Wenn die Zahl der Werte in einem Argument die Anzahl der Spalten in der Datenmenge unterschreitet, wird für die restlichen Werte **nil** angenommen.

Bei nicht indizierten Datenmengen fügt *AppendRecord* einen Datensatz am Ende der Datenmenge an, während *InsertRecord* ihn nach der aktuellen Cursor-Position einfügt. Wenn die zugrundeliegende Tabelle indiziert ist, fügen beide Methoden den Da-

tensatz an der korrekten (dem Index entsprechenden) Position ein. In jedem Fall wird der Cursor anschließend auf die Position des Datensatzes gesetzt.

SetFields weist den Feldern der Datenmenge die Werte zu, die im Parameter-Array angegeben sind. Vor der Verwendung von *SetFields* muß die Datenmenge durch einen Aufruf von *Edit* in den Modus *dsEdit* versetzt werden. Zur Übernahme der Änderungen in die Tabelle ist ein expliziter Aufruf von *Post* erforderlich.

Sie können *SetFields* einsetzen, um nur bestimmte Felder eines vorhandenen Datensatzes zu ändern. Dazu übergeben Sie **nil** für Felder, deren Inhalt nicht geändert werden soll. Wenn Sie nicht genügend Werte für alle Felder eines Datensatzes angeben, weist *SetFields* diesen Feldern Null-Werte zu. Werte, die sich bereits in diesen Feldern befinden, werden durch Null-Werte ersetzt.

Angenommen, zu einer Datenbank gehört die Tabelle *Country* mit den Spalten *Name*, *Capital*, *Continent*, *Arena* und *Population*. Wenn mit der Tabelle *Country* eine *TTable*-Komponente verbunden ist, kann mit der folgenden Anweisung ein Datensatz in die Tabelle eingefügt werden:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

Da diese Anweisung keine Werte für *Arena* und *Population* festlegt, werden für diese Spalten Null-Werte eingefügt. Die Tabelle ist über die Spalte *Name* indiziert. Die obige Anweisung fügt deshalb den Datensatz an der für »Japan« gültigen Position ein.

Mit den folgenden Anweisungen könnte eine Anwendung den Datensatz aktualisieren:

```
with CountryTable do
begin
   if Locate('Name', 'Japan', loCaseInsensitive) then;
   begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
   end;
end;
```

Dieses Quelltextfragment weist den Feldern Werte zu und speichert sie in der Datenbank. Die drei **nil**-Zeiger fungieren als Platzhalter für die drei ersten Spalten, damit diese ihren aktuellen Inhalt behalten.

Ereignisse für Datenmengen behandeln

Datenmengen verfügen über eine Anzahl von Ereignissen, die es einer Anwendung ermöglichen, Gültigkeitsprüfungen, Summenberechnungen und weitere Operationen auszuführen:

Tabelle 18.9 Ereignisse für Datenmengen

Ereignis	Beschreibung
BeforeOpen, AfterOpen	Wird vor bzw. nach dem Öffnen einer Datenmenge ausgelöst.
BeforeClose, AfterClose	Wird vor bzw. nach dem Schließen einer Datenmenge ausgelöst.
BeforeInsert, AfterInsert	Wird vor bzw. nach der Aktivierung des Einfügemodus für eine Datenmenge ausgelöst.
BeforeEdit, AfterEdit	Wird vor bzw. nach der Aktivierung des Bearbeitungsmodus für eine Datenmenge ausgelöst.
BeforePost, AfterPost	Wird vor bzw. nach dem Eintragen von Änderungen in eine Tabelle ausgelöst.
BeforeCancel, AfterCancel	Wird vor bzw. nach dem Abbruch des vorherigen Status ausgelöst.
BeforeDelete, AfterDelete	Wird vor bzw. nach dem Löschen eines Datensatzes ausgelöst.
OnNewRecord	Wird bei der Erzeugung eines neuen Datensatzes ausgelöst und ermöglicht die Festlegung von Vorgabewerten.
OnCalcFields	Wird bei der Berechnung von berechneten Feldern ausgelöst.

Weitere Informationen über Ereignisse der Komponente *TDataSet* finden Sie in der Online VCL-Referenz.

Eine Methode abbrechen

Sie können eine Methode wie *Open* oder *Insert* abbrechen, indem Sie in einer der *Before*-Ereignisbehandlungsroutinen (*BeforeOpen*, *BeforeInsert* usw.) die Prozedur *Abort* aufrufen. Der folgende Quelltext fordert beispielsweise vom Benutzer die Bestätigung einer Löschoperation. Falls der Benutzer nicht bestätigt, wird der Aufruf von *Delete* abgebrochen:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

Das Ereignis OnCalcFields

Das Ereignis *OnCalcFields* wird verwendet, um die Werte von berechneten Feldern festzulegen. Die Eigenschaft *AutoCalcFields* bestimmt, wann *OnCalcFields* ausgelöst wird. Wenn *AutoCalcFields* den Wert *True* hat, wird *OnCalcFields* in folgenden Fällen ausgelöst:

- Beim Öffnen einer Datenmenge.

- Wenn der Fokus von einer visuellen Komponente an eine andere übergeben wird, oder wenn er von einer Spalte in einem datensensitiven Gitter zu einer anderen Spalte wechselt und der aktuelle Datensatz verändert wurde.
- Wenn ein Datensatz aus der Datenbank abgerufen wird.

OnCalcFields wird immer ausgelöst, wenn sich ein Wert in einem nicht berechneten Feld ändert. Die Einstellung von *AutoCalcFields* ist dabei nicht relevant.

Vorsicht Da *OnCalcFields* sehr oft ausgelöst wird, sollte der Quelltext für dieses Ereignis möglichst kurz gehalten werden. Wenn *AutoCalcFields* den Wert *True* hat, darf *OnCalcFields* außerdem keine Aktionen ausführen, die zu einer Änderung der Datenmenge (bzw. der verknüpften Datenmenge in einer Haupt/Detail-Beziehung) führen. Andernfalls kann es zu einer Rekursion kommen. Wenn *OnCalcFields* beispielsweise durch einen Aufruf von *Post* Änderungen einträgt und *AutoCalcFields* den Wert *True* hat, wird *OnCalcFields* (und der zugehörige *Post*-Aufruf) immer wieder ausgelöst.

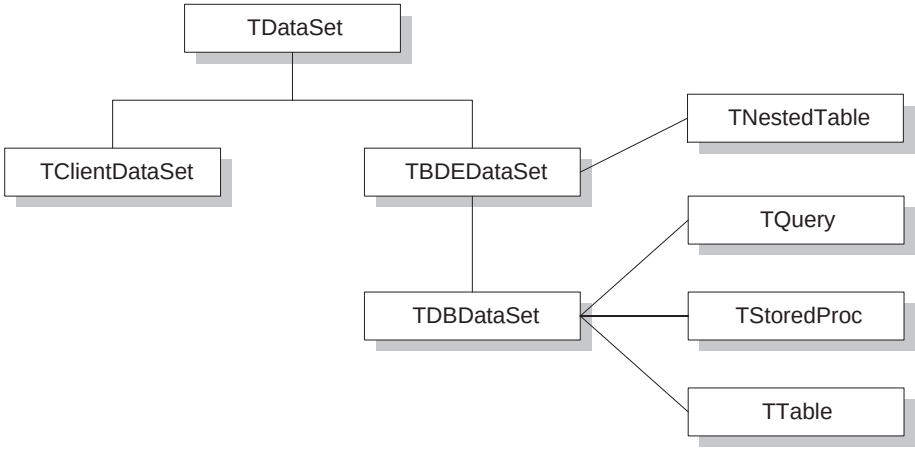
Hat *AutoCalcFields* den Wert *False*, wird *OnCalcFields* bei Änderungen an einzelnen Feldern innerhalb eines Datensatzes nicht ausgelöst.

Beim Ausführen von *OnCalcFields* befindet sich eine Datenmenge im Modus *dsCalcFields*, d.h., es können nur Werte für berechnete Felder festgelegt werden. Wenn die Ausführung von *OnCalcFields* beendet ist, befindet sich die Datenmenge wieder im Status *dsBrowse*.

BDE-Datenmengen

BDE-Datenmengen stellen die Funktionalität für Datenmengenkomponenten bereit, die über die BDE (Borland Database Engine) auf Daten zugreifen. Die Unterstützung von BDE-Datenmengen ist im Objekt *TBDEDataSet* gekapselt, das ein direkter Nachkomme von *TDataSet* ist. *TDBDataSet*, ein direkter Nachkomme von *TBDEDataSet*, führt weitere Funktionen zur Steuerung von Datenbanken und Sitzungen ein.

Abbildung 18.4 Die Hierarchie von Datenmengenkomponenten



Dieser Abschnitt befaßt sich mit den Datenmengenfunktionen von *TBDEDataSet* und *TDBDataSet*. Er geht davon aus, daß Sie mit den weiter vorne in diesem Kapitel erläuterten Merkmalen von *TDataSet* vertraut sind. Einzelheiten zu Datenmengenkomponenten, die von *TDataSet* abgeleitet sind, finden Sie am Anfang dieses Kapitels.

Hinweis *TBDEDataSet* und *TDBDataSet* werden nur bei der Entwicklung von benutzerdefinierten BDE-Datenmengen direkt in Anwendungen verwendet. Ansonsten verwenden Sie direkte Nachkommen von *TDBDataSet*: *TQuery*, *TStoredProc* und *TTable*. Informationen über *TStoredProc* finden Sie in Kapitel 22, »Stored Procedures«. Kapitel 21, »Abfragen«, enthält Einzelheiten zu *TQuery*, und in Kapitel 20, »Tabellen«, wird die Verwendung von *TTable* erläutert.

BDE-Datenmengen im Überblick

Die Komponente *TBDEDataSet* implementiert die abstrakten Methoden von *TDataSet* für die Datensatznavigation, die Indizierung und die Festlegung von Positionsmarken. Außerdem implementiert *TBDEDataSet* viele der virtuellen Methoden und Ereignisse von *TDataSet* neu, damit die Vorteile der BDE genutzt werden können. Für BDE-spezifische *TDataSet*-Implementationen gelten die gleichen Grundsätze wie bei der normalen Verwendung von *TDataSet*. Einzelheiten finden Sie am Beginn dieses Kapitels.

TBDEDataSet führt neben den BDE-spezifischen Charakteristika, die allen Datenmengen gemeinsam sind, auch neue Eigenschaften, Ereignisse und Methoden ein. Diese unterstützen die Verarbeitung von BLOBs, zwischengespeicherte Aktualisierungen und die Kommunikation mit einem Remote-Datenbankserver. *TDBDataSet* stellt eine Methode und mehrere Eigenschaften bereit, mit denen Datenbankverbindungen verwaltet und Datenmengen mit einer BDE-Sitzung verknüpft werden können. In den folgenden Abschnitten werden diese Elemente beschrieben. Gegebenenfalls finden Sie auch Verweise auf weitere in diesem Zusammenhang relevante Abschnitte im Entwicklerhandbuch.

Datenbank- und Sitzungsverbindungen verwalten

Die Komponente *TDBDataSet* führt die folgenden Eigenschaften und Funktionen für die Verwaltung von Datenbank- und Sitzungsverbindungen ein:

Tabelle 18.10 Eigenschaften und Funktionen von *TDBDataSet*

Funktion oder Eigenschaft	Beschreibung
CheckOpen (Funktion)	Legt fest, ob eine Datenbank geöffnet ist. Wenn die Verbindung aktiv ist, wird *True* zurückgegeben, andernfalls *False*.
Database (Eigenschaft)	Legt fest, mit welcher Datenbank-Komponente die Datenmenge verbunden ist.
DBHandle (Eigenschaft)	Enthält das BDE-Datenbank-Handle für die in der Eigenschaft *Database* angegebene Datenbank-Komponente. Die Eigenschaft *DBHandle* wird nur bei direkten BDE-API-Aufrufen verwendet.

Tabelle 18.10 Eigenschaften und Funktionen von *TDBDataSet* (Fortsetzung)

Funktion oder Eigenschaft	Beschreibung
DBLocale (Eigenschaft)	Enthält die BDE-Sprachtreiber-Informationen für die in der Eigenschaft *Database* angegebene Datenbank-Komponente. Die Eigenschaft *DBLocale* wird nur bei direkten BDE-API-Aufrufen verwendet.
DBSession (Eigenschaft)	Enthält das BDE-Sitzungs-Handle für die in der Eigenschaft *SessionName* angegeben Sitzungskomponente. Die Eigenschaft *DBSession* wird nur bei direkten BDE-API-Aufrufen verwendet.
DatabaseName (Eigenschaft)	Legt den BDE-Alias oder den Namen der Datenbank-Komponente für die Datenbank fest, die von der Datenmenge verwendet wird. Wenn die Datenmenge eine Paradox- oder dBASE-Tabelle ist, kann in *DatabaseName* der vollständige Pfad für das Datenbankverzeichnis angegeben werden.
SessionName (Eigenschaft)	Legt die Sitzung fest, mit der die Datenmenge verknüpft ist. Wenn für eine Datenmenge sowohl eine Datenbank- als auch eine Sitzungskomponente verwendet wird, muß der Wert von *SessionName* mit dem Wert der *SessionName*-Eigenschaft der Datenbank-Komponente übereinstimmen.

Die Eigenschaften DatabaseName und SessionName

Von allen Datenbank- und Sitzungseigenschaften, die *TDBDataSet* bereitstellt, werden *DatabaseName* und *SessionName* am häufigsten verwendet. Anwendungen, die mit Remote-Datenbanken wie Sybase, Oracle oder InterBase arbeiten, verwalten die Verbindung zu diesem Server normalerweise über eine *TDatabase*-Komponente. Die Eigenschaft *DatabaseName* jeder Datenmenge muß mit dem Namen der Datenbank-Komponente belegt sein, welche die Datenbankverbindung für die Datenmenge einrichtet. Wenn keine Datenbank-Komponenten Verwendung finden, sollte *DatabaseName* ein BDE-Alias zugewiesen werden (für Paradox und dBASE kann optional eine vollständige Pfadangabe benutzt werden).

In *SessionName* ist die BDE-Sitzung festgelegt, mit der eine Datenmenge verknüpft ist. Wenn Sie in Ihrer Anwendung keine expliziten Sitzungskomponenten verwenden, brauchen Sie dieser Eigenschaft keinen Wert zuzuweisen (dies geschieht automatisch). Werden in der Anwendung mehrere Sitzungen verwendet, legen Sie für die Eigenschaft *SessionName* einer Datenmenge den Wert fest, den die Eigenschaft *SessionName* der zugehörigen Sitzungskomponente enthält. In Anwendungen, die sowohl mehrere Sitzungskomponenten als auch eine oder mehrere Datenbank-Komponenten enthalten, muß die Eigenschaft *SessionName* einer Datenmenge mit der Eigenschaft *SessionName* der Datenbank-Komponente übereinstimmen, die mit der Datenmenge verknüpft werden soll.

Weitere Informationen über die Verwaltung von Datenbankverbindungen mit *TDatabase* finden Sie in Kapitel 17, »Datenbankverbindungen«. Einzelheiten zur Verwaltung von Sitzungen mit *TSession* und *TSessionList* enthält Kapitel 16, »Datenbanksitzungen«.

BDE-Handle-Eigenschaften

Die Eigenschaften *DBHandle*, *DBLocale* und *DBSession* werden nur benötigt, wenn die integrierte Funktionalität der Datenmengenkomponenten durch direkte API-Aufrufe der BDE umgangen wird. Es handelt sich dabei um Nur-Lesen-Eigenschaften, die einer Datenmenge automatisch zugewiesen werden, wenn diese über die BDE mit einem Datenbank-Server verknüpft wird. Die Eigenschaften können als Ressource für direkte API-Aufrufe von BDE-Funktionen verwendet werden, die eine Übergabe von Handle-Parametern erfordern. Ausführliche Informationen über die BDE-API finden Sie in der Hilfedatei BDE32.HLP.

Zwischengespeicherte Aktualisierungen

Mit Hilfe zwischengespeicherter Aktualisierungen können Daten aus einer Datenbank abgerufen, lokal gespeichert und bearbeitet und als Paket in die Datenbank eingetragen werden. Wenn die zwischengespeicherte Aktualisierung aktiviert ist, werden Änderungen an einer Datenmenge (z.B. bearbeitete oder gelöschte Datensätze) nicht sofort in die zugrundeliegende Tabelle, sondern zuerst in einen internen (lokalen) Zwischenspeicher geschrieben. Erst wenn der gesamte Bearbeitungsvorgang abgeschlossen ist, ruft die Anwendung eine Methode auf, die sämtliche Änderungen auf einmal in die Datenbank überträgt und anschließend den Zwischenspeicher leert.

Beim Zwischenspeichern von Aktualisierungen empfiehlt es sich, nicht mit einer BDE-Datenmenge, sondern mit einer Client-Datenmenge zu arbeiten. Allerdings ermöglicht *TBDEDataSet* einen alternativen Ansatz, bei dem die Handhabung zwischengespeicherter Aktualisierungen mit Hilfe integrierter Methoden erfolgt. Die folgende Tabelle enthält die relevanten Eigenschaften, Ereignisse und Methoden für zwischengespeicherte Aktualisierungen:

Tabelle 18.11 Eigenschaften, Ereignisse und Methoden für zwischengespeicherte Aktualisierungen

Eigenschaft, Ereignis oder Methode	Beschreibung
CachedUpdates (Eigenschaft)	Bestimmt, ob die Zwischenspeicherung von Aktualisierungen für die Datenmenge aktiviert ist. *True* = aktiviert, *False* = deaktiviert.
UpdateObject (Eigenschaft)	Enthält den Namen der *TUpdateSQL*-Komponente, mit der Datenmengen basierend auf Abfragen aktualisiert werden.
UpdatesPending (Eigenschaft)	Gibt an, ob der lokale Zwischenspeicher aktualisierte Datensätze enthält, die in die Datenbank eingetragen werden müssen. *True* zeigt an, daß entsprechende Datensätze vorhanden sind, *Fals*e bedeutet, daß der Zwischenspeicher leer ist.
UpdateRecordTypes (Eigenschaft)	Gibt an, welche Art von aktualisierten Datensätzen während des Eintragens der zwischengespeicherten Aktualisierungen für die Anwendung sichtbar sind.
UpdateStatus (Methode)	Zeigt an, ob ein Datensatz unverändert vorliegt, oder ob er bearbeitet, eingefügt oder gelöscht wurde.
OnUpdateError (Ereignis)	Eine benutzerdefinierte Prozedur, die Aktualisierungsfehler datensatzweise verarbeitet.

Tabelle 18.11 Eigenschaften, Ereignisse und Methoden für zwischengespeicherte Aktualisierungen (Fortsetzung)

Eigenschaft, Ereignis oder Methode	Beschreibung
OnUpdateRecord (Ereignis)	Eine benutzerdefinierte Prozedur, die Datensatzaktualisierungen datensatzweise verarbeitet.
ApplyUpdates (Methode)	Trägt die Datensätze im lokalen Zwischenspeicher in die Datenbank ein.
CancelUpdates (Methode)	Löscht alle anstehenden Aktualisierungen aus dem lokalen Zwischenspeicher, ohne sie in die Datenbank einzutragen.
CommitUpdates (Methode)	Leert den Aktualisierungszwischenspeicher, nachdem die Aktualisierungen eingetragen wurden.
FetchAll (Methode)	Kopiert alle Datensätze der Datenbank zur Bearbeitung und Aktualisierung in den lokalen Zwischenspeicher.
RevertRecord (Methode)	Macht Aktualisierungen am aktuellen Datensatz rückgängig, wenn sie noch nicht auf dem Server eingetragen wurden.

Die Verwendung von zwischengespeicherten Aktualisierungen und ihre Koordinierung mit anderen Anwendungen, die in einer Mehrbenutzerumgebung auf Daten zugreifen, wird in Kapitel 25, »Zwischengespeicherte Aktualisierungen«, ausführlich erläutert.

Wenn Sie statt dessen mit einer Client-Datenmenge arbeiten, finden Sie die entsprechenden Informationen in Kapitel 24, »Client-Datenmengen«.

BLOBs zwischenspeichern

Über die Eigenschaft *CacheBlobs* von *TBDEDataSet* können Sie festlegen, ob beim Lesen von BLOB-Datensätzen die BLOB-Felder lokal von der BDE zwischengespeichert werden sollen. Wenn *CacheBlobs* den Wert *True* hat (Vorgabe), legt die BDE eine lokale Kopie der BLOB-Felder an. Aufgrund der lokalen BLOB-Kopien muß die BDE diese Werte nicht jedesmal vom Datenbank-Server abrufen, wenn der Benutzer einen Bildlauf durchführt. Dies führt zu einer wesentlichen Verbesserung der Ausführungsgeschwindigkeit.

In Anwendungen und Umgebungen, in denen BLOB-Werte häufig aktualisiert oder ersetzt werden, ist eine aktuelle Ansicht der BLOB-Daten wichtiger als die Ausführungsgeschwindigkeit. In diesem Fall weisen Sie der Eigenschaft *CacheBlobs* den Wert *False* zu. Die Anwendung hat dann immer die aktuellste Version der BLOB-Felder zur Verfügung.

Kapitel

19

Felder

In diesem Kapitel werden die Eigenschaften, Ereignisse und Methoden des Objekts *TField* und seiner Nachkommen beschrieben. Die Nachkommen von *TField* repräsentieren einzelne Datenbankspalten in Datenmengen. In diesem Kapitel erfahren Sie außerdem, wie Sie mit Hilfe von abgeleiteten Feldkomponenten Daten in Ihren Anwendungen anzeigen und bearbeiten können.

Die Komponente *TField* wird niemals direkt in Ihren Anwendungen verwendet. Wenn Sie eine Datenmenge zum ersten Mal in Ihre Anwendung einfügen und öffnen, weist Delphi ihr automatisch für jede Spalte in der Datenbanktabelle (bzw. in den Datenbanktabellen) einen dynamischen, datentypspezifischen Nachkommen von *TField* zu. Sie können die voreingestellten Werte in den dynamischen Feldern zur Entwurfszeit überschreiben. Dazu rufen Sie den Felder-Editor zur Erstellung von persistenten Feldern auf, die dann diese Voreinstellungen ersetzen.

Die folgende Tabelle enthält die abgeleiteten Feldkomponenten, ihren Verwendungszweck und gegebenenfalls den repräsentierten Wertebereich:

Tabelle 19.1 Feldkomponenten

Komponente	Verwendungszweck
TADTField	Ein ADT-Feld (ADT = Abstract Data Type).
TAggregateField	Ein aktives Aggregat in einer Client-Datenmenge.
TArrayField	Ein Array-Feld.
TAutoIncField	Eine ganze Zahl zwischen -2.147.483.648 und 2.147.483.647. Wird in Paradox für Felder verwendet, deren Werte automatisch inkrementiert werden.
TBCDField	Reelle Zahlen mit einer festen Anzahl von Nachkommastellen. Die Genauigkeit beträgt 18 Stellen. Der Wertebereich hängt von der Anzahl der Nachkommastellen ab.
TBooleanField	Der Wert *True* oder *False*.
TBlobField	Binäre Daten. Die theoretische Höchstgrenze liegt bei 2 GByte.
TBytesField	Binäre Daten. Die theoretische Höchstgrenze liegt bei 2 GByte.

Tabelle 19.1 Feldkomponenten (Fortsetzung)

Komponente	Verwendungszweck
TCurrencyField	Reelle Zahlen zwischen $5,0 * 10^{-324}$ und $1,7 * 10^{308}$. Wird in Paradox für Felder mit einer Genauigkeit von zwei Dezimalstellen verwendet.
TDataSetField	Wert in einer verschachtelten Datenmenge.
TDateField	Datumswert.
TDateTimeField	Datums- und Zeitwert.
TFloatField	Reelle Zahlen zwischen $5,0 * 10^{-324}$ und $1,7 * 10^{308}$.
TBytesField	Binäre Daten: die maximale Anzahl Bytes beträgt 255.
TIntegerField	Ganze Zahlen zwischen -2.147.483.648 und 2.147.483.647.
TLargeintField	Ganze Zahlen zwischen -2^{63} und 2^{63}.
TMemoField	Textdaten: theoretische Höchstgrenze liegt bei 2 GByte.
TNumericField	Reelle Zahlen zwischen $3,4 * 10^{-4932}$ und $1,1 * 10^{4932}$.
TReferenceField	Ein Zeiger auf ein relationales Datenbankobjekt.
TSmallintField	Ganze Zahlen zwischen -32.768 und 32.768.
TStringField	Daten vom Typ String; die maximale Größe beträgt 8192 Byte, einschließlich des abschließenden Null-Zeichens.
TTimeField	Zeitwerte.
TVarBytesField	Binäre Daten; die maximale Anzahl Bytes beträgt 255.
TWordField	Ganze Zahlen zwischen 0 und 65.535.

In diesem Kapitel werden die Eigenschaften und Methoden beschrieben, die alle Feldkomponenten von *TField* erben. In vielen Fällen deklariert oder implementiert *TField* eine Standardfunktionalität, die von den abgeleiteten Objekten überschrieben werden kann. Wenn mehrere abgeleitete Objekte gemeinsam ein überschriebenes Funktionsmerkmal nutzen, wird dieses ebenfalls erläutert. Eine vollständige Beschreibung der einzelnen Feldkomponenten finden Sie in der VCL-Referenz.

Was sind Feldkomponenten?

Feldkomponenten sind wie alle Datenzugriffskomponenten von Delphi nichtvisuell. Sie sind auch während des Entwurfs nicht direkt sichtbar. Die Feldkomponenten werden statt dessen mit einer Datenmengenkomponente verknüpft und ermöglichen datensensitiven Steuerelementen wie *TDBEdit* und *TDBGrid* über diese Datenmenge den Zugriff auf die Datenbankspalten.

Eine Feldkomponente beschreibt die Charakteristika einer einzelnen Spalte in einem Datenbankfeld, wie beispielsweise den Datentyp und die Größe. Außerdem werden mit dieser Komponente die Anzeigemerkmale von Feldern repräsentiert, wie etwa die Ausrichtung und das Anzeige- und Bearbeitungsformat. Wenn Sie in einer Datenmenge einen Bildlauf von Datensatz zu Datensatz durchführen, können Sie mit Hilfe einer Feldkomponente den Wert für dieses Feld im aktuellen Datensatz anzeigen und

ändern. Beispielsweise verfügt die Komponente *TFloatField* über vier Eigenschaften, die direkt Einfluß auf die Darstellung ihrer Daten nehmen:

Tabelle 19.2 Die Eigenschaften von TFloatField für die Datenanzeige

Eigenschaft	Beschreibung
Alignment	Legt fest, ob die Daten linksbündig, zentriert oder rechtsbündig angezeigt werden.
DisplayWidth	Legt die Anzahl der in einem Steuerelement angezeigten Stellen fest.
DisplayFormat	Bestimmt die Formatierung der Daten für die Anzeige (z.B. die Anzahl der Dezimalstellen).
EditFormat	Legt fest, wie ein Wert während der Bearbeitung angezeigt wird.

Feldkomponenten besitzen sowohl viele gemeinsame Eigenschaften (wie *DisplayWidth* und *Alignment*) als auch für ihren Datenwert spezifische Eigenschaften (wie *Precision* für *TFloatField*). Alle diese Eigenschaften beeinflussen die Art, in der die Daten dem Benutzer in einem Formular angezeigt werden. Mit bestimmten Eigenschaften (z.B. *Precision*) kann festgelegt werden, welche Datenwerte der Benutzer beim Ändern oder Eingeben von Daten in einem Steuerelement verwenden kann.

Die Feldkomponenten für eine Datenmenge sind entweder *dynamisch* (automatisch generiert, basierend auf der zugrundeliegenden Struktur von Datenbanktabellen) oder *persistent* (basierend auf speziellen Feldnamen und Eigenschaften, die im Felder-Editor gesetzt wurden). Beide Feldarten haben Vor- und Nachteile und eignen sich für verschiedene Anwendungstypen. In den folgenden Abschnitten werden diese Felder detailliert erläutert. Dabei finden Sie auch Empfehlungen für die Auswahl des passenden Typs.

Dynamische Feldkomponenten

Standardmäßig werden Feldkomponenten dynamisch generiert. Tatsächlich sind alle Feldkomponenten für Datenmengen dynamische Felder, wenn Sie eine Datenmenge das erste Mal in ein Datenmodul integrieren, die Datenmenge mit einer Datenbank verbinden und öffnen. Eine Feldkomponente ist immer dann *dynamisch*, wenn sie automatisch erstellt wurde. Sie basiert auf den zugrundeliegenden physikalischen Charakteristika der Spalten in einer oder mehreren Datenbanktabellen, auf die eine Datenmenge zugreift. Delphi legt für jede Spalte in der zugrundeliegenden Tabelle oder Abfrage eine Feldkomponente an. Die von der Borland Database Engine (BDE) oder (in mehrschichtigen Anwendungen) von einer Provider-Komponente empfangenen Feldtypinformationen bestimmen den exakten Nachkommen von *TField*, der für jede Spalte in einer zugrundeliegenden Datenbanktabelle erstellt wird. Der Typ einer Feldkomponente wiederum legt die Eigenschaften und die Art und Weise fest, in der die mit dem Feld verbundenen Daten in den datensensitiven Steuerelementen im Formular angezeigt werden. Dynamische Felder sind temporäre Felder. Sie existieren nur, solange eine Datenmenge geöffnet ist.

Bei jedem erneuten Öffnen einer Datenmenge, die diese Felder verwendet, erstellt Delphi eine vollständig neue Menge von dynamischen Feldkomponenten für die Datenmenge. Die Feldkomponenten basieren dabei auf der aktuellen Struktur der Da-

tenbanktabellen, die der Datenmenge zugrunde liegen. Wenn die Spalten in diesen Datenbanktabellen geändert wurden, werden die automatisch generierten Feldkomponenten automatisch angepaßt, wenn Sie die betreffende Datenmenge das nächste Mal öffnen.

Verwenden Sie dynamische Felder in Anwendungen, die bei der Anzeige und Bearbeitung von Daten flexibel sein müssen. Wenn Sie z.B. ein Datenbankprogramm wie den SQL-Explorer erstellen wollen, müssen Sie dynamische Felder einsetzen, weil sich die Datenbanktabellen in bezug auf Anzahl und Typ der Spalten unterscheiden. Außerdem sind dynamische Felder auch für Anwendungen geeignet, in denen Benutzerinteraktionen mit Daten in erster Linie in Gitterkomponenten stattfinden und in denen die Datenbanktabellen häufig gewechselt werden.

Folgende Arbeitsschritte müssen ausgeführt werden, damit sich dynamische Felder in einer Anwendung einsetzen lassen:

1. Fügen Sie Datenmengen und Datenquellen in ein Datenmodul ein.
2. Verbinden Sie die Datenmengen mit Datenbanktabellen und Abfragen sowie die Datenquellen mit den Datenmengen.
3. Plazieren Sie datensensitive Steuerelemente in den Formularen der Anwendung, fügen Sie das Datenmodul in die **uses**-Klasuel aller Formular-Units ein, und verknüpfen Sie die datensensitiven Steuerelemente mit einer Datenquelle im Modul. Verknüpfen Sie außerdem mit jedem datensensitiven Steuerelement ein Feld, wenn das Steuerelement dies erfordert.
4. Öffnen Sie die Datenmengen.

Dynamische Felder sind zwar einfach einzusetzen, weisen aber auch Einschränkungen auf. Beispielsweise ist immer zusätzlicher Programmieraufwand nötig, wenn die Anzeige- und Bearbeitungsstandards geändert werden sollen, wenn eine Änderung der Anzeigereihenfolge erforderlich ist, oder wenn der Zugriff auf Felder in der Datenmenge eingeschränkt werden soll. Es ist außerdem nicht möglich, zusätzliche Felder (z.B. berechnete oder Lookup-Felder) für die Datenmenge zu erstellen und den vorgegebenen Datentyp eines dynamischen Feldes zu wechseln. Mit Hilfe des Felder-Editors können Sie persistente Feldkomponenten für die Datenmengen anlegen, die Ihnen die Steuerung und Festlegung von Feldern in Datenbankanwendungen ermöglichen.

Persistente Feldkomponenten

Datenmengenfelder sind per Voreinstellung dynamisch. Ihre Eigenschaften und ihre Verfügbarkeit werden automatisch festgelegt und können nicht geändert werden. Wenn Sie die Eigenschaften und Ereignisse von Feldern so steuern wollen, daß sich die Sichtbarkeit und die Anzeigemerkmale zur Entwurfs- oder Laufzeit setzen oder ändern lassen, müssen Sie neue Felder erstellen, die auf den in einer Datenmenge vorhandenen Feldern basieren. Auch zur Überprüfung der Dateneingabe müssen Sie persistente Felder für die Datenmenge anlegen.

Sie sollten zur Entwurfszeit mit dem Felder-Editor persistente Listen mit den Feldkomponenten anlegen, die von den Datenmengen der Anwendung eingesetzt wer-

den. Die Listen werden in der Anwendung gespeichert und bleiben unverändert. Dies gilt auch für den Fall, daß sich die Struktur der Datenbank ändert, die der Datenmenge zugrundeliegt.

Persistente Feldkomponenten bieten folgende Vorteile:

- Sie können die Felder in der Datenmenge auf eine Teilmenge der Spalten beschränken, die in der zugrundeliegenden Datenbank vorhanden sind.
- Feldkomponenten können in die Liste der persistenten Komponenten eingefügt werden.
- Feldkomponenten lassen sich aus der Liste der persistenten Komponenten entfernen, um die Anwendung am Zugriff auf bestimmte Spalten der zugrundeliegenden Datenbank zu hindern.
- Sie können neue Felder definieren (um beispielsweise vorhandene zu ersetzen), die auf den Spalten der Tabelle oder Abfrage basieren, die der Datenmenge zugrundeliegt.
- Sie können berechnete Felder definieren, deren Werte aus anderen Feldern in der Datenmenge errechnet werden.
- Sie können Lookup-Felder definieren, deren Werte aus Feldern in anderen Datenmengen errechnet werden.
- Sie können die Anzeige der Feldkomponente ändern und ihre Eigenschaften bearbeiten.

Ein persistentes Feld wird von Delphi angelegt und basiert auf den Feldnamen und Eigenschaften, die Sie im Felder-Editor angegeben haben. Sobald Sie persistente Felder mit Hilfe des Felder-Editors angelegt haben, können Sie Ereignisbehandlungsroutinen für diese Felder erstellen, die auf Änderungen in Datenwerten reagieren und Eingaben überprüfen.

Hinweis Wenn Sie für eine Datenmenge persistente Felder anlegen, stehen Ihrer Anwendung zur Entwurfs- oder Laufzeit nur diejenigen Felder zur Verfügung, die Sie auswählen. Zur Entwurfszeit können Sie mit dem Befehl *Felder hinzufügen* des Felder-Editors persistente Felder für eine Datenmenge hinzufügen oder entfernen.

Alle von einer Datenmenge verwendeten Felder sind entweder persistent oder dynamisch. Sie können in einer Anwendung nicht beide Feldtypen einsetzen. Wenn Sie für eine Datenmenge persistente Felder anlegen und diese in dynamische Felder umwandeln wollen, müssen Sie alle persistenten Felder aus der Datenmenge entfernen. Informationen über dynamische Felder finden Sie unter »Dynamische Feldkomponenten« auf Seite 19-3.

Hinweis Persistente Felder werden vorrangig zur Darstellung und Anzeige von Daten verwendet. Die Darstellung der Daten läßt sich aber auch auf andere Weise festlegen. So könnten Sie beispielsweise einer Feldkomponente Feldattribute mit Hilfe des Data-Dictionary zuweisen. Außerdem ist es möglich, die Darstellung der Spalten in datensensitiven Gitterkomponenten festzulegen. Weitere Informationen über das Data-Dictionary finden Sie unter »Attributsätze für Feldkomponenten erstellen« auf Seite 19-16. Die Konfiguration der Spaltendarstellung in Gittern wird unter »Angepaßte Gitter erstellen« auf Seite 26-21 erläutert.

Persistente Felder erstellen

Sie können persistente Feldkomponenten mit dem Felder-Editor erstellen, damit ein effizienter, übersichtlicher und typsicherer Zugriff auf die zugrundeliegenden Daten möglich ist. Persistente Feldkomponenten stellen sicher, daß bei jeder Anwendungsausführung dieselben Spalten in identischer Reihenfolge angezeigt und verwendet werden, auch wenn sich die physikalische Struktur der Datenbank geändert hat. Sie können immer davon ausgehen, daß die datensensitiven Komponenten und der Quelltext für bestimmte Felder wie erwartet arbeiten. Durch das Löschen oder Ändern einer Spalte wird eine Exception generiert, damit die Anwendung nicht mit einer fehlenden Spalte oder mit falschen Daten arbeitet.

Folgendermaßen erstellen Sie persistente Felder für eine Datenmenge:

1. Integrieren Sie eine Datenmenge in ein Datenmodul.
2. Setzen Sie die Eigenschaft *DatabaseName* für die Datenmenge.
3. Setzen Sie die Eigenschaft *TableName* (für eine *TTable*-Komponente) oder die Eigenschaft *SQL* (für eine *TQuery*-Komponente).
4. Doppelklicken Sie auf die Datenmengenkomponente im Datenmodul, um den Felder-Editor zu öffnen. Der Editor enthält eine Titelleiste, Navigationsschaltflächen und ein Listenfeld.

   Die Titelleiste zeigt den Namen des Datenmoduls bzw. Formulars an, in dem sich die Datenmenge befindet, sowie den Namen der Datenmenge selbst. Wenn Sie beispielsweise die Datenmenge *Customers* im Datenmodul *CustomerData* öffnen, wird in der Titelleiste »CustomerData.Customers« angezeigt.

   Unterhalb der Titelleiste finden Sie mehrere Navigationsschaltflächen, mit denen Sie zur Entwurfszeit einen datensatzweisen Bildlauf in der aktiven Datenmenge ausführen und zum ersten oder letzten Datensatz gelangen können. Ist die Datenmenge nicht aktiv oder leer, sind die Schalter deaktiviert.

   Im Listenfeld werden die Namen der persistenten Feldkomponenten der Datenmenge angezeigt. Wird der Felder-Editor erstmalig für eine neue Datenmenge aufgerufen, enthält die Liste noch keine Einträge, da die Feldkomponenten der Datenmenge dynamisch und nicht persistent sind. Bei einer Datenmenge, für die bereits persistente Feldkomponenten definiert wurden, werden die Namen der Komponenten in der Liste angezeigt.
5. Wählen Sie *Felder hinzufügen* im lokalen Menü des Felder-Editors.
6. Markieren Sie die Felder, die im Dialogfeld *Felder hinzufügen* als persistent definiert werden sollen. Per Voreinstellung sind alle Felder markiert, wenn das Dialogfeld geöffnet wird. Die von Ihnen markierten Felder werden zu persistenten Feldern.

Das Dialogfeld *Felder hinzufügen* wird geschlossen, und die von Ihnen ausgewählten Felder werden im Listenfeld des Felder-Editors angezeigt. Diese Felder sind persistent. Beachten Sie, daß bei aktiver Datenmenge die beiden Navigationsschaltflächen oberhalb der Liste aktiviert sind.

Von nun an erstellt Delphi beim nächsten Öffnen der Datenmenge keine dynamischen Feldkomponenten für die Spalten der zugrundeliegenden Datenbank. Statt dessen werden persistente Komponenten für die von Ihnen angegebenen Felder angelegt.

Jedesmal, wenn Sie die Datenmenge öffnen, überprüft Delphi, ob alle nicht berechneten persistenten Felder vorhanden sind oder aus den Daten in der Datenbank erstellt werden können. Wenn dies nicht der Fall ist, löst Delphi eine Exception aus, die meldet, daß das Feld nicht gültig ist. Die Datenmenge wird dann nicht geöffnet.

Persistente Felder anordnen

Die Standardreihenfolge der Felder in den datensensitiven Gitterkomponenten entspricht der Anordnung der persistenten Feldkomponenten im Felder-Editor. Sie können die Reihenfolge beliebig ändern, indem Sie die Felder mit der Maus ziehen und an der gewünschten Position ablegen.

Folgendermaßen ändern Sie die Feldreihenfolge:

1. Wählen Sie die Felder aus. Sie können mehrere Felder gleichzeitig auswählen und anordnen.
2. Ziehen Sie die Felder an die neue Position.

Wenn die ausgewählten Felder nicht unmittelbar aufeinander folgen, werden sie beim Ziehen an ihre neue Position als fortlaufender Block eingefügt. Innerhalb des Blocks bleibt die relative Position der Felder erhalten.

Sie können ein Feld auch auswählen und es mit Hilfe der Tastenkombination *Strg+Auf* oder *Strg+Ab* an die gewünschte Position verschieben.

Neue persistente Felder erstellen

Sie können nicht nur festlegen, welche vorhandenen Datenmengenfelder in persistente Felder umgewandelt werden sollen. Es ist auch möglich, neue persistente Felder zu erstellen, um diese den vorhandenen Feldern hinzuzufügen oder existierende Felder zu ersetzen. Die folgende Tabelle enthält die Typen für die Felder, die Sie zusätzlich erstellen können:

Tabelle 19.3 Spezielle persistente Feldtypen

Feldtyp	Beschreibung
Daten	Ersetzt vorhandene Felder (um beispielsweise den Datentyp eines Feldes zu ändern) und basiert auf den Spalten in der zugrundeliegenden Datenmenge (Tabelle oder Abfrage).
Berechnet	Zeigt Werte an, die zur Laufzeit von der Ereignisbehandlungsroutine für *OnCalcFields* der Datenmenge errechnet werden.
InternalCalc	Zeigt Werte an, die zur Laufzeit von einer Client-Datenmenge berechnet und mit ihren Daten gespeichert werden.

Tabelle 19.3 Spezielle persistente Feldtypen (Fortsetzung)

Feldtyp	Beschreibung
Lookup	Ruft zur Laufzeit anhand bestimmter Kriterien Werte aus der angegebenen Datenmenge ab.
Aggregate	Zeigt einen Zusammenfassungswert für die Daten einer Datensatzmenge an.

Diese persistenten Feldtypen dienen nur der Anzeige von Daten. Die zur Laufzeit in ihnen angezeigten Daten werden nicht gespeichert, da sie bereits an einer anderen Stelle in der Datenbank existieren oder temporärer Natur sind. Die der Datenmenge zugrundeliegende physikalische Tabellen- und Datenstruktur wird nicht geändert.

Zur Erstellung einer neuen Feldkomponente rufen Sie das lokale Menü des Felder-Editors auf und wählen *Neues Feld*. Das Dialogfeld *Neues Feld* wird geöffnet.

Das Dialogfeld enthält drei Gruppenfelder: *Feldeigenschaften*, *Feldtyp* und *Lookup-Definition*.

Geben Sie den Typ der neuen Komponente in der Optionsfeldgruppe *Feldtyp* an. Der Standardtyp ist *Daten*. Wenn Sie hier den Typ *Lookup* wählen, werden die Eingabefelder *Datenmenge* und *Schlüsselfelder* im Gruppenfeld *Lookup-Definition* aktiviert. Sie können auch berechnete Felder anlegen. Wenn Sie mit einer *TClientDataSet*-Komponente arbeiten, ist außerdem die Erstellung von *InternalCalc*-Feldern möglich.

Im Gruppenfeld *Feldeigenschaften* werden allgemeine Informationen über die Feldkomponente übergeben. Geben Sie den Feldnamen der Komponente in das Eingabefeld *Name* ein. Dieser Name entspricht der Eigenschaft *FieldName* der Feldkomponente. Anhand dieses Namens trägt Delphi einen Komponentennamen in das Eingabefeld *Komponente* ein. Der Name in diesem Eingabefeld entspricht der Eigenschaft *Name* der Feldkomponente und dient nur zur Information. (*Name* enthält den Bezeichner, mit dem Sie die Komponente im Quelltext referenzieren.) Delphi verwirft jede direkte Eingabe in dieses Feld.

Geben Sie den Datentyp der Feldkomponente in das Kombinationsfeld *Typ* ein. Für jede neue Feldkomponente muß ein Datentyp festgelegt werden. Sollen in einem Feld beispielsweise Gleitkommawerte im Währungsformat angezeigt werden, wählen Sie *Currency* aus der Dropdown-Liste. Im Eingabefeld *Größe* können Sie die maximale Zeichenanzahl für String-Felder bzw. die Größe für Felder des Typs *Bytes* oder *VarBytes* angeben. Bei allen anderen Datentypen ist die Größe ohne Bedeutung.

Das Gruppenfeld *Lookup-Definition* wird nur für die Erstellung von Lookup-Feldern benötigt. Informationen dazu finden Sie im Abschnitt »Lookup-Felder definieren« auf Seite 19-11.

Datenfelder definieren

Ein Datenfeld wird verwendet, um ein vorhandenes Feld in einer Datenmenge zu ersetzen. Soll beispielsweise aus programmtechnischen Gründen ein *TSmallIntField* durch ein *TIntegerField* ersetzt werden, muß ein neues Feld mit dem gewünschten Typ definiert werden. Der Datentyp eines Feldes kann nicht direkt geändert werden. Daher müssen Sie ein neues Feld erstellen, um es zu ersetzen.

Wichtig Obwohl Sie ein neues Feld definieren, um ein vorhandenes zu ersetzen, muß das neu erstellte Feld seine Datenwerte aus einer existierenden Spalte der zugrundeliegenden Tabelle ableiten.

Um ein Ersatz-Datenfeld für ein Feld der Tabelle zu erzeugen, die der Datenmenge zugrundeliegt, führen Sie die folgenden Schritte aus:

1. Entfernen Sie das Feld aus der Liste persistenter Felder, die der Datenmenge zugeordnet sind, und wählen Sie den Befehl *Neues Feld* im lokalen Menü.
2. Im Dialogfeld *Neues Feld* tragen Sie den Namen eines Feldes, das in der Datenbanktabelle vorhanden ist, in das Eingabefeld *Name* ein. Den Namen des neuen Feldes dürfen Sie nicht angeben. Sie bestimmen hier lediglich den Namen des Feldes, aus dem das neue Feld seine Daten ableitet.
3. Wählen Sie aus dem Kombinationsfeld *Typ* einen neuen Datentyp für das Feld. Es ist wichtig, daß sich die Auswahl vom Datentyp des Feldes unterscheidet, das Sie ersetzen. Beispielsweise ist es nicht möglich, ein String-Feld einer bestimmten Größe durch ein String-Feld einer anderen Größe zu ersetzen. Außerdem gilt, daß der Datentyp zwar unterschiedlich, aber immer noch kompatibel zum eigentlichen Datentyp des Feldes in der zugrundeliegenden Tabelle sein muß.
4. Geben Sie im Eingabefeld *Größe* bei Bedarf die Feldgröße an. Dieses Feld ist nur für Felder des Typs *TStringField*, *TBytesField* und *TVarBytesField* von Bedeutung.
5. Wählen Sie in der Optionsfeldgruppe *Feldtyp* die Option *Daten* aus, falls diese nicht bereits markiert ist.
6. Klicken Sie auf *OK*, um das Dialogfeld *Neues Feld* zu schließen. Das in Schritt 1 bezeichnete Feld wird durch das neu definierte Datenfeld ersetzt, und die Komponentendeklaration im Datenmodul oder in der Typdeklaration des Formulars wird entsprechend aktualisiert.

Um die Eigenschaften oder Ereignisse einer Feldkomponente zu bearbeiten, wählen Sie deren Namen in der Liste des Felder-Editors aus und führen anschließend im Objektinspektor die gewünschten Änderungen durch. Weitere Informationen hierzu finden Sie im Abschnitt »Eigenschaften und Ereignisse persistenter Felder« auf Seite 19-14.

Berechnete Felder definieren

In einem berechneten Feld werden Werte angezeigt, die zur Laufzeit von der Ereignisbehandlungsroutine für *OnCalcFields* errechnet werden. Sie können beispielsweise ein String-Feld erstellen, in dem verkettete Werte aus anderen Feldern angezeigt werden.

Gehen Sie wie folgt vor, um im Dialogfeld *Neues Feld* ein berechnetes Feld zu erstellen:

1. Geben Sie einen Namen für das berechnete Feld in das Eingabefeld *Name* ein. Verwenden Sie nicht den Namen eines vorhandenen Feldes.
2. Wählen Sie im Kombinationsfeld *Typ* einen Datentyp für das Feld aus.

3 Geben Sie im Eingabefeld *Größe* bei Bedarf die Feldgröße an. Dieses Feld ist nur für Felder des Typs *TStringField*, *TBytesField* und *TVarBytesField* von Bedeutung.

4 Markieren Sie in der Optionsfeldgruppe *Feldtyp* die Option *Berechnet*.

5 Klicken Sie auf *OK*. Das neue berechnete Feld wird automatisch an das Ende der Liste mit den persistenten Feldern im Felder-Editor angehängt, und die Komponentendeklaration wird in die Typdeklaration des Formulars im Quelltext aufgenommen.

6 Fügen Sie in die Ereignisbehandlungsroutine für *OnCalcFields* der Datenmenge Anweisungen ein, um Werte für das Feld zu errechnen. Weitere Informationen über das Berechnen von Feldwerten im Quelltext finden Sie im Abschnitt »Berechnete Felder programmieren« auf Seite 19-10.

Hinweis Um die Eigenschaften oder Ereignisse einer Feldkomponente zu bearbeiten, wählen Sie die Komponente im Felder-Editor aus und führen anschließend im Objektinspektor die gewünschten Änderungen durch. Weitere Informationen hierzu finden Sie im Abschnitt »Eigenschaften und Ereignisse persistenter Felder« auf Seite 19-14.

Wenn Sie mit einer Client-Datenmenge oder -Abfrage arbeiten, können Sie auch ein *InternalCalc*-Feld erstellen. Ein intern berechnetes Feld wird wie ein normales berechnetes Feld angelegt. Bei einer Client-Datenmenge liegt der Hauptunterschied zwischen diesen beiden Feldarten darin, daß die für ein *InternalCalc*-Feld berechneten Werte als Teil der Datenmengendaten des Clients gespeichert und ermittelt werden. Zur Erstellung eines *InternalCalc*-Feldes markieren Sie in der Optionsfeldgruppe *Feldtyp* die Option *InternalCalc*.

Berechnete Felder programmieren

Nachdem Sie ein berechnetes Feld definiert haben, müssen Sie geeigneten Quelltext schreiben, um den Wert des Feldes zu berechnen. Andernfalls hat das Feld immer den Wert Null. Der Quelltext für das berechnete Feld wird in der Ereignisbehandlungsroutine für *OnCalcFields* der zugehörigen Datenmenge eingefügt.

So programmieren Sie den Wert für ein berechnetes Feld:

1 Wählen Sie die Datenmengenkomponente aus der Dropdown-Liste des Objektinspektors.

2 Wechseln Sie im Objektinspektor zur Registerkarte *Ereignisse*.

3 Doppelklicken Sie auf die Eigenschaft *OnCalcFields*, um die *OnCalcFields*-Routine für die Datenmengenkomponente zu erstellen bzw. zu öffnen.

4 Schreiben Sie Quelltext, um den Wert und andere Eigenschaften des berechneten Feldes festzulegen.

Nehmen wir an, Sie haben das berechnete Feld *CityStateZip* für die Tabelle *Customers* im Datenmodul *CustomerData* erstellt. Das Feld *CityStateZip* soll mit Stadt, Land und Postleitzahl eines Kunden gefüllt und einzeilig in einem datensensitiven Steuerelement angezeigt werden.

Um die Anweisungen zum Berechnen des Feldes in die Prozedur *CalcFields* einzufügen, wählen Sie die Tabellenkomponente in der Dropdown-Liste des Objektinspektors aus, wechseln zur Registerkarte *Eigenschaften* und doppelklicken auf die Eigenschaft *OnCalcFields*.

Die Prozedur *TCustomerData.CustomersCalcFields* wird im Quelltextfenster der Unit angezeigt. Fügen Sie folgende Anweisung in die Prozedur ein, um das Feld zu berechnen:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
     + ' ' + CustomersZip.Value;
```

Lookup-Felder definieren

Ein Lookup-Feld ist ein Nur-Lesen-Feld, das zur Laufzeit die auf den angegebenen Suchkriterien basierenden Werte anzeigt. In der einfachsten Form wird einem Lookup-Feld der Name der zu durchsuchenden Spalte, der gesuchte Feldwert und das Feld in der Lookup-Datenmenge übergeben, dessen Wert angezeigt werden soll.

Stellen Sie sich beispielsweise eine Anwendung zur Auftragserfassung vor, in der automatisch anhand der vom Benutzer angegebenen Postleitzahl die zugehörige Stadt und das Land in einem Lookup-Feld angezeigt werden können. Die zu durchsuchende Spalte heißt in diesem Fall *ZipTable.Zip*, der gesuchte Wert ist die in *Order.CustZip* angegebene Postleitzahl des Kunden. Die Werte der Spalten *ZipTable.City* und *ZipTable.State* werden für den Datensatz zurückgegeben, bei dem *ZipTable.Zip* mit dem aktuellen Wert des Feldes *Order.CustZip* übereinstimmt.

Gehen Sie wie folgt vor, um im Dialogfeld *Neues Feld* ein Lookup-Feld zu erstellen:

1. Geben Sie in das Eingabefeld *Name* einen Namen für das Lookup-Feld ein. Verwenden Sie nicht den Namen eines vorhandenen Feldes.
2. Wählen Sie im Kombinationsfeld *Typ* einen Datentyp für das Feld.
3. Geben Sie im Eingabefeld *Größe* bei Bedarf die Feldgröße an. Dieses Feld ist nur für Felder des Typs *TStringField*, *TBytesField* und *TVarBytesField* von Bedeutung.
4. Wählen Sie in der Optionsfeldgruppe *Feldtyp* die Option *Lookup*. Dadurch werden zusätzlich die Kombinationsfelder *Datensatz* und *Schlüsselfelder* aktiviert.
5. Wählen Sie aus der Dropdown-Liste des Kombinationsfeldes *Datensatz* die gewünschte Datenmenge. Die Lookup-Datenmenge muß sich von der Datenmenge für die Feldkomponente unterscheiden, da sonst zur Laufzeit eine Exception wegen eines zirkulären Bezugs ausgelöst wird. Nach Auswahl einer Lookup-Datenmenge stehen zusätzlich die beiden Kombinationsfelder *Lookup-Schlüssel* und *Ergebnisfeld* zur Verfügung.
6. Wählen Sie aus der Dropdown-Liste *Schlüsselfelder* ein Feld der aktuellen Datenmenge, für das nach übereinstimmenden Werten gesucht werden soll. Sollen mehrere Felder verwendet werden, geben Sie die Feldnamen direkt an. Trennen Sie die Feldnamen durch Semikolons voneinander. Wenn es sich um mehr als ein Feld handelt, müssen Sie persistente Feldkomponenten verwenden.

7 Wählen Sie aus der Dropdown-Liste *Lookup-Schlüssel* das Feld der Lookup-Datenmenge, das mit dem in Schritt 6 angegebenen Schlüsselfeld verglichen werden soll. Wenn Sie mehrere Schlüsselfelder angegeben haben, müssen Sie die entsprechende Anzahl von Lookup-Schlüsseln definieren. Zur Definition mehrerer Felder geben Sie die Feldnamen, jeweils getrennt durch ein Semikolon, direkt ein.

8 Wählen Sie aus der Dropdown-Liste *Ergebnisfeld* das Feld in der Lookup-Datenmenge, das als Wert des Lookup-Feldes zurückgegeben werden soll.

Wenn Sie Ihre Anwendung entwerfen und ausführen, werden die Werte der Lookup-Felder vor denjenigen der berechneten Felder ermittelt. Die berechneten Felder können auf Lookup-Feldern basieren, was umgekehrt aber nicht möglich ist. Mit Hilfe der Eigenschaft *LookupCache* läßt sich dieses Verhalten anpassen. *LookupCache* legt fest, ob die Werte eines Lookup-Feldes zwischengespeichert werden, wenn die Datenmenge zum ersten Mal geöffnet wird, oder ob die Werte dynamisch gesucht werden, wenn sich der aktuelle Datensatz in der Datenmenge ändert.

Wenn eine Änderung von *LookupDataSet* unwahrscheinlich und die Anzahl der unterschiedlichen Lookup-Werte niedrig ist, sollten die Werte eines Lookup-Feldes zwischengespeichert werden. Setzen Sie dazu die Eigenschaft *LookupCache* auf *True*. Das Zwischenspeichern von Lookup-Werten kann zur Erhöhung der Ausführungsgeschwindigkeit beitragen. Die Lookup-Werte für jede Menge von *LookupKeyFields*-Werten werden im voraus beim Öffnen von *DataSet* geladen. Wenn sich der aktuelle Datensatz ändert, kann das Feldobjekt seinen Wert im Zwischenspeicher ausfindig machen und muß dazu nicht auf die Eigenschaft *LookupDataSet* zugreifen. Die erhöhte Ausführungsgeschwindigkeit ist vor allem dann deutlich spürbar, wenn sich *LookupDataSet* in einem Netzwerk befindet, in dem Zugriffe länger dauern.

Tip Das Zwischenspeichern von Lookup-Werten bietet die Möglichkeit, diese Werte programmseitig zur Verfügung zu stellen, anstatt sie aus einer zweiten Datenmenge abzurufen. Erzeugen Sie zur Laufzeit ein *TLookupList*-Objekt, und füllen Sie es mit Hilfe seiner Methode *Add* mit Lookup-Werten. Setzen Sie die Eigenschaft *LookupList* des Lookup-Feldes auf dieses *TLookupList*-Objekt und dessen Eigenschaft *LookupCache* auf *True*. Wenn die weiteren Lookup-Eigenschaften des Feldes nicht gesetzt werden, verwendet das Feld die übergebene Lookup-Liste, ohne sie mit den Werten aus einer Lookup-Datenmenge zu überschreiben.

Wenn alle Datensätze in *DataSet* unterschiedliche Werte für *KeyFields* haben, kann der Geschwindigkeitsvorteil durch das aufwendige Suchen der Werte im Zwischenspeicher wieder zunichte gemacht werden. Der Aufwand bei der Suche im Zwischenspeicher wird um so höher, je größer die Anzahl der unterschiedlichen Werte in *KeyFields* ist.

Wenn *LookupDataSet* nicht persistent ist, kann das Zwischenspeichern von Lookup-Werten zu ungenauen Ergebnissen führen. Rufen Sie *RefreshLookupLis* auf, um die Werte im Lookup-Zwischenspeicher zu aktualisieren. *RefreshLookupList* regeneriert die Eigenschaft *LookupList*, die für jeden Satz mit *LookupKeyFields*-Werten den Wert von *LookupResultField* enthält.

Wenn *LookupCache* zur Laufzeit gesetzt wird, initialisieren Sie den Zwischenspeicher durch einen Aufruf von *RefreshLookupList*.

Aggregatfelder definieren

In einem Aggregatfeld werden die Werte eines Aggregats einer Client-Datenmenge angezeigt. Unter einem Aggregat versteht man eine Berechnung, in der die Daten einer Datensatzgruppe zusammengefaßt werden.

Gehen Sie folgendermaßen vor, um im Dialogfeld *Neues Feld* ein Aggregatfeld zu erstellen:

1. Geben Sie in das Eingabefeld *Name* einen Namen für das Aggregatfeld ein. Verwenden Sie nicht den Namen eines vorhandenen Feldes.
2. Wählen Sie im Kombinationsfeld *Typ* den Datentyp für das Feld.
3. Wählen Sie in der Optionsfeldgruppe *Feldtyp* die Option *Aggregat*.
4. Wählen Sie *OK*. Das neue Aggregatfeld wird automatisch aktualisiert und enthält nun die Aggregatspezifikation. Gleichzeitig wird im Quelltext die Deklaration der Komponente zur **type**-Deklaration des Formulars hinzugefügt.
5. Legen Sie in der Eigenschaft *ExprText* des neuen Aggregatfeldes den Ausdruck für das Aggregat fest. Ausführliche Informationen zur Definition von Aggregaten finden Sie unter »Aggregate angeben« auf Seite 24-10.

Nach der Erstellung einer persistenten *TAggregateField*-Komponente kann diese mit einem *TDBText*-Steuerelement verknüpft werden. In diesem Steuerelement wird dann der berechnete Wert des Aggregats für den aktuellen Datensatz der zugrundeliegenden Client-Datenmenge angezeigt.

Persistente Feldkomponenten löschen

Sie können persistente Feldkomponenten löschen, um auf eine Teilmenge der verfügbaren Spalten in einer Tabelle zuzugreifen und bestimmte Tabellenspalten durch eigene persistente Felder zu ersetzen. Folgendermaßen entfernen Sie eine oder mehrere persistente Feldkomponenten aus einer Datenmenge:

1. Wählen Sie das oder die zu löschenden Felder im Listenfeld des Felder-Editors aus.
2. Drücken Sie die Taste *Entf*.

Hinweis Sie können die in der Liste ausgewählten Felder auch entfernen, indem Sie das lokale Menü *Öffnen* und *Löschen* wählen.

Die Felder sind anschließend nicht mehr für die Datenmenge verfügbar und können von den datensensitiven Steuerelementen nicht mehr angezeigt werden. Eine versehentlich gelöschte persistente Feldkomponente läßt sich zwar jederzeit erneut erstellen, jedoch sind dann die Werte der zuvor geänderten Eigenschaften oder Ereignisse nicht mehr vorhanden. Weitere Informationen finden Sie unter »Persistente Felder erstellen« auf Seite 19-6.

Hinweis Wenn Sie alle persistenten Feldkomponenten einer Datenmenge löschen, verwendet die Datenmenge für alle Spalten der zugrundeliegenden Datenbanktabelle wieder dynamische Feldkomponenten.

Eigenschaften und Ereignisse persistenter Felder

Während des Entwurfs können die Eigenschaften und Ereignisse der persistenten Feldkomponenten bearbeitet werden. Eigenschaften steuern die Art und Weise, in der ein Feld in einer datensensitiven Komponente angezeigt wird (ob es z.B. in einem *TDBGrid*-Objekt angezeigt wird oder ob sein Wert geändert werden kann). Mit Ereignissen legen Sie fest, was geschieht, sobald die Daten in einem Feld abgerufen, geändert, geschrieben oder überprüft werden.

Um die Eigenschaften einer Feldkomponente zu setzen oder eine angepaßte Ereignisbehandlungsroutine dafür zu schreiben, wählen Sie die betreffende Komponente im Felder-Editor oder in der Komponentenliste des Objektinspektors aus.

Anzeige- und Bearbeitungseigenschaften zur Entwurfszeit festlegen

In der Registerkarte *Ereignisse* des Objektinspektors können die Anzeigeeigenschaften der ausgewählten Feldkomponente bearbeitet werden. Die folgende Tabelle enthält die Anzeigeeigenschaften, die bearbeitet werden können.

Tabelle 19.4 Feldkomponenteneigenschaften

Eigenschaft	Beschreibung
Alignment	Richtet den Inhalt des Feldes in einer datensensitiven Komponente linksbündig, rechtsbündig oder zentriert aus.
ConstraintErrorMessage	Legt den Anzeigetext fest, wenn Bearbeitungen mit einer Beschränkung in Konflikt geraten.
CustomConstraint	Legt eine lokale Beschränkung fest, die während einer Datenbearbeitung gilt.
Currency	Nur für numerische Felder. *True:* Währungsbeträge werden angezeigt. *False* (Voreinstellung): Es werden keine Währungsbeträge angezeigt.
DisplayFormat	Legt das Format fest, in dem die Daten in einer datensensitiven Komponente angezeigt werden.
DisplayLabel	Gibt den Spaltennamen für ein Feld in einer datensensitiven Gitterkomponente an.
DisplayWidth	Legt die Breite der Gitterspalte fest, in der das Feld angezeigt wird. Als Maßeinheit gelten Zeichen.
EditFormat	Gibt das Bearbeitungsformat der Daten in einer datensensitiven Komponente an.
EditMask	Beschränkt die Dateneingabe in einem Feld auf den angegebenen Zeichentyp und -bereich und gibt gegebenenfalls spezielle, unzulässige Zeichen für das Feld an (Gedankenstriche, Klammern usw.).
FieldKind	Legt den Typ des zu erstellenden Feldes fest.
FieldName	Gibt den tatsächlichen Namen der Tabellenspalte an, von der das Feld seinen Wert und Datentyp ableitet.
HasConstraints	Gibt an, ob einschränkende Bedingungen für ein Feld bestehen.
ImportedConstraint	Legt eine SQL-Beschränkung fest, die aus einem Daten-Dictionary oder von einem SQL-Server importiert wurde.

Tabelle 19.4 Feldkomponenteneigenschaften (Fortsetzung)

Eigenschaft	Beschreibung
Index	Bestimmt die Position des Feldes in einer Datenmenge.
LookupDataSet	Legt die Lookup-Tabelle fest, wenn *Lookup* den Wert *True* hat.
LookupKeyFields	Legt das oder die Felder in der Lookup-Datenmenge fest, mit denen der Vergleich in einer Lookup-Operation durchgeführt wird.
LookupResultField	Bestimmt das Feld in der Lookup-Datenmenge, dessen Wert in das Lookup-Feld kopiert wird.
MaxValue	Nur für numerische Felder. Gibt den maximalen Wert an, der in das Feld eingegeben werden kann.
MinValue	Nur für numerische Felder. Gibt den minimalen Wert an, der in das Feld eingegeben werden kann.
Name	Gibt den internen Namen der Feldkomponente an.
Origin	Gibt den Namen des Feldes an, wie er in der zugrundeliegenden Datenbank angezeigt wird.
Precision	Nur für numerische Felder. Gibt an, wie viele Dezimalstellen des Wertes gespeichert werden, bevor er gerundet wird.
ReadOnly	*True:* Die Werte des Feldes werden in datensensitiven Steuerelementen angezeigt, können aber nicht geändert werden. *False* (Voreinstellung): Die Feldwerte werden angezeigt und können auch bearbeitet werden.
Size	Die maximale Zeichenanzahl, die in einem String-Feld angezeigt oder eingegeben werden kann, bzw. die Größe in Byte bei Feldern des Typs *TBytesField* und *TVarBytesField.*
Tag	Ermöglicht dem Programmierer, einen Wert für anwendungsspezifische Zwecke zu speichern.
Transliterate	*True* (Voreinstellung): Legt fest, daß Übersetzungen in oder von den entsprechenden lokalen Positionen durchgeführt werden, sobald die Daten zwischen Datenmenge und Datenbank übertragen werden. *False*: Eine lokale Übersetzung findet nicht statt.
Visible	*True* (Voreinstellung): Das Feld wird in einer datensensitiven Gitterkomponente angezeigt. *False*: Das Feld wird nicht in der Gitterkomponente angezeigt. Benutzerdefinierte Komponenten können mit dieser Eigenschaft die Anzeige aktivieren oder deaktivieren.

Nicht jede Eigenschaft ist für alle Feldkomponenten verfügbar. So besitzt beispielsweise eine Feldkomponente des Typs *TStringField* nicht die Eigenschaften *Currency, MaxValue* und *DisplayFormat*, und eine Komponente des Typs *TFloatField* hat keine Eigenschaft *Size*.

Die meisten Eigenschaften sind einfach zu verwenden. Einige erfordern jedoch zusätzlichen Programmieraufwand (z.B. *Calculated*), damit sie sinnvoll eingesetzt werden können. Andere Eigenschaften wie *DisplayFormat, EditFormat* und *EditMask* beeinflussen sich gegenseitig, so daß ihre Einstellungen koordiniert werden müssen. Informationen über diese Eigenschaften finden Sie unter »Benutzereingaben steuern« auf Seite 19-18.

Die Eigenschaften von Feldkomponenten zur Laufzeit festlegen

Die Eigenschaften einer Feldkomponente können zur Laufzeit verwendet und bearbeitet werden. Die folgende Anweisung setzt beispielsweise die Eigenschaft *ReadOnly* des Feldes *CityStateZip* der *Customers* auf *True*:

```
CustomersCityStateZip.ReadOnly := True;
```

In der nächsten Anweisung wird die Eigenschaft *Index* des Feldes *CityStateZip* der *Customers* auf 3 gesetzt und so die Feldreihenfolge geändert:

```
CustomersCityStateZip.Index := 3;
```

Attributsätze für Feldkomponenten erstellen

Wenn mehrere Felder der Datenmenge identische Formatierungseigenschaften haben (z.B. *Alignment, DisplayWidth, DisplayFormat, EditFormat, MaxValue, MinValue* usw.), können die Eigenschaften für ein einzelnes Feld festgelegt und als Attributsatz im Daten-Dictionary gespeichert werden. Die gespeicherten Attribute können dann für andere Felder verwendet werden.

Folgendermaßen erstellen Sie einen Attributsatz für eine Feldkomponente in einer Datenmenge:

1. Doppelklicken Sie auf die Datenmenge, um den Felder-Editor zu öffnen.
2. Wählen Sie das Feld aus, für das die Eigenschaften festgelegt werden sollen.
3. Legen Sie die gewünschten Feldeigenschaften im Objektinspektor fest.
4. Klicken Sie mit der rechten Maustaste in der Feldliste des Editors, um das lokale Menü zu öffnen.
5. Wählen Sie *Attribute speichern*, um die Eigenschaftseinstellungen des aktuellen Feldes als Attributsatz im Daten-Dictionary zu speichern.

Als Standardname für den Attributsatz wird der Name des aktuellen Feldes verwendet. Sie können jedoch einen anderen Namen angeben, indem Sie im lokalen Menü nicht *Attribute speichern*, sondern *Attribute speichern unter* wählen.

Hinweis Attributsätze können auch direkt im SQL-Explorer erstellt werden. In diesem Fall wird der Attributsatz jedoch keinem bestimmten Feld zugeordnet. Sie können aber zwei zusätzliche Attribute angeben: einen Feldtyp (z.B. *TFloatField*, *TStringField* usw.) und ein datensensitives Steuerelement (z.B. *TDBEdit*, *TDBCheckBox* usw.), das automatisch in ein Formular eingefügt wird, sobald ein auf dem Attributsatz basierendes Feld in das Formular gezogen wird. Weitere Informationen hierzu finden Sie in der Online-Hilfe zum SQL-Explorer.

Attributsätze mit Feldkomponenten verbinden

Wenn Formatierungseigenschaften (z.B. *Alignment, DisplayWidth, DisplayFormat, EditFormat, MaxValue, MinValue* usw.) von mehreren Feldern der Datenmenge gemeinsam verwendet werden und als Attributsätze im Daten-Dictionary gespeichert sind, können Sie die Attribute den Feldern zuweisen, ohne die Einstellungen erneut manuell durchführen zu müssen. Außerdem wirken sich spätere Änderungen der Attributeinstellungen im Daten-Dictionary automatisch auf jedes mit dem Attributsatz verknüpfte Feld aus, sobald die Feldkomponenten der Datenmenge hinzugefügt werden.

Folgendermaßen können Sie einer Feldkomponente einen Attributsatz zuweisen:

1. Doppelklicken Sie auf die Datenmenge, um den Felder-Editor zu öffnen.
2. Wählen Sie das Feld aus, für das die Attribute verwendet werden sollen.
3. Öffnen Sie das lokale Menü, und wählen Sie *Verbundene Attribute*.
4. Wählen Sie im Dialogfeld *Attribute zuordnen* den gewünschten Attributsatz aus, oder geben Sie seinen Namen ein. Ist bereits ein Attributsatz mit diesem Namen im Daten-Dictionary gespeichert, wird er im Eingabefeld angezeigt.

Wichtig Wenn der Attributsatz im Daten-Dictionary zu einem späteren Zeitpunkt geändert wird, muß er jeder Feldkomponente, die ihn benutzt, neu zugewiesen werden. Sie können dazu den Felder-Editor aufrufen und die betreffenden Komponenten in der Datenmenge per Mehrfachauswahl markieren.

Attributsatz-Zuordnungen löschen

Sie können die Zuordnung eines bestimmten Attributsatzes zu einer Feldkomponente jederzeit aufheben. Gehen Sie dazu folgendermaßen vor:

1. Rufen Sie den Felder-Editor für die Datenmenge auf, die das betreffende Feld enthält.
2. Wählen Sie das Feld oder die Felder aus, deren Zuordnung gelöscht werden soll.
3. Öffnen Sie das lokale Menü des Felder-Editors, und wählen Sie *Attribute-Zuordnung lösen*.

Wichtig Das Aufheben einer Attributzuordnung für eine Feldkomponente führt nicht zu einer Änderung der Feldeigenschaften. Ein Feld behält die Eigenschaftseinstellungen bei, die gültig waren, als ihm ein Attributsatz zugewiesen wurde. Zur Änderung dieser Eigenschaften wählen Sie das betreffende Feld im Felder-Editor aus und nehmen anschließend im Objektinspektor die gewünschten Änderungen vor.

Benutzereingaben steuern

Über die Eigenschaft *EditMask* lassen sich Typ und Bereich der Daten festlegen, die in ein datensensitives Steuerelement eingegeben werden können, das mit einer Feldkomponente des Typs *TStringField*, *TDateField*, *TTimeField*, und *TDateTimeField* verknüpft ist. Sie können die vordefinierten Masken verwenden oder eigene erstellen. Am einfachsten geschieht dies mit dem Eingabemasken-Editor. Masken können jedoch auch direkt in das Feld *EditMask* im Objektinspektor eingegeben werden.

Hinweis Bei *TStringField*-Komponenten legt die Eigenschaft *EditMask* gleichzeitig das Anzeigeformat fest.

Folgendermaßen rufen Sie den Eingabemasken-Editor für eine Feldkomponente auf:

1. Wählen Sie die gewünschte Komponente im Felder-Editor oder im Objektinspektor aus.
2. Wechseln Sie zur Registerkarte *Eigenschaften* des Objektinspektors.
3. Doppelklicken Sie auf die Wertespalte des Feldes *EditMask*, oder klicken Sie auf die Ellipsen-Schaltfläche am rechten Rand der Wertespalte, um den Eingabemasken-Editor zu öffnen.

Im Eingabemasken-Editor kann ein Maskenformat erstellt und bearbeitet werden. Im Feld *Beispielmasken* können Sie eine der vordefinierten Masken auswählen. Sobald Sie eine Beispielmaske wählen, wird diese in das Eingabefeld *Eingabemaske* übernommen. Hier kann die Maske gegebenenfalls geändert werden. Im Feld *Testeingabe* können Sie die Maske testen.

Mit der Schaltfläche *Masken* können benutzerdefinierte Maskengruppen (falls vorhanden) in das Feld *Beispielmasken* geladen werden.

Standardformate für numerische, Datums- und Zeitfelder

Delphi unterstützt für Komponenten des Typs *TFloatField*, *TCurrencyField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, und *TTimeField* interne Anzeige- und Formatierungsroutinen sowie eine intelligente Standardformatierung.

Die Standardformatierung wird mit folgenden Routinen durchgeführt:

Tabelle 19.5 Formatierungsroutinen für Feldkomponenten

Routine	Feldkomponente
FormatFloat	*TFloatField, TCurrencyField*
FormatDateTime	*TDateField, TTimeField, TDateTimeField*
FormatCurr	*TCurrencyField*

Für jede Komponente sind nur die ihrem Datentyp entsprechenden Formateigenschaften verfügbar.

Die Standardformate für Datums-, Zeit-, Währungs- und numerische Werte basieren auf den Ländereinstellungen in der Systemsteuerung. So wird beispielsweise bei den

deutschen Standardeinstellungen für eine Spalte des Typs *TFloatField* bei auf *True* gesetzter Eigenschaft *Currency* die Eigenschaft *DisplayFormat* für den Wert 1234,56 auf DM 1234,56 gesetzt, während *EditFormat* 1234,56 ist.

Sie können die Eigenschaften *DisplayFormat* und *EditFormat* einer Feldkomponente während des Entwurfs oder zur Laufzeit ändern, um die Standard-Anzeigeeigenschaften des Feldes zu überschreiben. Sie können auch Behandlungsroutinen für die Ereignisse *OnGetText* und *OnSetText* schreiben, um zur Laufzeit eine spezielle Formatierung durchzuführen. Weitere Informationen über die Eigenschaften von Feldkomponenten finden Sie unter »Die Eigenschaften von Feldkomponenten zur Laufzeit festlegen« auf Seite 19-16.

Ereignisse verarbeiten

Feldkomponenten verfügen wie die meisten Komponenten über Ereignisse. Mit entsprechenden Ereignisbehandlungsroutinen können Sie steuern, wie sich Eingaben in datensensitiven Steuerelementen auf die Felder auswirken. In der folgenden Tabelle sind die verschiedenen Ereignisse für Feldkomponenten aufgeführt:

Tabelle 19.6 Feldkomponentenereignisse

Ereignis	Beschreibung
OnChange	Wird aufgerufen, sobald sich der Wert eines Feldes ändert.
OnGetText	Wird aufgerufen, sobald der Wert einer Feldkomponente für die Anzeige oder Bearbeitung abgerufen wird.
OnSetText	Wird aufgerufen, sobald der Wert einer Feldkomponente gesetzt wird.
OnValidate	Wird aufgerufen, um den Wert einer Feldkomponente zu überprüfen, wenn sich dieser geändert hat.

Die Ereignisse *OnGetText* und *OnSetText* sind hauptsächlich für Programmierer nützlich, die über die integrierten Formatierungsfunktionen hinausgehende Formatdefinitionen einsetzen wollen. Mit *OnChange* können anwendungsspezifische Aufgaben als Reaktion auf Datenänderungen (z.B. Aktivieren oder Deaktivieren von Menüs oder visuellen Steuerelementen) durchgeführt werden. Mit *OnValidate* können Sie eingegebene Daten überprüfen, bevor die Werte an einen Datenbank-Server übergeben werden.

Folgendermaßen schreiben Sie eine Ereignisbehandlungsroutine für eine Feldkomponente:

1. Wählen Sie die betreffende Komponente aus.
2. Wechseln Sie zur Registerkarte *Ereignisse* des Objektinspektors.
3. Doppelklicken Sie auf die Wertespalte, um das Quelltextfenster mit der entsprechenden Ereignisbehandlungsroutine zu öffnen.
4. Erstellen oder bearbeiten Sie den Quelltext der Routine.

Zur Laufzeit mit Feldkomponentenmethoden arbeiten

Mit den zur Laufzeit verfügbaren Methoden der Feldkomponenten können Sie Feldwerte von einem Datentyp in einen anderen konvertieren und dem ersten datensensitiven Steuerelement in einem Formular, das mit einer Feldkomponente verknüpft ist, den Fokus übergeben.

Die Steuerung des Fokus von datensensitiven Komponenten, die mit einem Feld verknüpft sind, ist wichtig, wenn eine Anwendung in einer Ereignisbehandlungsroutine der Datenmenge (z.B. *BeforePost*) datensatzbezogene Validierungen durchführt. Die Felder in einem Datensatz können auch validiert werden, wenn das zugeordnete datensensitive Steuerelement nicht den Fokus hat. Schlägt die Prüfung für ein bestimmtes Feld fehl, kann dem Steuerelement mit den falschen Daten der Fokus übergeben werden, damit der Benutzer seine Eingabe berichtigen kann.

Der Fokus für die datensensitiven Komponenten eines Feldes kann mit der Feldmethode *FocusControl* gesteuert werden. *FocusControl* übergibt den Fokus an das erste datensensitive Steuerelement im Formular, das mit einem Feld verknüpft ist. In einer Ereignisbehandlungsroutine sollte die Methode *FocusControl* eines Feldes vor dessen Validierung aufgerufen werden. In der folgenden Anweisung wird die Methode *FocusControl* für das Feld *Company* der Tabelle *Customers* aufgerufen:

```
CustomersCompany.FocusControl;
```

Die folgende Tabelle enthält einige Methoden für Feldkomponenten. Eine vollständige Liste mit allen Methoden von Feldkomponenten und ausführliche Erläuterungen dazu finden Sie unter *TField* in der VCL-Referenz.

Tabelle 19.7 Methoden von Feldkomponenten

Methode	Beschreibung
AssignValue	Setzt einen Feldwert auf einen bestimmten Wert. Dabei wird je nach Datentyp des Feldes die entsprechende Konvertierungsfunktion für Feldkomponenten verwendet.
Clear	Löscht das Feld und setzt seinen Wert auf Null.
GetData	Ruft nicht formatierte Daten aus einem Feld ab.
IsValidChar	Ermittelt, ob ein vom Benutzer in ein datensensitives Steuerelement eingegebenes Zeichen für dieses Feld gültig ist.
SetData	Weist dem Feld unformatierte Daten zu.

Feldwerte anzeigen, konvertieren und abrufen

Datensensitive Steuerelemente wie *TDBEdit* und *TDBGrid* zeigen automatisch die Werte der Feldkomponenten an. Wenn der Bearbeitungsmodus für die Datenmenge und die Steuerelemente aktiviert ist, können die datensensitiven Steuerelemente auch neue und geänderte Werte an die Datenbank senden. Normalerweise können die datensensitiven Steuerelemente mit den integrierten Eigenschaften und Methoden eine Verbindung zu Datenmengen herstellen, Werte anzeigen und Aktualisierungen durchführen, ohne daß zusätzlicher Quelltext erforderlich ist. Verwenden Sie die-

se Komponenten daher in Ihren Datenbankanwendungen so oft wie möglich. Weitere Informationen zu den datensensitiven Steuerelementen finden Sie in Kapitel 26, »Datensensitive Steuerelemente«.

Mit den Standard-Steuerelementen können auch die in der Datenbank gespeicherten Werte angezeigt und bearbeitet werden. Die Verwendung dieser Steuerelemente erfordert jedoch möglicherweise zusätzlichen Programmieraufwand.

Werte in Standard-Steuerelementen anzeigen

In einer Anwendung kann über die Eigenschaft *Value* auf den Wert einer Datenbankspalte zugegriffen werden. Die folgende Anweisung weist beispielsweise den Wert des Feldes *CustomersCompany* einem *TEdit*-Steuerelement zu:

```
Edit3.Text := CustomersCompany.Value;
```

Diese Vorgehensweise ist für Strings geeignet, kann aber bei anderen Datentypen zusätzlichen Programmieraufwand für das Konvertieren in andere Datentypen bedeuten. Die Feldkomponenten verfügen über integrierte Konvertierungsfunktionen.

Hinweis Sie können auch Varianten verwenden, um Feldwerte abzurufen und zu setzen. Eine Variante ist ein flexibler Datentyp. Weitere Informationen zu diesem Thema finden Sie im Abschnitt »Mit der Standard-Datenmengeneigenschaft auf Werte zugreifen« auf Seite 19-22.

Feldwerte konvertieren

Die Konvertierungsfunktionen wandeln einen Datentyp in einen anderen um. So konvertiert beispielsweise die Funktion *AsString* numerische und Boolesche Werte in entsprechende Strings. In der folgenden Tabelle sind die verschiedenen Konvertierungsfunktionen für Feldkomponenten aufgeführt. Sie können ihr auch entnehmen, welche Funktionen für welchen Typ verwendet werden können:

Tabelle 19.8 Konvertierungsfunktionen für Feldkomponenten

Funktion	TStringField	TIntegerField	TSmallintField	TWordField	TFloatField	TCurrencyField	TBCDField	TDateTimeField	TDateField	TTimeField	TBooleanField	TBytesField	TVarBytesField	TBlobField	TMemoField	TGraphicField
AsVariant	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AsString		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AsInteger	✓				✓	✓	✓									
AsFloat	✓	✓	✓	✓				✓	✓	✓						
AsCurrency	✓	✓	✓	✓				✓	✓	✓						
AsDateTime	✓															
AsBoolean	✓															

Beachten Sie, daß die Methode *AsVariant* für die Konvertierung zwischen allen Datentypen benutzt werden kann. Verwenden Sie im Zweifelsfall einfach *AsVariant*.

In bestimmten Fällen sind Konvertierungen nicht möglich. So kann mit *AsDateTime* beispielsweise ein String nur dann in einen Wert des Typs *Date, Time* oder *DateTime* konvertiert werden, wenn er ein erkennbares Datums-/Zeitformat enthält. Ein fehlgeschlagener Konvertierungsversuch führt zu einer Exception.

Manchmal ist die Konvertierung möglich, aber das Ergebnis nicht aussagekräftig. Was geschieht beispielsweise, wenn ein *TDateTimeField*-Wert in ein Gleitkommaformat konvertiert wird? *AsFloat* konvertiert den Datumsteil des Feldes in die Anzahl der Tage seit dem 31.12.1899 und den Zeitanteil des Feldes in einen Bruchteil von 24 Stunden. Dies kann nützlich sein, wenn Sie beispielsweise die verstrichene Zeit berechnen wollen. In der folgenden Tabelle sind die zulässigen Konvertierungen aufgeführt, die zu speziellen Ergebnissen führen:

Tabelle 19.9 Spezielle Konvertierungsergebnisse

Konvertierung	Ergebnis
String zu *Boolean*	Konvertiert die Strings *»True«, »False«, »Yes«* und *»No«* in einen Booleschen Wert. Andere Werte lösen eine Exception aus.
Float zu *Integer*	Rundet einen Gleitkommawert auf den nächsten Integer-Wert.
DateTime zu *Float*	Konvertiert einen Datumswert in die Anzahl der Tage seit dem 31.12.1899 und die Zeit in einen Bruchteil von 24 Stunden.
Boolean zu *String*	Konvertiert einen beliebigen Booleschen Wert in den String *»True«* oder *»False«*.

In anderen Fällen sind keine Konvertierungen möglich, und jeder Versuch führt zu einer Exception.

Eine Konvertierungsfunktion wird wie jede andere Komponentenmethode verwendet: Fügen Sie in einer Anweisung den Funktionsnamen an den Komponentennamen an. Die Konvertierung wird immer vor der eigentlichen Zuweisung durchgeführt. Die folgende Anweisung konvertiert beispielsweise den Wert von *CustomersCustNo* in einen String und weist diesen einem Eingabefeld zu:

```
Edit1.Text := CustomersCustNo.AsString;
```

Umgekehrt weist die folgende Anweisung den Text eines Eingabefeldes dem Feld *CustomersCustNo* als Integer zu:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

Die Ausführung einer nicht unterstützten Konvertierung zur Laufzeit führt zu einer Exception.

Mit der Standard-Datenmengeneigenschaft auf Werte zugreifen

Die beste Vorgehensweise beim Zugriff auf den Wert eines Feldes ist die Verwendung von Varianten mit der Eigenschaft *FieldValues*. So weist beispielsweise folgende Anweisung dem Feld *CustNo* der Tabelle *Customers* den Wert eines Eingabefeldes zu:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Weitere Informationen zu Varianten finden Sie in der Online-Hilfe.

Mit der Datenmengeneigenschaft Fields auf Werte zugreifen

Auf den Wert eines Feldes können Sie mit der Eigenschaft *Fields* der zugehörigen Datenmengenkomponente zugreifen. Diese Art des Zugriffs ist hilfreich, wenn Sie mehrere Spalten in einer Schleife verarbeiten wollen oder die Anwendung mit Tabellen arbeitet, die während des Entwurfs nicht zur Verfügung stehen.

Um die Eigenschaft *Fields* verwenden zu können, müssen Sie die Reihenfolge und den Datentyp der Felder in der Datenmenge kennen. Das gewünschte Feld wird mit einer Ordinalzahl angegeben. Das erste Feld in einer Datenmenge hat die Nummer 0. Die Feldwerte müssen mit der Konvertierungsroutine der Feldkomponente umgewandelt werden. Weitere Informationen über die Konvertierungsfunktionen für Feldkomponenten finden Sie unter »Feldwerte konvertieren« auf Seite 19-21.

Die folgende Anweisung weist beispielsweise einem Eingabefeld den aktuellen Wert der siebten Spalte (*Country*) der Tabelle *Customers* zu:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Umgekehrt können Sie einem Feld einen Wert zuweisen, indem Sie die Eigenschaft *Fields* der Datenmenge auf das betreffende Feld setzen:

```
begin
   Customers.Edit;
   Customers.Fields[6].AsString := Edit1.Text;
   Customers.Post;
end;
```

Mit der Datenmengenmethode FieldByName auf Werte zugreifen

Sie können auf den Wert eines Feldes auch mit der Methode *FieldByName* einer Datenmenge zugreifen. Dieses Vorgehen ist nützlich, wenn Sie den Namen des Feldes kennen, jedoch während des Entwurfs keinen Zugriff auf die zugrundeliegende Tabelle haben.

Um die Methode *FieldByName* verwenden zu können, müssen Sie die Datenmenge und den Namen des betreffenden Feldes kennen. Der Feldname wird als Argument an die Methode übergeben. Soll der Wert des Feldes gelesen oder geändert werden, konvertieren Sie das Ergebnis mit der entsprechenden Konvertierungsfunktion (z.B. *AsString* oder *AsInteger)*. Die folgende Anweisung weist beispielsweise einem Eingabefeld den Wert des Feldes *CustNo* der Tabelle *Customers* zu:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Sie können auch umgekehrt einem Feld einen Wert zuweisen:

```
begin
   Customers.Edit;
   Customers.FieldByName('CustNo').AsString := Edit2.Text;
   Customers.Post;
end;
```

Den aktuellen Wert eines Feldes überprüfen

Wenn Ihre Anwendung *TClientDataSet* verwendet oder eine Datenmenge verwaltet, die als Quelldatenmenge einer *TProvider*-Komponente auf einem Anwendungsserver fungiert, können bei der Aktualisierung von Datensätzen Probleme auftreten. In diesem Fall kann mit Hilfe der Eigenschaft *CurValue* der Feldwert im fehlerverursachenden Datensatz geprüft werden. *CurValue* spiegelt den aktuellen Wert der Feldkomponente wider (einschließlich der Änderungen, die andere Benutzer vorgenommen haben).

Verwenden Sie *CurValue* zum Prüfen eines Feldwertes, wenn beim Eintragen des Wertes in die Datenbank ein Problem aufgetreten ist. Wurde das Problem (z.B. ein Indexfehler) durch den aktuellen Feldwert verursacht, tritt das Ereignis *OnReconcileError* ein. In einer Behandlungsroutine für *OnReconcileError* ist *NewValue* der noch nicht eingetragene, fehlerverursachende Wert, *OldValue* ist der Wert, den das Feld vor der Änderung hatte, und *CurValue* ist der Wert, der dem Feld aktuell zugewiesen ist. Wenn der Wert nach dem Lesen von *OldValue* von einem anderen Benutzer geändert wurde, ist *CurValue* möglicherweise nicht mehr mit *OldValue* identisch.

Einen Standardwert für ein Feld festlegen

Über die Eigenschaft *DefaultExpression* können Sie festlegen, wie ein Standardwert für ein Feld zu Laufzeit berechnet wird. *DefaultExpression* kann jeder gültige SQL-Wertausdruck sein, der sich nicht auf Feldwerte bezieht. Enthält der Ausdruck Literale, die keine numerischen Werte sind, müssen diese in Anführungszeichen eingeschlossen werden. Der Standardwert 12.00 Uhr für ein Zeitfeld würde beispielsweise so angegeben:

```
'12:00:00'
```

Der literale Wert muß in halbe Anführungszeichen gesetzt werden.

Datenbeschränkungen

Beschränkungen von SQL-Servern können in Feldkomponenten übernommen werden. Außerdem ist es möglich, in einer Anwendung selbstdefinierte, lokale Beschränkungen zu definieren und zu nutzen. Beschränkungen sind Bedingungen, die den Bereich der zulässigen Werte für ein Feld eingrenzen. Die folgenden Abschnitte erläutern den Einsatz von Beschränkungen auf Feldkomponentenebene.

Selbstdefinierte Beschränkungen

Eine selbstdefinierte Beschränkung wird nicht vom Server importiert, sondern von Ihnen in einer lokalen Anwendung deklariert, implementiert und angewendet. Selbstdefinierte Beschränkungen ermöglichen somit eine Überprüfung der Dateneingabe des Benutzers. Es ist nicht möglich, eine selbstdefinierte Beschränkung auf Da-

ten anzuwenden, die von einer Server-Anwendung empfangen bzw. an diese gesendet werden.

Zur Definition einer selbstdefinierten Beschränkung legen Sie in der Eigenschaft *CustomConstraint* eine entsprechende Bedingung fest und weisen der Eigenschaft *ConstraintErrorMessage* eine Fehlermeldung zu, die angezeigt wird, wenn der Benutzer zur Laufzeit unzulässige Daten eingibt.

CustomConstraint ist ein SQL-String, in dem anwendungsspezifische Beschränkungen für den Feldwert festgelegt sind. Durch eine entsprechende Zuweisung an *CustomConstraint* können Sie die Werte festlegen, die der Benutzer in ein Feld eingeben darf. *CustomConstraint* kann jeden gültigen SQL-Suchausdruck enthalten, z.B.:

```
x > 0 and x < 100
```

Als Referenz auf den Feldwert kann jeder String benutzt werden, der kein reserviertes SQL-Schlüsselwort darstellt. Dieser String muß dann aber konsequent im gesamten Beschränkungsausdruck beibehalten werden.

Selbstdefinierte Beschränkungen gelten immer zusätzlich zu den Einschränkungen, die der Server für den Feldwert festlegt. Mit einem Lesezugriff auf die Eigenschaft *ImportedConstraint* können die Server-Beschränkungen ermittelt werden.

Server-Beschränkungen

Die meisten SQL-Datenbanken verwenden Beschränkungen, um die zulässigen Werte für ein Feld festzulegen. Beispielsweise läßt sich durch eine entsprechende Beschränkung die Eingabe von Null-Werten in ein Feld verhindern. Über eine Beschränkung kann aber auch festgelegt werden, daß ein Wert für eine Spalte eindeutig sein muß, oder daß nur Werte zulässig sind, die größer als 0 und kleiner als 150 sind.

Zwar könnten entsprechende Beschränkungen auch in der Client-Anwendung selbst definiert werden, es ist aber einfacher, die Server-Beschränkungen mit Hilfe der Eigenschaft *ImportedConstraint* zu importieren. *ImportedConstraint* ist eine Nur-Lesen-Eigenschaft, die mit Hilfe einer SQL-Klausel die Feldwerte auf eine bestimmte Art und Weise einschränkt. Beispiel:

```
Value > 0 and Value < 100
```

Der Wert von *ImportedConstraint* darf nur geändert werden, wenn serverspezifische oder nicht dem Standard entsprechende SQL-Anweisungen bearbeitet werden müssen, die von der Datenbank-Engine nicht interpretiert werden können und deshalb als Kommentar importiert wurden.

Verwenden Sie die Eigenschaft *CustomConstraint*, um zusätzliche Beschränkungen für den Feldwert einzurichten. Selbstdefinierte Beschränkungen gelten immer zusätzlich zu den Einschränkungen, die importiert wurden. Wenn sich die Server-Beschränkungen ändern, hat dies eine Änderung des Wertes von *ImportedConstraint* zur Folge. Die über die Eigenschaft *CustomConstraint* festgelegten Beschränkungen ändern sich dadurch nicht.

Wenn Beschränkungen aus der Eigenschaft *ImportedConstraint* entfernt werden, führt dies nicht zu einer Änderung der Feldwerte, die diese Beschränkungen verletzen. Das Entfernen von Beschränkungen hat zur Folge, daß die Einhaltung nicht mehr auf

lokaler Ebene, sondern durch den Server geprüft wird. Wenn bei einer lokalen Prüfung eine Verletzung der Beschränkungen festgestellt wird, führt dies zur Anzeige der in der Eigenschaft *ConstraintErrorMessage* enthaltenen Fehlermeldung. Diese ersetzt dann die Fehlermeldung des Servers.

Objektfelder

Die Nachkommen von Objektfeldern (*TObjectField*) unterstützen die Feldtypen ADT (Abstract Data Type), Array, DataSet und Reference. Alle diese Feldtypen enthalten oder referenzieren untergeordnete Felder oder andere Datenmengen.

ADT-Felder und Referenzfelder entsprechen Feldern, die untergeordnete Felder enthalten. Ein ADT-Feld enthält untergeordnete Felder, die ihrerseits von jedem skalaren Typ oder jedem Objekttyp sein können. Ein Array-Feld wiederum enthält ein Array von untergeordneten Feldern, die alle vom gleichen Typ sind.

Datenmengen- und Referenzfelder entsprechen Feldern, die auf andere Datenmengen zugreifen. Ein Datenmengenfeld gewährt Zugriff auf eine verschachtelte Datenmenge, und ein Referenzfeld speichert einen Zeiger (eine Referenz) auf ein anderes persistentes Objekt (ADT).

Tabelle 19.10 Typen von Objektfeldkomponenten

Komponentenname	Beschreibung
TADTField	Repräsentiert ein ADT-Feld (Abstract Data Type).
TArrayField	Repräsentiert ein Array-Feld.
TDataSetField	Repräsentiert ein Feld, das eine Referenz auf eine verschachtelte Datenmenge enthält.
TReferenceField	Repräsentiert ein REF-Feld (Zeiger auf ein ADT-Feld).

Werden mit dem Felder-Editor Felder in eine Datenmenge eingefügt, die Objektfelder enthält, werden automatisch persistente Objektfelder des korrekten Typs erstellt. Das Hinzufügen persistenter Objektfelder zu einer Datenmenge setzt automatisch die Eigenschaft *ObjectView* dieser Datenmenge auf *True*, wodurch die Felder dann entsprechend ihrer Hierarchie gespeichert werden.

In der folgenden Tabelle sind die Eigenschaften aufgeführt, über die alle Objektfeld-Nachkommen verfügen. Über diese Eigenschaften wird die Funktionalität für die Verarbeitung von untergeordneten Feldern und Datenmengen implementiert.

Tabelle 19.11 Eigenschaften von Objektfeld-Nachkommen

Eigenschaft	Beschreibung
Fields	Enthält die untergeordneten Felder des Objektfeldes.
ObjectType	Klassifiziert das Objektfeld.
FieldCount	Enthält die Anzahl der Felder, die dem Objektfeld untergeordnet sind.
FieldValues	Ermöglicht den Zugriff auf die Werte der untergeordneten Felder des Objektfeldes.

ADT- und Array-Felder anzeigen

ADT- und Array-Felder enthalten untergeordnete Felder, die mittels datensensitiver Steuerelemente angezeigt werden können. Die automatische Anzeige von ADT- und Array-Feldern kann mit datensensitiven Komponenten wie *TDBEdit* und *TDBGrid* erfolgen.

Datensensitive Steuerelemente, die über die Eigenschaft *DataField* verfügen, zeigen ADT- und Array-Felder mit ihren untergeordneten Feldern in einer Dropdown-Liste an. Wenn ein ADT- oder Array-Feld mit einem datensensitiven Steuerelement verknüpft wird, werden die untergeordneten Felder im Steuerelement in Form eines Strings angezeigt. Dieser String kann nicht bearbeitet werden. Untergeordnete Felder sind mit dem Steuerelement als normale Datenfelder verknüpft.

In einem *TDBGrid*-Steuerelement werden ADT- und Array-Felder unterschiedlich angezeigt. Maßgeblich für die Art der Anzeige ist der Wert der *ObjectView*-Eigenschaft der Datenmenge. Wenn *ObjectView* den Wert *False* hat, wird jedes untergeordnete Feld in einer eigenen Spalte angezeigt. Ist *ObjectView* dagegen *True*, kann die erweiterte Anzeige des ADT- oder Array-Feldes durch Klicken auf den Pfeil in der Titelleiste der Spalte ein- und ausgeblendet werden. Ist die Anzeige eingeblendet, wird unter der Titelleiste des ADT- bzw. Array-Feldes für jedes untergeordnete Feld eine eigene Spalte und Titelleiste angezeigt. Andernfalls ist nur eine Spalte mit einem String zu sehen, der die untergeordneten Felder des ADT- oder Array-Feldes enthält. Dieser String kann nicht geändert werden.

ADT-Felder

ADT-Felder sind benutzerdefinierte Typen, die auf dem Server erstellt wurden. Sie sind in etwa mit Strukturen vergleichbar. ADT-Felder können die meisten skalaren Feldtypen, Array-Felder, Referenzfelder und verschachtelte ADT-Felder enthalten.

Auf Werte von ADT-Feldern zugreifen

Es gibt verschiedene Möglichkeiten, auf die Daten von ADT-Feldern zuzugreifen. Unbedingt empfehlenswert ist die Erstellung und Verwendung von persistenten Feldern. Im folgenden Beispiel wird einem Eingabefeld mit dem Namen *CityEdit* der Wert eines untergeordneten Feldes zugewiesen. Die verwendete ADT-Struktur sieht folgendermaßen aus:

```
Address
   Street
   City
   State
   Zip
```

Die folgenden persistenten Felder wurden für die Tabellenkomponente *Customer* erstellt:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
```

```
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

In der folgenden Anweisung wird für den Zugriff auf das ADT-Feld ein persistentes Feld verwendet.

```
CityEdit.Text := CustomerAddrCity.AsString;
```

Damit die folgenden Beispielanweisungen compiliert werden können, ist es erforderlich, daß die Eigenschaft *ObjectView* der Datenmenge den Wert *True* hat. Persistente Felder werden hier nicht benötigt.

In der folgenden Anweisung wird die Methode *FieldByName* der Datenmenge und ein vollständig qualifizierter Name verwendet.

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

Der Zugriff auf den Wert eines untergeordneten Feldes kann auch über die Eigenschaft *FieldValues* der *TADTField*-Komponente erfolgen. Da diese Eigenschaft eine Variante entgegennimmt und zurückliefert, kann sie Felder jeden Typs verarbeiten und konvertieren. Im Parameter *Index* wird ein Wert übergeben, der den Offset des Feldes festlegt. *FieldValues* ist die Standardeigenschaft von *TObjectField* und kann daher auch weggelassen werden. Die Anweisung

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

ist deshalb mit der folgenden identisch:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

In der folgenden Quelltextzeile wird die Eigenschaft *Fields* der *TADTField*-Komponente verwendet:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

In der nächsten Anweisung wird die Eigenschaft *Fields* der *TADTField*-Komponente in einem kombinierten Aufruf der Methode *FieldByName* der Datenmenge und des *TFields*-Objekts verwendet:

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

Dieses Beispiel zeigt deutlich, daß sich der Zugriff auf die Felddaten mit Hilfe von persistenten Feldern einfacher bewerkstelligen läßt. Die anderen Methoden sollten nur eingesetzt werden, wenn sich die Struktur der Datenbanktabelle ändern kann oder zur Entwurfszeit nicht bekannt ist.

Auf die Werte von ADT-Feldern kann auch über die Eigenschaft *FieldValues* der Datenmenge zugegriffen werden:

```
Customer.Edit;
Customer['Address.City'] := CityEdit.Text;
Customer.Post;
```

Die nächste Anweisung liest einen String-Wert aus dem untergeordneten Feld *City* des ADT-Feldes *Address* in ein Eingabefeld ein:

```
CityEdit.Text := Customer['Address.City'];
```

Hinweis Für die Compilierung der obigen Quelltextzeilen ist es unerheblich, ob die Eigenschaft *ObjectView* der Datenmenge *True* oder *False* ist.

Array-Felder

Array-Felder bestehen aus mehreren Feldern desselben Typs. Die Feldtypen können skalar (z.B. Float oder String) oder nichtskalar sein (ADT). Array-Felder, die weitere Arrays enthalten, sind nicht zulässig. Die Eigenschaft *SparseArrays* einer *TDataSet*-Komponente bestimmt, ob für jedes Element im Array-Feld ein eindeutiges *TField*-Objekt erstellt wird.

Auf Werte von Array-Feldern zugreifen

Es gibt verschiedene Möglichkeiten, auf die Daten von Array-Feldern zuzugreifen. Im folgenden Beispiel wird ein Listenfeld mit allen Array-Elementen gefüllt, die nicht Null sind:

```
var
   OrderDates: TArrayField;
   I: Integer;
begin
   for I := 0 to OrderDates.Size - 1 do
   begin
    if OrderDates.Fields[I].IsNull then Break;
    OrderDateListBox.Items.Add(OrderDates[I]);
   end;
end;
```

In den nächsten Beispielen wird einem Eingabefeld mit dem Namen *TelEdit* der Wert eines untergeordneten Feldes zugewiesen. Dabei wird ein Array namens *TelNos_Array* verwendet, das aus sechs Elementen (Strings) besteht. Das Beispiel basiert auf folgenden persistenten Feldern für die Tabellenkomponente *Customer*:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

In der folgenden Anweisung wird einem Eingabefeld der Wert eines Array-Elements mit Hilfe eines persistenten Feldes zugewiesen:

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

Damit die folgenden Beispielanweisungen compiliert werden können, ist es erforderlich, daß die Eigenschaft *ObjectView* der Datenmenge den Wert *True* hat. Persistente Felder werden hier nicht benötigt.

Der Zugriff auf den Wert eines untergeordneten Feldes kann auch über die Eigenschaft *FieldValues* der Datenmenge erfolgen. Da diese Eigenschaft eine Variante entgegennimmt und zurückliefert, können mit ihr Felder jeden Typs verarbeitet und konvertiert werden. Die Anweisung

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

ist mit der folgenden identisch:

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

In der nächsten Anweisung wird die Eigenschaft *Fields* der *TArrayField*-Komponente verwendet:

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).Fields[1].AsString;
```

Datenmengenfelder

Mit Hilfe von Datenmengenfeldern kann auf Daten zugegriffen werden, die in verschachtelten Datenmengen gespeichert sind. Die Eigenschaft *NestedDataSet* referenziert die verschachtelte Datenmenge. Der Zugriff auf die Daten der verschachtelten Datenmenge erfolgt über ihre Feldobjekte.

Datenmengenfelder anzeigen

Die in einem Datenmengenfeld gespeicherten Daten lassen sich mit Hilfe eines Datengitters (*TDBGrid*) anzeigen. In einem *TDBGrid*-Steuerelement ist ein Datenmengenfeld in den einzelnen Zellen der Datenmengenspalte am Kürzel *(DataSet)* zu erkennen. Zur Laufzeit wird rechts neben dem Datenmengenfeld eine Ellipsen-Schaltfläche angezeigt. Wird diese Schaltfläche angeklickt, wird ein neues Formular eingeblendet, das in einem Gitter die Datenmenge für das Datenmengenfeld des aktuellen Datensatzes enthält. Das Formular kann aber auch programmseitig mit der Methode *ShowPopupEditor* des Datengitters eingeblendet werden. Wenn z.B. die siebte Spalte im Gitter ein Datenmengenfeld repräsentiert, zeigt die folgende Anweisung die Datenmenge für das Feld des aktuellen Datensatzes an:

```
DBGrid1.ShowPopUpEditor(DbGrid1.columns[7]);
```

Auf Daten in einer verschachtelten Datenmenge zugreifen

Normalerweise ist ein Datenmengenfeld nicht direkt mit einem datensensitiven Steuerelement verbunden. Da nun aber eine verschachtelte Datenmenge auch nichts anderes ist als eben eine Datenmenge, erfolgt der Zugriff auf die darin enthaltenen Daten über einen Nachkommen von *TDataSet*. Dieser besondere Nachkomme, *TNestedTable*, bietet genau die Funktionalität, die der Zugriff auf die in einer verschachtelten Datenmenge enthaltenen Daten erfordert. Da ja ein *TDataSetField*-Objekt mit einem Datenmengenfeld verbunden ist, können für die Felder der verschachtelten Datenmenge persistente Felder erstellt werden.

Um auf die in einem Datenmengenfeld enthaltenen Daten zuzugreifen, erstellen Sie zuerst mit Hilfe des Felder-Editors der Tabelle ein persistentes *TDataSetField*-Objekt und verknüpfen es dann mittels der Eigenschaft *DatasetField* mit einem *TNestedTable*- oder einem *TClientDataSet*-Objekt. Wenn das verschachtelte Datenmengenfeld für den aktuellen Datensatz zugewiesen ist, enthält die verschachtelte Datenmenge Datensätze mit den verschachtelten Daten; anderenfalls bleibt die verschachtelte Datenmenge leer.

Bevor Sie Datensätze in eine verschachtelte Datenmenge einfügen, sollten Sie sicherstellen, daß der korrespondierende Datensatz in die Haupttabelle eingetragen wurde. Wenn der eingefügte Datensatz nicht eingetragen wurde, wird er vor dem Eintragen der verschachtelten Datenmenge automatisch eingetragen.

Referenzfelder

Referenzfelder enthalten einen Zeiger oder eine Referenz auf ein anderes ADT-Objekt. Das ADT-Objekt ist ein Datensatz in einer anderen Objekttabelle. Referenzfelder nehmen immer auf einen einzelnen Datensatz in einer Datenmenge (Objekttabelle) Bezug. Die Daten im referenzierten Objekt werden in einer verschachtelten Datenmenge zurückgegeben. Sie können aber auch über die Eigenschaft *Fields* von *TReferenceField* auf die Daten zugreifen.

Referenzfelder anzeigen

In einem *TDBGrid*-Steuerelement erkennen Sie ein Referenzfeld in den einzelnen Zellen der Datenmengenspalte am Kürzel *(Reference)*. Zur Laufzeit wird rechts neben dem Referenzfeld eine Ellipsen-Schaltfläche angezeigt. Wenn der Benutzer auf diese Schaltfläche klickt, wird ein neues Formular eingeblendet, das in einem Gitter das Objekt für das Referenzfeld des aktuellen Datensatzes enthält.

Dieses Formular kann auch programmseitig mit der Methode *ShowPopupEditor* des Datengitters eingeblendet werden. Wenn z.B. die siebte Spalte im Gitter ein Referenzfeld repräsentiert, zeigt die folgende Anweisung das Objekt für das Feld des aktuellen Datensatzes an:

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Auf Daten in einem Referenzfeld zugreifen

Um auf die in einem Referenzfeld enthaltenen Daten zuzugreifen, erstellen Sie zuerst ein persistentes *TDataSetFieldDatasetField* mit einem *TNestedTable*- oder einem *TClientDataSet*-Objekt. Wenn die Referenz zugewiesen ist, enthält sie einen einzelnen Datensatz mit den referenzierten Daten. Wenn die Referenz null ist, ist sie leer.

Die beiden folgenden Anweisungen haben den gleichen Effekt: Sie weisen einem Eingabefeld mit dem Namen *CityEdit* Daten aus dem Referenzfeld *CustomerRefCity* zu:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

Wenn die Daten eines Referenzfeldes bearbeitet werden, führt dies zur Änderung der referenzierten (zugrundeliegenden) Daten.

Um ein Referenzfeld zuzuweisen, wählen Sie zunächst aus der Tabelle mit Hilfe einer SELECT-Anweisung die Referenz aus und nehmen dann die Zuweisung vor. Beispiel:

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
```

```
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
  Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

Kapitel

20

Tabellen

Dieses Kapitel beschreibt die Verwendung der Datenmengenkomponente *TTable* in einer Datenbankanwendung. Eine Tabellenkomponente kapselt die vollständige Struktur und die Daten in der zugrundeliegenden Datenbanktabelle. Eine Tabellenkomponente erbt viele grundlegende Eigenschaften und Methoden der Klassen *TDataSet*, *TBDEDataSet*, und *TDBDataSet*. Sie sollten daher mit der allgemeinen Beschreibung von Datenmengen in »Datenmengen« und der BDE-spezifischen Beschreibung von Datenmengen in »BDE-Datenmengen« vertraut sein, bevor Sie sich mit den hier beschriebenen Eigenschaften und Methoden von Tabellenkomponenten befassen.

Tabellenkomponenten

Mit einer Tabellenkomponente kann auf jede Zeile und Spalte in der zugrundeliegenden Datenbanktabelle zugegriffen werden. Dabei ist es unerheblich, ob es sich um eine Paradox-, dBASE-, ODBC-kompatible (z.B. Microsoft Access) oder um eine SQL-Datenbank auf einem Remote-Server wie InterBase, Sybase oder SQL Server handelt.

Sie können Daten in jeder Zeile und Spalte einer Tabelle anzeigen und bearbeiten. Es kann mit einem Bereich von Datensätzen in einer Tabelle gearbeitet werden. Sie können alle Datensätze mit Filtern nach bestimmten Datensätzen durchsuchen, die von Ihnen festgelegte Kriterien erfüllen. Und Sie können Datensätze suchen, ganze Tabellen kopieren, umbenennen oder löschen und Haupt/Detail-Beziehungen erstellen.

Hinweis Eine Tabellenkomponente kann immer nur auf eine Datenbanktabelle zugreifen. Wenn Sie mit einer Komponente auf mehrere Tabellen oder nur auf bestimmte Zeilen und Spalten in einer oder mehreren Tabellen zugreifen wollen, müssen Sie eine Abfragekomponente statt einer Tabellenkomponente verwenden. Weitere Informationen über Abfragekomponenten finden Sie in Kapitel 21, »Abfragen«.

Tabellenkomponenten erstellen

Die folgenden allgemeinen Schritte zeigen, wie eine Tabellenkomponente zur Entwurfszeit erstellt wird. Eventuell müssen Sie zusätzlich die Eigenschaften der Tabelle an die Anforderungen der Anwendung anpassen.

- Gehen Sie wie folgt vor, um eine Tabellenkomponente zu erstellen:

  1. Plazieren Sie eine Tabellenkomponente aus der Registerkarte *Datenzugriff* der Komponentenpalette im Datenmodul bzw. Formular, und setzen Sie ihre Eigenschaft *Name* auf einen in der Anwendung eindeutigen Wert.
  2. Geben Sie in der Eigenschaft *DatabaseName* der Komponente die Datenbank an, auf die zugegriffen werden soll.
  3. Setzen Sie die Eigenschaft *TableName* auf den Namen der Datenbanktabelle. Wurde zuvor die Eigenschaft *DatabaseName* angegeben, kann die Tabelle aus der Dropdown-Liste gewählt werden.
  4. Fügen Sie eine Datenquellenkomponente in das Datenmodul oder Formular ein, und setzen Sie ihre Eigenschaft *DataSet* auf den Namen der Tabellenkomponente. Die Datenquellenkomponente wird verwendet, um eine Ergebnismenge aus der Tabelle an die datensensitiven Steuerelemente zur Anzeige zu übergeben.

- Führen Sie die folgenden Schritte durch, um auf die von der Tabellenkomponente gekapselten Daten zuzugreifen:

  5. Plazieren Sie eine Datenquellenkomponente aus der Registerkarte *Datenzugriff* der Komponentenpalette im Datenmodul bzw. Formular, und setzen Sie ihre Eigenschaft *DataSet* auf den Namen der Tabellenkomponente.
  6. Plazieren Sie im Formular ein datensensitives Steuerelement (z.B. *TDBGrid*), und geben Sie in dessen Eigenschaft *DataSource* den Namen der in Schritt 1 eingefügten Datenquellenkomponente an.
  7. Setzen Sie die Eigenschaft *Active* der Tabellenkomponente auf *True*.

Datenbankposition angeben

Die Eigenschaft *DatabaseName* legt fest, wo nach einer Datenbanktabelle gesucht wird. Bei Paradox- und dBASE-Tabellen kann für *DatabaseName* ein BDE-Alias oder ein explizit angegebener Verzeichnispfad verwendet werden. Bei SQL-Tabellen müssen Sie einen BDE-Alias verwenden.

Der Einsatz von BDE-Aliasen hat den Vorteil, daß die Datenquelle für die gesamte Anwendung einfach durch Änderung der Alias-Definition im SQL-Explorer geändert werden kann. Klicken Sie hierzu mit der rechten Maustaste in den Explorer, und wählen Sie *Umbenennen* aus dem lokalen Menü. Damit wird die BDE-Konfiguration geöffnet. Weitere Information zum Erstellen und Verwenden von BDE-Aliasen finden Sie in der Online-Hilfe des SQL-Explorers.

Führen Sie die folgenden Schritte durch, um die Eigenschaft *DatabaseName* zu setzen:

1. Weisen Sie der Eigenschaft *Active* der Tabelle gegebenenfalls den Wert *False* zu.
2. Geben Sie in der Eigenschaft *DatabaseName* den BDE-Alias oder den Verzeichnispfad an.

Tip Werden in Ihrer Anwendung Datenbanktransaktionen mit Datenbankkomponenten gesteuert, weisen Sie *DatabaseName* den für die Komponente definierten lokalen Alias zu. Weitere Informationen über Datenbankkomponenten finden Sie in Kapitel 17, »Datenbankverbindungen«.

Tabellennamen festlegen

Die Eigenschaft *TableName* bezeichnet die Tabelle in einer Datenbank, auf die mit der Tabellenkomponente zugegriffen werden soll. Die Angabe einer Tabelle erfordert folgende Schritte:

1. Weisen Sie der Eigenschaft *Active* der Tabelle gegebenenfalls den Wert *False* zu.
2. Geben Sie in der Eigenschaft *DatabaseName* einen BDE-Alias oder den Verzeichnispfad an. Weitere Informationen zum Einstellen von *DatabaseName* finden Sie unter »Datenbankposition angeben« auf Seite 20-2
3. Weisen Sie der Eigenschaft *TableName* die Tabelle zu, auf die zugegriffen wird. Während des Entwurfs können Sie aus der Dropdown-Liste der Eigenschaft *TableName* im Objektinspektor gültige Tabellennamen auswählen. Zur Laufzeit muß ein gültiger Name im Quelltext angegeben werden.

Nachdem ein gültiger Tabellenname angegeben wurde, kann der Eigenschaft *Active* der Tabellenkomponente der Wert *True* zugewiesen werden. Anschließend kann die Verbindung zur Datenbank hergestellt, die Tabelle geöffnet und können die Daten angezeigt und bearbeitet werden.

Zur Laufzeit können Sie die mit der Tabellenkomponente verknüpfte Tabelle bestimmen, indem Sie zwei Schritte durchführen:

- Setzen Sie die Eigenschaft *Active* auf *False*.
- Weisen Sie der Eigenschaft *TableName* einen gültigen Tabellennamen zu.

Im folgenden Beispiel wird für die Tabellenkomponente *OrderOrCustTable* der Tabellenname nach Maßgabe des aktuellen Tabellennamens geändert:

```
with OrderOrCustTable do
begin
   Active := False; {Tabelle schließen}
   if TableName = 'CUSTOMER.DB' then
    TableName := 'ORDERS.DB'
   else
    TableName := 'CUSTOMER.DB';
   Active := True; {Mit einer neuen Tabelle erneut öffnen}
end;
```

Tabellentypen für lokale Tabellen festlegen

Bei einem Zugriff auf Paradox-, dBASE- oder FoxPro-Tabellen oder ASCII-Tabellen mit Kommas als Trennzeichen (Format »Comma-Delimited«) ermittelt die BDE mit der Eigenschaft *TableType* den Typ der Tabelle (die erwartete Struktur). *TableType* hat keine Funktion, wenn eine Anwendung auf eine SQL-Tabelle auf einen Datenbankserver zugreift.

Der Vorgabewert von *TableType* ist *ttDefault*. Die Tabelle 20.1 zeigt die von der BDE erkannten Dateinamenserweiterungen und die entsprechenden Tabellentypen:

Tabelle 20.1 Dateinamenserweiterungen und Tabellentypen

Erweiterung	Tabellentyp
Keine Erweiterung	Paradox-Tabelle
.DB	Paradox-Tabelle
.DBF	dBASE-Tabelle
.TXT	ASCII-Tabelle

Werden für lokale Paradox-, dBASE- und ASCII-Tabellen die in obiger Tabelle aufgeführten Dateinamenserweiterungen eingesetzt, kann in *TableType* der Standardwert *ttDefault* verwendet werden. Andernfalls muß die Anwendung den richtigen Tabellentyp in der Eigenschaft *TableType* einstellen. Tabelle 20.2 führt die möglichen Werte für *TableType* auf:

Tabelle 20.2 Werte von TableType

Wert	Tabellentyp
ttDefault	Der Tabellentyp wird automatisch durch die BDE ermittelt
ttParadox	Paradox-Tabelle
ttDBase	dBASE-Tabelle
ttFoxPro	FoxPro-Tabelle
ttASCII	ASCII-Tabelle im Format Comma-Delimited

Tabellen öffnen und schließen

Damit Tabellendaten in einem datensensitiven Steuerelement wie *TDBGrid* angezeigt oder bearbeitet werden können, wird die Tabelle geöffnet. Sie können eine Tabelle auf zwei Arten öffnen. Sie können ihre Eigenschaft *Active* auf *True* setzen oder ihre Methode *Open* aufrufen. Beim Öffnen wird die Tabelle in den Tabellenlayout-Modus versetzt, und die Daten werden in aktiven Steuerelementen angezeigt, die mit der Datenquelle der Tabelle verknüpft sind.

Speichern oder verwerfen Sie anstehende Änderungen, bevor Sie die Anzeige oder die Bearbeitung der Daten beenden oder die Werte der grundlegenden Eigenschaften der Tabellenkomponente (zum Beispiel *DatabaseName*, *TableName* und *TableType*) ändern. Ist die Zwischenspeicherung von Aktualisierungen aktiviert, rufen Sie zum

Speichern der Änderungen in der Datenbank die Methode *ApplyUpdates* auf. Schließen Sie dann die Tabelle.

Sie können eine Tabelle auf zwei Arten schließen. Sie können ihre Eigenschaft *Active* auf *False* setzen oder ihre Methode *Close* aufrufen. Beim Schließen wird der Tabelle der Status *dsInactive* zugewiesen. Die mit der Datenquelle der Tabelle verknüpften aktiven Steuerelemente werden gelöscht.

Zugriff auf Tabellen steuern

Beim Öffnen einer Tabelle fordert diese standardmäßig das Lese- und Schreibrecht für die zugrundeliegende Datenbanktabelle an. In Abhängigkeit von den Merkmalen der zugrundeliegenden Datenbanktabelle ist es möglich, daß das angeforderte Recht nicht erteilt wird (wenn Sie beispielsweise den Schreibzugriff auf eine SQL-Tabelle auf einem Remote-Server anfordern, für den Sie nur Leserechte besitzen).

Mit den Eigenschaften *CanModify*, *ReadOnly* und *Exclusive* der Tabellenkomponenten können die Rechte für den Zugriff auf eine Tabelle beeinflußt werden.

Mit der schreibgeschützten Eigenschaft *CanModify* wird festgelegt, ob eine Tabellenkomponente für die zugrundeliegende Datenbanktabelle Lese- und Schreibrechte besitzt. Nach dem Öffnen einer Tabelle zur Laufzeit ermittelt die Anwendung durch Abfragen der Eigenschaft *CanModify*, ob das Schreibrecht für die Tabelle gewährt wurde. Hat *CanModify* den Wert *False*, kann die Anwendung nicht in die Datenbank schreiben. Hat *CanModify* den Wert *True*, kann die Anwendung in die Datenbank schreiben, wenn die Eigenschaft *ReadOnly* der Tabelle auf *False* gesetzt ist.

Mit der Eigenschaft *ReadOnly* wird festgelegt, ob ein Benutzer Daten anzeigen und bearbeiten kann. Ist *ReadOnly* auf *False* gesetzt (Voreinstellung), kann der Benutzer Daten anzeigen und bearbeiten. Soll der Benutzer die Daten nur anzeigen können, müssen Sie *ReadOnly* vor dem Öffnen der Tabelle den Wert *True* zuweisen.

Die Eigenschaft *Exclusive* bestimmt, ob eine Anwendung exklusive Zugriffsrechte für eine Paradox-, dBASE- oder FoxPro-Tabelle besitzt. Wollen Sie exklusiv auf Tabellen dieser Typen zugreifen, weisen Sie der Eigenschaft *Exclusive* der Tabellenkomponente vor dem Öffnen der Tabelle den Wert *True* zu. Wird die Tabelle erfolgreich geöffnet, können andere Anwendungen nicht auf diese Tabelle zugreifen. Ein exklusiver Zugriff ist nicht möglich, wenn bereits auf die Tabelle zugegriffen wird.

Die folgenden Anweisungen öffnen eine Tabelle für den exklusiven Zugriff:

```
CustomersTable.Exclusive := True; {Anforderung für exklusiven Zugriff}
CustomersTable.Active := True; {Tabelle öffnen}
```

Hinweis Einige Server unterstützen den exklusiven Zugriff auf SQL-Tabellen nicht. Andere unterstützen eventuell exklusive Sperren, gewähren aber anderen Anwendungen Leserechte für die Tabelle. Weitere Informationen zum exklusiven Sperren von Datenbanktabellen finden Sie in der Server-Dokumentation.

Datensätze suchen

Zum Auffinden eines bestimmten Datensatzes in einer Tabelle gibt es verschiedene Möglichkeiten. Die generischen Methoden *Locate* und *Lookup* sind besonders für die Suche eines Datensatzes geeignet. Mit diesen Methoden können Sie in Feldern beliebigen Typs in allen Tabellentypen suchen. Es muß sich nicht um indizierte oder Schlüsselfelder handeln.

- *Locate* setzt den Cursor in die erste Datenzeile, die den angegebenen Suchkriterien entspricht.
- *Lookup* gibt Werte aus der ersten Datenzeile zurück, die mit den angegebenen Suchkriterien übereinstimmt, ändert dabei jedoch nicht die Position des Cursors.

Sie können *Locate* und *Lookup* nicht nur für *TTable*-Komponenten, sondern für alle Datenmengen verwenden. Weitere Informationen zu den Methoden *Locate* und *Lookup* finden Sie in Kapitel 18, »Datenmengen«.

Von Tabellenkomponenten werden auch die Methoden *Goto* und *Find* unterstützt. Die Methoden werden hier beschrieben, um die Arbeit mit älteren Anwendungen zu erleichtern. Sie sollten aber in neuen Anwendungen die Methoden *Locate* und *Lookup* verwenden. Möglicherweise können Sie die Ausführungsgeschwindigkeit alter Anwendungen steigern, indem Sie die alten Methoden durch diese neuen Methoden ersetzen.

Datensätze über indizierte Felder suchen

Aus Kompatibilitätsgründen unterstützen Tabellenkomponenten auch verschiedene Goto-Suchmethoden. Mit Goto-Suchmethoden können Sie mit Hilfe indizierter Felder (Schlüssel) nach Datensätzen suchen und den ersten gefundenen zum neuen aktuellen Datensatz machen.

Bei Paradox- und dBASE-Tabellen muß der Schlüssel immer ein Index sein, den Sie mit Hilfe der Eigenschaft *IndexName*einer Tabellenkomponente angeben können. Bei SQL-Tabellen kann der Index auch aus einer mittels der Eigenschaft *IndexFieldNames* definierten Liste von Feldern bestehen. Auch für Paradox- oder dBASE-Tabellen kann eine Feldliste angegeben werden, doch müssen die dafür benutzten Felder indiziert sein. Weitere Informationen über *IndexName* und *IndexFieldNames* finden Sie unter »Datensätze über Sekundärindizes suchen« auf Seite 20-9.

Tip Verwenden Sie *Locate*, um in nichtindizierten Feldern einer Paradox- oder dBASE-Tabelle nach Daten zu suchen. Alternativ können Sie auch eine *TQuery*-Komponente und eine SELECT-Anweisung verwenden. Weitere Informationen zu *TQuery* finden Sie in Kapitel 21, »Abfragen«.

Mit den folgenden Goto- und Find-Methoden können Datensätze in einer Anwendung gesucht werden:

Tabelle 20.3 Alte TTable-Suchmethoden

Methode	Beschreibung
EditKey	Schützt den aktuellen Inhalt des Suchschlüsselpuffers und ändert den Modus der Tabelle in *dsSetKey*. Die Anwendung kann bestehende Suchkriterien vor dem Ausführen der Suche ändern.
FindKey	Faßt die Methoden *SetKey* und *GotoKey* in einer Methode zusammen.
FindNearest	Faßt die Methoden *SetKey* und *GotoNearest* in einer Methode zusammen.
GotoKey	Sucht nach dem ersten Datensatz in einer Datenmenge, der den Suchkriterien entspricht. Bei Erfolg wird der Cursor auf diesen Datensatz gesetzt.
GotoNearest	Sucht nach dem Datensatz, der die größte Übereinstimmung mit den Schlüsselwerten aufweist, und setzt den Cursor auf diesen Datensatz. Diese Methode kann nur für String-Felder verwendet werden.
SetKey	Löscht den Suchschlüsselpuffer und ändert den Modus der Tabelle in *dsSetKey*. Die Anwendung kann bestehende Suchkriterien vor dem Ausführen der Suche ändern.

GotoKey und *FindKey* sind Boolesche Funktionen, die bei erfolgreicher Suche den Cursor auf den gefundenen Datensatz setzen und den Wert *True* zurückgeben. Kann kein entsprechender Datensatz gefunden werden, wird *False* zurückgegeben und der Cursor nicht verschoben.

Die Methoden *GotoNearest* and *FindNearest* verschieben den Cursor immer. Bei genauer Übereinstimmung wird der Cursor auf den entsprechenden Datensatz gesetzt. Andernfalls wird der Cursor auf den ersten Datensatz gesetzt, der größer als das angegebene Suchkriterium ist.

Datensätze mit Goto-Methoden suchen

Folgendermaßen führen Sie eine Suche mit Goto-Methoden durch:

1. Geben Sie gegebenenfalls den Index für die Suche in der Eigenschaft *IndexName* an. (Legen Sie bei SQL-Tabellen die Felder für den Schlüssel mit *IndexFieldNames* fest.) Wird der Primärindex der Tabelle verwendet, müssen Sie diese Eigenschaften nicht angeben.
2. Öffnen Sie die Tabelle.
3. Ändern Sie mit der Methode *SetKey* den Modus der Tabelle in *dsSetKey*.
4. Geben Sie die Suchwerte in der Eigenschaft *Fields* an. *Fields* ist eine String-Liste, bei der auf die verschiedenen Strings über ordinale Indizes zugegriffen wird, die den Spalten der Tabelle entsprechen. Die erste Spalte in der Tabelle hat den Index 0.
5. Führen Sie die Suchoperation mit *GotoKey* oder *GotoNearest* durch. Es wird dann nach dem ersten übereinstimmenden Datensatz gesucht. Der Cursor wird gegebenenfalls auf den gefundenen Datensatz gesetzt.

Der folgende Programmcode ist dem Ereignis *OnClick* einer Schaltfläche zugeordnet. Die Schaltfläche startet die Suche nach einem Feldwert, der genau mit dem Text in einem Eingabefeld übereinstimmt. Der Cursor wird auf den gefundenen Datensatz gesetzt:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
   Table1.SetKey;
   Table1.Fields[0].AsString := Edit1.Text;
   if not Table1.GotoKey then
    ShowMessage('Datensatz nicht gefunden');
end;
```

Die Methode *GotoNearest* führt eine ähnliche Operation durch. Sie sucht nach einer Übereinstimmung mit dem Teil eines Feldwertes und kann nur für String-Spalten verwendet werden. Ein Beispiel:

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

Wird ein Datensatz mit den Anfangsbuchstaben »Sm« gefunden, setzt die Methode den Cursor auf diesen. Andernfalls wird der Cursor nicht verschoben, und *GotoNearest* gibt *False* zurück.

Datensätze mit Find-Methoden suchen

Folgendermaßen führen Sie eine Suche mit Find-Methoden durch:

1. Geben Sie den Index für die Suche bei Bedarf in der Eigenschaft *IndexName* an. (Legen Sie bei SQL-Tabellen die Felder für den Schlüssel in *IndexFieldNames* fest.) Wird der Primärindex der Tabelle verwendet, müssen Sie diese Eigenschaften nicht angeben.
2. Öffnen Sie die Tabelle.
3. Führen Sie eine Suche mit *FindKey* oder *FindNearest* durch. Es wird dabei nach dem ersten Datensatz mit exakter oder der größten Übereinstimmung gesucht, auf den dann der Cursor gesetzt wird. Beide Methoden verwenden nur einen Parameter: ein Array mit durch Kommas getrennten Feldwerten. Jeder Wert entspricht einer indizierten Spalte in der zugrundeliegenden Tabelle.

Hinweis Die Methode *FindNearest* kann nur für String-Felder verwendet werden.

Aktuellen Datensatz nach einer erfolgreichen Suche bestimmen

Bei einer erfolgreichen Suche wird der Cursor normalerweise auf den ersten Datensatz gesetzt, der den Suchkriterien entspricht. Sie können die Eigenschaft *KeyExclusive* einer Tabellenkomponente auf *True* setzen, um den Cursor nicht auf den ersten übereinstimmenden, sondern den nächsten Datensatz zu setzen.

Per Voreinstellung hat *KeyExclusive* den Wert *False*. Der Cursor wird dann bei einer erfolgreichen Suche auf den ersten übereinstimmenden Datensatz gesetzt.

Datensätze über Teilschlüssel suchen

Soll in einer Tabelle mit mehreren Schlüsselspalten nur eine Teilmenge der Schlüssel für eine Suche verwendet werden, setzen Sie *KeyFieldCount* auf die Anzahl der gewünschten Spalten. Hat eine Tabelle beispielsweise einen aus drei Spalten bestehenden Primärschlüssel und soll nur die erste Spalte in der Suche berücksichtigt werden, setzen Sie *KeyFieldCount* auf 1.

In Tabellen mit mehrspaltigen Schlüsseln können nur aufeinanderfolgende Spalten verwendet werden, beginnend mit der ersten. Ist beispielsweise ein aus drei Spalten bestehender Schlüssel vorhanden, können Sie für die Suche die erste Spalte, die erste und die zweite Spalte oder die erste, zweite und dritte Spalte verwenden. Die Verwendung der ersten und dritten Spalte ist nicht möglich.

Datensätze über Sekundärindizes suchen

Soll ein anderer als der primäre Index für die Suche in einer Tabelle verwendet werden, müssen Sie den betreffenden Indexnamen in der Eigenschaft *IndexName* der Tabelle festlegen. Diese Eigenschaft kann nur bei geschlossenen Tabellen bearbeitet werden. Besitzt beispielsweise die Tabelle CUSTOMER den Sekundärindex »CityIndex« und soll mit Hilfe dieses Index nach einem Wert gesucht werden, müssen Sie *IndexName* vor Beginn der Suchoperation auf »CityIndex« setzen:

```
Table1.Close;
Table1.IndexName := 'CityIndex';
Table1.Open;
Table1.SetKey;
Table1['City'] := Edit1.Text;
Table1.GotoNearest;
```

Sie können auch statt des Indexnamens die als Schlüssel zu verwendenden Felder in der Eigenschaft *IndexFieldNames* angeben. Bei Paradox- und dBASE-Tabellen müssen die angegebenen Felder indiziert sein, da andernfalls beim Ausführen der Suche eine Exception ausgelöst wird. In SQL-Tabellen müssen die angegebenen Felder nicht indiziert sein.

Suchoperationen wiederholen oder erweitern

Bei jedem Aufruf der Methode *SetKey* oder *FindKey* wird der vorherige Wert der Eigenschaft *Fields* gelöscht. Um eine Suchoperation mit den zuvor festgelegten Feldern zu wiederholen oder weitere Felder hinzuzufügen, rufen Sie *EditKey* anstelle von *SetKey* oder *FindKey* auf. Wenn beispielsweise der Index »CityIndex« die beiden Felder CITY und COUNTRY enthält, gehen Sie folgendermaßen vor, um einen Datensatz mit einem bestimmten Firmennamen in einer bestimmten Stadt zu finden:

```
Table1.EditKey;
Table1['Country'] := Edit2.Text;
Table1.GotoNearest;
```

Datensätze sortieren

Ein Index bestimmt die Reihenfolge, in der die Datensätze einer Tabelle angezeigt werden. Normalerweise werden Datensätze aufsteigend in der Reihenfolge des Primärindex angezeigt (bei dBASE-Tabellen ohne Primärindex basiert die Sortierreihenfolge auf der physischen Reihenfolge der Datensätze). Dieses Standardverhalten setzt keine Programmierung voraus. Soll jedoch eine andere Reihenfolge verwendet werden, müssen Sie folgendes angeben:

- einen sekundären Index oder
- eine Liste der Spalten, nach denen sortiert werden soll (nur SQL).

Zur Festlegung einer anderen Sortierreihenfolge führen Sie die folgenden Schritte durch:

1. Ermitteln Sie die verfügbaren Indizes.
2. Legen Sie einen sekundären Index oder eine Spaltenliste fest, nach der sortiert werden soll.

Verfügbare Indizes mit GetIndexNames abrufen

Zur Laufzeit können Sie mit der Methode *GetIndexNames* eine Liste der für eine Tabelle verfügbaren Indizes abrufen. Die Indizes werden dabei in Form einer String-Liste mit gültigen Indexnamen zurückgegeben. Im folgenden Beispiel werden die verfügbaren Indizes der Datenmenge *CustomersTable* ermittelt:

```
var
   IndexList: TList;
...
CustomersTable.GetIndexNames(IndexList);
```

Hinweis Bei Paradox-Tabellen hat der Primärindex keinen Namen und kann daher nicht von *GetIndexNames* zurückgegeben werden. Muß nach Nutzung eines Sekundärindex erneut der Primärindex verwendet werden, weisen Sie der Eigenschaft *IndexName* der Tabelle einen Null-String zu.

Sekundärindizes in IndexName festlegen

Soll eine Tabelle nach einem Sekundärindex sortiert werden, geben Sie den betreffenden Indexnamen in der Eigenschaft *IndexName* der Tabellenkomponente an. Während des Entwurfs können Sie den Indexnamen im Objektinspektor angeben, und zur Laufzeit kann auf die Eigenschaft im Programmcode zugegriffen werden. In der folgenden Anweisung wird der Index der Tabellenkomponente *CustomersTable* auf *CustDescending* gesetzt:

```
CustomersTable.IndexName := 'CustDescending';
```

Informationen zu dBASE-Indizes finden Sie unter »dBASE-Indexdateien angeben«.

dBASE-Indexdateien angeben

Bei dBASE-Tabellen, die nichtgewartete Indizes verwenden, müssen Sie vor dem Setzen von *IndexName* der Eigenschaft *IndexFiles* die Namen der zu verwendenden Indexdateien zuweisen. Während des Entwurfs können Sie in der Wertespalte von *IndexFiles* im Objektinspektor den Indexdateien-Editor öffnen.

Soll eine Liste der verfügbaren Indexdateien angezeigt werden, klicken Sie auf *Hinzufügen* und wählen dann eine oder mehrere Dateien aus der Liste. Eine dBASE-Indexdatei kann mehrere Indizes enthalten. Wählen Sie den gewünschten Index der Indexdatei aus der Dropdown-Liste von *IndexName* im Objektinspektor. Sie können auch mehrere Indizes gleichzeitig angeben, indem Sie die Namen durch Semikolons getrennt eingeben.

IndexFiles und *IndexName* können auch zur Laufzeit gesetzt werden. So wird im folgenden Beispiel die Eigenschaft *IndexFiles* der Tabellenkomponente *AnimalsTable* auf ANIMALS.MDX und anschließend *IndexName* auf NAME gesetzt:

```
AnimalsTable.IndexFiles := 'ANIMALS.MDX';
AnimalsTable.IndexName := 'NAME';
```

Sortierreihenfolge für SQL-Tabellen

In einer SQL-Tabelle wird die Reihenfolge der Spalten mit der Klausel ORDER BY festgelegt. Der von der Klausel verwendete Index kann folgendermaßen angegeben werden:

- mit der Eigenschaft *IndexName*, um einen vorhandenen Index zu verwenden, oder
- mit der Eigenschaft *IndexFieldNames*, um anhand einer Teilmenge der Spalten einen Pseudoindex zu erstellen.

Die Eigenschaften *IndexName* und *IndexFieldNames* schließen sich gegenseitig aus. Das Setzen der einen Eigenschaft löscht die andere. Informationen über die Verwendung von *IndexName* finden Sie in unter »Datensätze über Sekundärindizes suchen« auf Seite 20-9.

Felder in IndexFieldNames angeben

Die Eigenschaft *IndexFieldNames* enthält eine String-Liste. Um eine bestimmte Sortierreihenfolge festzulegen, geben Sie alle Spaltennamen in der gewünschten Reihenfolge getrennt durch Semikolons an. Die Sortierung erfolgt nur in aufsteigender Richtung. Es hängt vom Server ab, ob bei der Sortierung zwischen Groß- und Kleinschreibung unterschieden wird. Entsprechende Informationen finden Sie in der Dokumentation zum Server.

Mit der folgenden Anweisung wird die Tabelle *PhoneTable* zuerst nach der Spalte LASTNAME und dann nach der Spalte FIRSTNAME sortiert:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

Hinweis Wird *IndexFieldNames* für Paradox- oder dBASE-Tabellen verwendet, sucht Delphi einen Index, der die angegebenen Spalten verwendet. Schlägt die Operation fehl, wird eine Exception ausgelöst.

Feldlisten nach einem Index durchsuchen

Wird in einer Anwendung zur Laufzeit ein Index verwendet, können mit Hilfe der folgenden Eigenschaften Informationen abgerufen werden:

- *IndexFieldCount* enthält die Anzahl der Spalten im aktuellen Index.
- *IndexFields* enthält eine Liste der Spaltennamen, aus denen der Index besteht.

IndexFields ist eine String-Liste mit den Spaltennamen des Index. Das folgende Beispiel zeigt, wie Sie mit *IndexFieldCount* und *IndexFields* die Spaltenliste in einer Schleife verarbeiten können:

```
var
   I: Integer;
   ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
   begin
   for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I];
   end;
end;
```

Hinweis *IndexFieldCount* ist nicht gültig, wenn eine dBASE-Tabelle mit einem berechneten Index geöffnet wurde.

Mit Teilmengen der Daten arbeiten

Tabellen können sehr groß werden, so daß in Anwendungen häufig die Anzahl der zu verarbeitenden Datensätze eingeschränkt werden muß. Bei Tabellenkomponenten gibt es zwei Möglichkeiten, die von einer Anwendung abgerufenen Datensätze zu beschränken: Filter und Bereiche. Filter können für alle Datenmengen (einschließlich *TTable*-, *TQuery*- und *TStoredProc*-Komponenten) verwendet werden. Ausführliche Informationen zu Filtern finden Sie in Kapitel 18, »Datenmengen«.

Bereiche können nur für *TTable*-Komponenten verwendet werden. Trotz ihrer Gemeinsamkeiten sind Bereiche und Filter für unterschiedliche Einsatzgebiete vorgesehen. Im folgenden Abschnitt werden die Unterschiede zwischen Bereichen und Filtern und die Verwendung von Bereichen beschrieben.

Unterschiede zwischen Bereichen und Filtern

Bereiche und Filter beschränken die sichtbaren Datensätze auf eine Teilmenge der verfügbaren Datensätze, verwenden dabei aber unterschiedliche Verfahren. Ein Bereich besteht aus fortlaufend indizierten Datensätzen, die in einem angegebenen

Wertebereich liegen. In einer Datenbank mit Arbeitnehmerdaten, die als Index den Nachnamen verwendet, können Sie zum Beispiel mit einem Bereich alle Arbeitnehmer anzeigen, deren Nachnamen größer als »Müller« und kleiner als »Schmitt« sind. Da Bereiche auf Indizes basieren, können Bereiche nur für indizierte Paradox- und dBase-Tabellen verwendet werden. (Bei SQL-Tabellen können Bereiche allen Feldern zugewiesen werden, die in der Eigenschaft *IndexFieldNames* angegeben sind). Bereiche können nur basierend auf existierenden Indizes geordnet werden.

Ein Filter ist dagegen eine Menge von Datensätzen, in denen bestimmte Daten übereinstimmen. Die Menge wird unabhängig von der Indizierung gebildet. Sie können zum Beispiel in einer Datenbank mit Arbeitnehmerdaten alle Arbeitnehmer anzeigen, die in Kalifornien leben und mehr als fünf Jahre für die Firma tätig sind. Obwohl Filter verfügbare Indizes nutzen können, sind diese keine Voraussetzung. Filter werden nacheinander jedem Datensatz in einer Datenmenge zugewiesen.

Im allgemeinen sind Filter flexibler als Bereiche. Bereiche sind jedoch effizienter, wenn die Datenmengen sehr groß sind und die benötigten Datensätze bereits in fortlaufenden Gruppen vorliegen. Bei sehr großen Datenmengen kann eine Abfragekomponente für die Anzeige und Bearbeitung der Daten besser geeignet sein. Weitere Informationen zu Filtern finden Sie in Kapitel 18, »Datenmengen«. Zusätzliche Informationen zu Abfragen finden Sie in Kapitel 21, »Abfragen«.

Bereiche erstellen und zuweisen

Folgendermaßen können Sie einen Bereich erstellen und zuweisen:

1. Die Datenmenge muß in den Modus *dsSetKey* versetzt werden. Außerdem muß der Indexwert des Bereichsanfangs angegeben werden.
2. Das Bereichsende muß angegeben werden.
3. Der Bereich muß der Datenmenge zugewiesen werden.

Bereichsanfang festlegen

Mit der Methode *SetRangeStart* können Sie die Datenmenge in den Modus *dsSetKey* versetzen und die Liste mit Werten für den Bereichsanfang erstellen. Nach einem Aufruf von *SetRangeStart* werden nachfolgende Zuweisungen an die Eigenschaft *Fields* bei der Anwendung des Bereichs als Indexwerte für den Bereichsanfang verwendet. Wenn Sie Paradox- oder dBASE-Tabellen verwenden, müssen die angegebenen Felder indiziert sein.

Ein Beispiel: In einer Anwendung wird die Tabellenkomponente *Customers* verwendet, die mit der Tabelle CUSTOMER verknüpft ist. Außerdem wurden für alle Felder in der Datenmenge *Customers* persistente Feldkomponenten erstellt. Die Tabelle CUSTOMER ist nach der ersten Spalte (CUSTNO) indiziert. In einem Formular befinden sich die zwei Eingabefelder *StartValue* und *EndValue*, mit denen Anfangs- und Endwerte für einen Bereich angegeben werden können. Mit folgendem Quelltext kann dann ein Bereich erstellt und zugewiesen werden:

```
with Customers do
begin
```

```
    SetRangeStart;
    FieldByName('CustNo') := StartVal.Text;
    SetRangeEnd;
    if EndVal.Text <> '' then
     FieldByName('CustNo') := EndVal.Text;
    ApplyRange;
end
```

Vor der Zuweisung eines Wertes an *Fields* wird überprüft, ob der in *EndVal* enthaltene Wert ungleich Null ist. Wird für *StartVal* der Wert Null angegeben, werden alle Datensätze vom Beginn der Tabelle an berücksichtigt. Ist jedoch der für *EndVal* vorgegebene Wert Null, enthält der Bereich keine Datensätze.

Bei einem Index mit mehreren Spalten können Sie einen Anfangswert für beliebig viele Felder im Index angeben. Wird für ein Feld kein Wert angegeben, wird beim Zuweisen des Bereichs für dieses Feld ein Nullwert verwendet. Geben Sie mehr Werte an als Indexfelder vorhanden sind, werden die überzähligen Felder bei der Berechnung des Bereichs ignoriert.

Beenden Sie die Angabe des Bereichs durch einen Aufruf der Methode *SetRangeEnd* oder *ApplyRange*. Diese Methoden werden in den folgenden Abschnitten beschrieben.

Tip Soll der Bereich mit dem ersten Datensatz der Datenmenge beginnen, müssen Sie *SetRangeStart* nicht aufrufen.

Sie können die Werte für den Anfang und das Ende des Bereichs auch mit der Methode *SetRange* einstellen. Weitere Informationen zu *SetRange* finden Sie unter »Bereichsanfang und -ende festlegen« auf Seite 20-15.

Bereichsende festlegen

Mit der Methode *SetRangeEnd* können Sie die Datenmenge in den Modus *dsSetKey* versetzen und die Liste mit Werten für das Bereichsende erstellen. Nach einem Aufruf von *SetRangeEnd* werden nachfolgende Zuweisungen an die Eigenschaft *Fields* bei der Anwendung des Bereichs als Indexwerte für das Bereichsende verwendet. Wenn Sie Paradox- oder dBASE-Tabellen verwenden, müssen die angegebenen Felder indiziert sein.

Hinweis Das Bereichsende muß immer angegeben werden, auch wenn das Ende des Bereichs der letzte Datensatz der Datenmenge sein soll. Werden keine Endwerte angegeben, wird ein Nullwert verwendet. In diesem Fall ist der Bereich immer leer.

Sie können mit der Methode *FieldByName* Endwerte zuweisen. Ein Beispiel:

```
with Table1 do
begin
   SetRangeStart;
   FieldByName('LastName') := Edit1.Text;
   SetRangeEnd;
   FieldByName('LastName') := Edit2.Text;
   ApplyRange;
end;
```

Bei einem Index mit mehreren Spalten können Sie einen Anfangswert für beliebig viele Felder im Index angeben. Wenn Sie für ein Feld keinen Wert angeben, wird beim Zuweisen des Bereichs ein Nullwert verwendet. Geben Sie mehr Werte an als Indexfelder vorhanden sind, wird eine Exception ausgelöst.

Schließen Sie die Angabe des Bereichsendes durch Aufruf der Methode *ApplyRange* ab. Weitere Informationen finden Sie unter »Bereiche zuweisen« auf Seite 20-16.

Hinweis Sie können die Werte für den Anfang und das Ende des Bereichs mit der Methode *SetRange* einstellen. *SetRange* wird im nächsten Abschnitt beschrieben.

Bereichsanfang und -ende festlegen

Sie können mit der Methode *SetRange* die Datenmenge in den Modus *dsSetKey* versetzen und den Bereichsanfang und das Bereichsende einstellen, ohne die Methoden *SetRangeStart* und *SetRangeEnd* einzeln aufzurufen.

SetRange verwendet zwei Array-Konstanten als Parameter: Anfangswerte und Endwerte. Die folgende Anweisung erstellt beispielsweise basierend auf einem zweispaltigen Index einen Bereich:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

Bei einem Index mit mehreren Spalten können Sie Anfangs- und Endwerte für beliebig viele Felder im Index angeben. Wenn Sie für ein Feld keinen Wert angeben, wird beim Zuweisen des Bereichs ein Nullwert verwendet. Wollen Sie das erste Feld in einem Index übergehen und für die folgenden Felder Werte angeben, müssen Sie dem ersten Feld einen Null-Wert zuweisen. Geben Sie mehr Werte an als Indexfelder vorhanden sind, werden die überzähligen Felder bei der Berechnung des Bereichs ignoriert.

Hinweis Das Bereichsende muß immer angegeben werden, auch wenn das Ende des Bereichs der letzte Datensatz der Datenmenge sein soll. Werden keine Endwerte angegeben, wird als Endwert Null verwendet. In diesem Fall ist der Bereich leer.

Bereiche mit Teilschlüsseln festlegen

Besteht ein Index aus einem oder mehreren String-Feldern, kann mit den *SetRange*-Methoden ein Bereich auch anhand einer Teilmenge der indizierten Spalten (Teilschlüssel) festgelegt werden. Basiert ein Index beispielsweise auf den Spalten LASTNAME und FIRSTNAME, sind folgende Bereichsangaben zulässig:

```
Table1.SetRangeStart;
Table1['LastName'] := 'Smith';
Table1.SetRangeEnd;
Table1['LastName'] := 'Zzzzzz';
Table1.ApplyRange;
```

In diesem Beispiel werden alle Datensätze in den Bereich aufgenommen, in denen LASTNAME größer oder gleich »Smith« ist. Die Angabe könnte auch folgendermaßen lauten:

```
Table1['LastName'] := 'Sm';
```

Durch diese Anweisung werden alle Datensätze berücksichtigt, in denen das Feld LASTNAME größer oder gleich »Sm« ist. Im folgenden Beispiel werden alle Datensätze eingeschlossen, in denen LASTNAME größer oder gleich »Smith« und FIRSTNAME größer oder gleich »J« ist:

```
Table1['LastName'] := 'Smith';
Table1['FirstName'] := 'J';
```

Datensätze ein- oder ausschließen, die mit Bereichsgrenzen übereinstimmen

In der Standardeinstellung umfaßt ein Bereich alle Datensätze, die größer oder gleich dem angegebenen Anfangswert und kleiner oder gleich dem angegebenen Endwert sind. Sie können dieses Verhalten mit der Eigenschaft *KeyExclusive* steuern. In der Voreinstellung ist *KeyExclusive* auf *False* gesetzt.

Sie können die Eigenschaft *KeyExclusive* einer Tabellenkomponente auf *True* setzen, um die Datensätze auszuschließen, die mit dem angegebenen Bereichsende identisch sind. Ein Beispiel:

```
KeyExclusive := True;
Table1.SetRangeStart;
Table1['LastName'] := 'Smith';
Table1.SetRangeEnd;
Table1['LastName'] := 'Tyler';
Table1.ApplyRange;
```

In diesem Beispiel werden alle Datensätze in den Bereich aufgenommen, in denen LASTNAME größer oder gleich »Smith« und kleiner als »Tyler« ist.

Bereiche zuweisen

Mit den im vorhergehenden Abschnitt beschriebenen *SetRange*-Methoden kann der Bereich definiert, aber nicht zugewiesen, also wirksam gemacht werden. Rufen Sie dazu die Methode *ApplyRange* auf. Nach dem Aufruf dieser Methode ist eine Teilmenge der Datensätze in der Datenbanktabelle für die Anwendung sichtbar.

Bereiche entfernen

Die Methode *CancelRange* entfernt die Bereichsbeschränkungen. Nach dem Aufruf dieser Methode kann wieder auf alle Datensätze zugegriffen werden. Die Bereichsgrenzen bleiben erhalten, so daß zu einem späteren Zeitpunkt der Bereich erneut zugewiesen werden kann. Das folgende Beispiel zeigt die Verwendung:

```
%
Table1.CancelRange;
%
{Denselben Bereich später erneut verwenden. SetRangeStart usw. müssen nicht aufgerufen
werden.}
Table1.ApplyRange;
%
```

Bereiche ändern

Es stehen zwei Methoden zur Verfügung, mit denen bestehende Bereichsgrenzen geändert werden können. Mit *EditRangeStart* können Anfangswerte und mit *EditRangeEnd* Endwerte von Bereichen geändert werden.

Die Bearbeitung und Zuweisung eines Bereichs umfaßt folgende Schritte:

1. Die Datenmenge in den Modus *dsSetKey* versetzen und den Indexwert für den Bereichsanfang ändern.
2. Den Indexwert für das Bereichsende ändern.
3. Den Bereich der Datenmenge zuweisen.

Sie können eine oder beide Bereichsgrenzen ändern. Ändern Sie die Bereichsgrenzen eines aktiven Bereichs, werden die Änderungen erst wirksam, wenn Sie *ApplyRange* erneut aufrufen.

Bereichsanfang ändern

Mit der Methode *EditRangeStart* können Sie die Datenmenge in den Modus *dsSetKey* versetzen und die aktuelle Liste der Startwerte für den Bereichsanfang ändern. Nach einem Aufruf von *EditRangeStart* werden nachfolgende Zuweisungen an die Eigenschaft *Fields* bei der Zuweisung des Bereichs als Indexwerte für den Bereichsanfang verwendet. Wenn Sie Paradox- oder dBASE-Tabellen verwenden, müssen die angegebenen Felder indiziert sein.

Tip Mit *EditRangeStart* können Sie den Startwert für den Bereichsanfang ändern, nachdem Sie zuvor einen Teilschlüssel angegeben haben. Weitere Informationen zu Bereichen, deren Grenzen mit Teilschlüsseln festgelegt werden, finden Sie unter »Bereiche mit Teilschlüsseln festlegen« auf Seite 20-15.

Bereichsende ändern

Mit der Methode *EditRangeEnd* können Sie die Datenmenge in den Modus *dsSetKey* versetzen und die aktuelle Liste der Endwerte für das Bereichsende ändern. Nach einem Aufruf von *EditRangeEnd* werden nachfolgende Zuweisungen an die Eigenschaft *Fields* bei der Zuweisung des Bereichs als Indexwerte für das Bereichsende verwendet. Wenn Sie Paradox- oder dBASE-Tabellen verwenden, müssen die angegebenen Felder indiziert sein.

Hinweis Das Bereichsende muß immer angegeben werden, auch wenn das Ende des Bereichs der letzte Datensatz der Datenmenge sein soll. Werden keine Endwerte angegeben, wird als Endwert Null verwendet. In diesem Fall ist der Bereich immer leer.

Alle Datensätze einer Tabelle löschen

Zur Laufzeit können Sie mit der Methode *EmptyTable* einer Tabellenkomponente alle Datensätze aus der Tabelle löschen. Bei SQL-Tabellen kann diese Methode nur verwendet werden, wenn Sie die entsprechenden Zugriffsrechte für die Tabelle besitzen. Die folgende Anweisung löscht beispielsweise alle Datensätze einer Datenmenge:

```
PhoneTable.EmptyTable;
```

Achtung Mit *EmptyTable* gelöschte Daten können nicht wiederhergestellt werden.

Tabellen löschen

Wenn eine Tabelle während des Entwurfs aus einer Datenbank gelöscht werden soll, klicken Sie mit der rechten Maustaste auf die Tabellenkomponente und wählen aus dem lokalen Menü den Eintrag *Tabelle löschen*. Diese Option ist allerdings nur verfügbar, wenn die Tabellenkomponente eine vorhandene Datenbanktabelle repräsentiert (die Eigenschaften *DatabaseName* und *TableName* sind hier relevant).

Mit der Methode *DeleteTable* einer Tabellenkomponente können Sie eine Tabelle zur Laufzeit löschen. Die folgende Anweisung entfernt zum Beispiel die einer Datenmenge zugrundeliegende Tabelle:

```
CustomersTable.DeleteTable;
```

Achtung Mit *DeleteTable* gelöschte Daten können nicht wiederhergestellt werden.

Tabellen umbenennen

Wenn eine Paradox- oder dBase-Tabelle zur Entwurfszeit umbenannt werden soll, klicken Sie mit der rechten Maustaste auf die Tabellenkomponente und wählen aus dem lokalen Menü den Eintrag *Tabelle umbenennen*. Alternativ dazu können Sie den Namen überschreiben, der im Objektinspektor neben der Eigenschaft *TableName* angezeigt wird. Im Anschluß an die Änderung erscheint ein Dialogfeld mit einer Rückfrage, ob die Tabelle umbenannt werden soll. Sie können die Aktion jetzt bestätigen oder abbrechen. Ein Abbrechen ist beispielsweise dann nötig, wenn die Eigenschaft *TableName* geändert werden soll, um eine neue Tabelle zu erstellen, ohne jedoch den Namen derjenigen Tabelle zu verändern, die durch den ursprünglichen Wert von *TableName* repräsentiert wurde.

Zum Umbenennen einer Paradox- oder dBASE-Tabelle zur Laufzeit rufen Sie die Methode *RenameTable* der Tabellenkomponente auf. Beispiel:

```
Customer.RenameTable('CustInfo');
```

Tabellen erstellen

Neue Datenbanktabellen können während des Entwurfs und auch zur Laufzeit erzeugt werden. In der IDE dient dazu ein Menübefehl und zur Laufzeit die Methode *CreateTable* . In beiden Fällen sind keinerlei SQL-Kenntnisse erforderlich. Allerdings müssen Sie sich gut mit den Eigenschaften, Ereignissen und Methoden auskennen, die mit Datenmengenkomponenten in Zusammenhang stehen. Dies gilt besonders für *TTable.* Zur Definition der gewünschten Tabelle führen Sie die folgenden Schritte aus:

- Setzen Sie die Eigenschaft *DatabaseName* auf die Datenbank, in der die neue Tabelle enthalten sein wird.
- Setzen Sie die Eigenschaft *TableType* entsprechend dem gewünschten Typ der neuen Tabelle. Für Paradox, dBASE oder ASCII sind die Eigenschaftswerte *ttParadox*, *ttDBase* bzw. *ttASCII* vorgesehen. Bei allen anderen Tabellentypen setzen Sie *TableType* auf *ttDefault*.
- Setzen Sie die Eigenschaft *TableName* auf den Namen der neuen Tabelle. Falls die Eigenschaft *TableType* einen der Werte *ttParadox* und *ttDBase* besitzt, müssen Sie keine Dateinamenserweiterung angeben.
- Zur Beschreibung der einzelnen Felder werden jetzt noch Felddefinitionen benötigt. Zur Entwurfszeit doppelklicken Sie dazu auf die Eigenschaft *FieldDefs* im Objektinspektor, um den Kollektionseditor zu öffnen. Hier können Sie die Eigenschaften von Felddefinitionen hinzufügen, löschen oder ändern. Zur Laufzeit löschen Sie alle vorhandenen Felddefinitionen und fügen mit der Methode *AddFieldDef* die neuen ein. Dabei müssen jeweils die Eigenschaften des Objekts *TFieldDef* entsprechend den gewünschten Feldeigenschaften gesetzt werden.
- Optional fügen Sie Indexdefinitionen ein, welche die gewünschten Indizes der neuen Tabelle beschreiben. Zur Entwurfszeit doppelklicken Sie auf die Eigenschaft *IndexDefs* im Objektinspektor, um den Kollektionseditor zu öffnen und die Eigenschaften der Indexdefinitionen einzufügen, zu löschen oder zu ändern. Zur Laufzeit müssen erst alle vorhandenen Indexdefinitionen gelöscht und dann mit der Methode *AddIndexDef* einzeln hinzugefügt werden. Für jede neue Indexdefinition müssen die Eigenschaften des Objekts *TIndexDef* entsprechend den gewünschten Indexattributen gesetzt werden.

Hinweis Zur Entwurfszeit können die Feld- und Indexdefinitionen einer vorhandenen Tabelle in die Eigenschaften *FieldDefs* bzw. *IndexDefs* geladen werden. Setzen Sie die Eigenschaften *DatabaseName* und *TableName* so, daß sie die vorhandene Tabelle referenzieren. Klicken Sie mit der rechten Maustaste auf die Tabellenkomponente, und wählen Sie *Tabellendefinition aktualisieren*. Dadurch werden die Werte der Eigenschaften *FieldDefs* und *IndexDefs* automatisch an die Spezifikation der vorhandenen Tabelle angepaßt. Als nächstes stellen Sie die Eigenschaften *DatabaseName* und *TableName* so ein, daß Sie die zu erstellende Tabelle bezeichnen. Alle Rückfragen, ob die vorhandene Tabelle umbenannt werden soll, verneinen Sie. Wenn diese Definitionen zusammen mit der Tabellenkomponente gespeichert werden sollen, setzen Sie die Eigenschaft *StoreDefs* auf *True.*

Sobald alle Merkmale der Tabelle vollständig beschrieben sind, kann sie erzeugt werden. Zur Entwurfszeit klicken Sie dazu mit der rechten Maustaste auf die Tabellenkomponente und wählen den Befehl *Tabelle erstellen*. Zur Laufzeit generieren Sie die Tabelle mit der Methode *CreateTable*.

Achtung Wenn Sie eine neue Tabelle mit dem Namen einer bereits vorhandenen einrichten, wird die bestehende Tabelle durch die neue Tabelle überschrieben. Die alte Tabelle und deren Daten können nicht wiederhergestellt werden.

Im folgenden Beispiel wird eine neue Tabelle zur Laufzeit erzeugt und mit dem DBDEMOS-Alias verbunden. Vor der Erzeugung der neuen Tabelle wird überprüft, ob bereits eine Tabelle mit diesem Namen existiert:

```
var
   NewTable: TTable;
   NewIndexOptions: TIndexOptions;
   TableFound: Boolean;
begin
   NewTable := TTable.Create;
   NewIndexOptions := [ixPrimary, ixUnique];
   with NewTable do
   begin
    Active := False;
    DatabaseName := 'DBDEMOS';
    TableName := Edit1.Text;
    TableType := ttDefault;
    FieldDefs.Clear;
    FieldDefs.Add(Edit2.Text, ftInteger, 0, False);
    FieldDefs.Add(Edit3.Text, ftInteger, 0, False);
    IndexDefs.Clear;
    IndexDefs.Add('PrimaryIndex', Edit2.Text, NewIndexOptions);
   end;
   {Überprüfen, ob bereits eine Tabelle mit diesem Namen existiert}
   TableFound := FindTable(Edit1.Text); {Quelltext für FindTable wird nicht
                                         angezeigt}
   if TableFound = True then
    if MessageDlg('Überschreiben der vorhandenen Tabelle ' + Edit1.Text + '?',
mtConfirmation,
         mbYesNo, 0) = mrYes then
    TableFound := False;
   if not TableFound then
    CreateTable; { Tabelle erzeugen }
   end;
end;
```

Daten aus einer anderen Tabelle importieren

Mit der Methode *BatchMove* einer Tabellenkomponente können Daten aus einer anderen Tabelle importiert werden. Mit *BatchMove* können folgende Operationen durchgeführt werden:

- Datensätze aus einer anderen Tabelle kopieren
- Datensätze aktualisieren, die in dieser und in einer anderen Tabelle vorliegen
- Datensätze aus einer anderen Tabelle am Ende dieser Tabelle anfügen
- Datensätze aus dieser Tabelle löschen, die in einer anderen Tabelle vorliegen

BatchMove können zwei Parameter übergeben werden. Mit einem Parameter wird der Name der Tabelle übergeben, deren Daten importiert werden sollen. Mit dem anderen Parameter wird der Import-Modus festgelegt. Tabelle 20.4 führt die möglichen Werte für den Modus auf:

Tabelle 20.4 Import-Modi von BatchMove

Wert	Bedeutung
batAppend	Alle Datensätze der Quelltabelle werden am Ende dieser Tabelle angefügt.
batAppendUpdate	Alle Datensätze der Quelltabelle werden am Ende dieser Tabelle angefügt. Bereits vorhandene Datensätze werden mit den entsprechenden Datensätzen der Quelltabelle aktualisiert.
batCopy	Alle Datensätze der Quelltabelle werden in die Tabelle kopiert.
batDelete	Alle Datensätze, die auch in der Quelltabelle bestehen, werden gelöscht.
batUpdate	Datensätze in dieser Tabelle werden mit den entsprechenden Datensätzen der Quelltabelle aktualisiert.

Die folgende Anweisung aktualisiert Datensätze in der aktuellen Tabelle mit Datensätzen der Tabelle CUSTOMER:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

BatchMove gibt die Anzahl der erfolgreich importierten Datensätze zurück.

Achtung Beim Importieren von Datensätzen im Modus *batCopy* werden vorhandene Datensätze überschrieben. Sollen bestehende Datensätze erhalten bleiben, verwenden Sie statt dessen *batAppend*.

TBatchMove-Komponenten stellen nicht nur die Funktion *BatchMove* für Anwendungen bereit. Werden größere Datenmengen zwischen Tabellen verschoben, verwenden Sie statt der Methode *BatchMove* der Tabelle eine *TBatchMove*-Komponente. Weitere Informationen zu *TBatchMove*-Komponenten finden Sie unter »TBatchMove verwenden«.

TBatchMove verwenden

TBatchMove kapselt BDE-Features, mit deren Hilfe Sie eine Datenmenge kopieren, Datensätze einer Datenmenge an eine andere anhängen, mit Datensätzen einer anderen Datenmenge aktualisieren und übereinstimmende Datensätze löschen können. Üblicherweise wird *TBatchMove* für folgende Aufgaben eingesetzt:

- Herunterladen von Server-Daten in eine lokale Datenquelle für Analysen oder andere Operationen.
- Verschieben einer Desktop-Datenbank in Tabellen auf einen Remote-Server als Teil einer Upsizing-Operation.

Eine Batch-Move-Komponente kann im Ziel Tabellen erzeugen, die den Quelltabellen entsprechen, wobei Spaltennamen und Datentypen in geeigneter Weise zugeordnet werden.

Batch-Move-Komponenten erzeugen

So erzeugen Sie eine Batch-Move-Komponente:

1. Plazieren Sie für die Datenmenge, aus der Sie Datensätze holen wollen (diese Datenmenge wird als *Quell-Datenmenge* bezeichnet), eine Tabellen- oder Abfragekomponente in einem Formular oder Datenmodul.
2. Plazieren Sie die Datenmengen-Komponente, in die Sie Datensätze verschieben wollen (die *Ziel-Datenmenge*), in einem Formular oder Datenmodul.
3. Plazieren Sie eine *TBatchMove*-Komponente aus dem Register *Datenzugriff* der Komponentenpalette in einem Datenmodul oder Formular, und weisen Sie ihrer Eigenschaft *Name* einen eindeutigen, der Verwendung entsprechenden Wert zu.
4. Weisen Sie der Eigenschaft *Source* der Batch-Move-Komponente den Namen der Tabelle zu, aus der Daten kopiert, angehängt oder aktualisiert werden sollen. Sie können die Tabelle aus der Dropdown-Liste der verfügbaren Datenmengen-Komponenten auswählen.
5. Weisen Sie der Eigenschaft *Destination* die Datenmenge zu, die erzeugt, erweitert oder aktualisiert werden soll. Sie können die Zieltabelle aus der Dropdown-Liste der verfügbaren Datenmengen-Komponenten auswählen oder als Ziel eine neue Tabellenkomponente hinzufügen.

   Wenn bei der Operation Daten angehängt, aktualisiert oder gelöscht werden sollen, muß *Destination* den Namen einer existierenden Tabelle enthalten.

   Wenn bei der Operation eine Tabelle kopiert wird und *Destination* den Namen einer existierenden Tabelle enthält, werden bei der Batch-Move-Operation alle bestehenden Daten in der Zieltabelle gelöscht.

   Wenn Sie durch Kopieren einer existierenden Tabelle eine vollkommen neue Tabelle erzeugen, erhält die neue Tabelle den Namen, der in der Eigenschaft *Name* der Tabellenkomponente angegeben ist, in die Sie kopieren. Die Struktur des neu-

en Tabellentyps entspricht dem Server, der in der Eigenschaft *DatabaseName* angegeben ist.

6 Legen Sie mit der Eigenschaft *Mode* die Art der auszuführenden Operation fest. Zulässige Werte sind *batAppend* (Voreinstellung), *batUpdate*, *batAppendUpdate*, *batCopy* und *batDelete*. Weitere Informationen zu diesen Modi finden Sie unter »Batch-Move-Modi« auf Seite 20-23.

7 Weisen Sie der Eigenschaft *Transliterate* einen Wert zu (optional). Wenn diese Eigenschaft *True* ist (Voreinstellung), werden, falls erforderlich, Zeichen vom Zeichensatz der Quell-Datenmenge in den Zeichensatz der Ziel-Datenmenge umgewandelt.

8 Legen Sie mit der Eigenschaft *Mappings* die gewünschten Spaltenzuordnungen fest (optional). Diese Eigenschaft braucht nicht gesetzt zu werden, wenn die Spalten bei der Batch-Move-Operation aufgrund ihrer Position in der Quelltabelle und der Zieltabelle zugeordnet werden sollen. Weitere Informationen über die Zuordnung von Spalten finden Sie unter »Datentypen zuordnen« auf Seite 20-25.

9 Weisen Sie den Eigenschaften *ChangedTableName*, *KeyViolTableName* und *ProblemTableName* Werte zu (optional). In der mit *ProblemTableName* festgelegten Tabelle werden Datensätze gespeichert, die während einer Batch-Move-Operation Probleme verursachen. Wenn Sie mit der Operation eine Paradox-Tabelle aktualisieren, können Sie in der mit *KeyViolTableName* festgelegten Tabelle Schlüsselverletzungen festhalten. Die in *ChangedTableName* angegebene Tabelle enthält alle Datensätze, die sich in der Zieltabelle im Rahmen der Batch-Move-Operation geändert haben. Wenn Sie die genannten Eigenschaften nicht setzen, werden diese Fehlertabellen weder erzeugt noch verwendet. Weitere Informationen zur Behandlung von Batch-Move-Fehlern finden Sie unter »Batch-Move-Fehler« auf Seite 20-26.

Batch-Move-Modi

Die Eigenschaft *Mode* legt fest, welche Operation die Batch-Move-Komponente ausführt:

Tabelle 20.5 Batch-Move-Modi

Eigenschaft	Verwendungszweck
batAppend	Hängt Datensätze an die Zieltabelle an.
batUpdate	Aktualisiert Datensätze in der Zieltabelle mit entsprechenden Datensätzen der Quelltabelle. Die Aktualisierung erfolgt auf der Basis des aktuellen Index der Zieltabelle.
batAppendUpdate	Falls ein entsprechender Datensatz in der Zieltabelle existiert, wird er aktualisiert. Andernfalls werden die Datensätze an die Zieltabelle angehängt.
batCopy	Erzeugt die Zieltabelle auf Grundlage der Struktur der Quelltabelle. Wenn die Zieltabelle bereits existiert, wird sie gelöscht und neu erzeugt.
batDelete	Löscht Datensätze der Zieltabelle, die mit Datensätzen der Quelltabelle übereinstimmen.

Anhängen

Wenn Daten angehängt werden sollen, muß die Zieldatenmenge bereits existieren. Beim Anhängen der Datensätze konvertiert die BDE Daten gegebenenfalls in für die Zieldatenmenge geeignete Datentypen und Größen. Wenn eine Konvertierung nicht möglich ist, wird eine Exception ausgelöst, und die Daten werden nicht angehängt.

Aktualisieren

Bei der Aktualisierung von Daten muß die Zieldatenmenge bereits existieren und indiziert sein, damit die Datensätze zugeordnet werden können. Wenn die Zuordnung aufgrund der primären Indexfelder hergestellt wird, werden diejenigen Datensätze der Zieltabelle, deren Indexfelder mit Indexfeldern der Quelldatenmenge übereinstimmen, mit den Quelldaten überschrieben. Dabei konvertiert die BDE Daten gegebenenfalls in für die Zieldatenmenge geeignete Datentypen und Größen.

Anhängen und Aktualisieren

Zum Anhängen und Aktualisieren von Daten muß die Zieldatenmenge bereits existieren und indiziert sein, damit die Datensätze zugeordnet werden können. Wenn die Zuordnung über die primären Indexfelder hergestellt wird, werden diejenigen Datensätze der Zieltabelle, deren Indexfelder mit Indexfeldern der Quelldatenmenge übereinstimmen, mit den Quelldaten überschrieben. Andernfalls werden die Daten der Quelldatenmenge an die Zieldatenmenge angehängt. Dabei konvertiert die BDE Daten gegebenenfalls in für die Zieldatenmenge geeignete Datentypen und Größen.

Kopieren

Wenn eine Quelldatenmenge kopiert werden soll, sollte die Zieldatenmenge noch nicht existieren. Andernfalls wird bei der Batch-Move-Operation die existierende Zieldatenmenge mit der kopierten Quelldatenmenge überschrieben.

Gehören die Quelldatenmenge und die Zieldatenmenge zu verschiedenen Plattformen, z.B. Paradox und InterBase, erstellt die BDE eine Zieldatenmenge mit einer Struktur, die der Struktur der Quelldatenmenge so nahe wie möglich kommt, und führt automatisch die erforderliche Konvertierung des Datentyps und der Größe durch.

Hinweis Andere Metadateien, z.B. Indizes, Beschränkungen und Stored Procedures werden von *TBatchMove* nicht kopiert. Diese Metadaten-Objekte müssen Sie auf dem Datenbank-Server oder durch den SQL-Explorer gegebenenfalls neu erzeugen.

Löschen

Zum Löschen von Datensätzen in der Zieldatenmenge muß diese bereits existieren und indiziert sein, damit die Datensätze zugeordnet werden können. Wenn die Zuordnung aufgrund der primären Indexfelder hergestellt wird, werden diejenigen Datensätze der Zieltabelle, deren Indexfelder mit Indexfeldern der Quelldatenmenge übereinstimmen, gelöscht.

Datentypen zuordnen

Im Modus *batAppend* erzeugt eine Batch-Move-Komponente die Zieltabelle auf Grundlage der Datentypen der Quelltabelle. Spalten und Datentypen werden anhand ihrer Position in der Quelltabelle und der Zieltabelle zugeordnet. Das heißt, die erste Spalte der Quelle wird der ersten Spalte des Ziels zugeordnet usw.

Mit der Eigenschaft *Mappings* kann die Voreinstellung für die Zuordnung der Spalten überschrieben werden. *Mappings* ist eine Liste mit Spaltenzuordnungen (eine Zuordnung pro Zeile). Für die Erstellung einer solchen Liste gibt es zwei Möglichkeiten. Um eine Spalte der Quelltabelle einer gleichnamigen Spalte der Zieltabelle zuzuordnen, können Sie eine einfache Liste benutzen, in der der entsprechende Spaltenname angegeben ist. Die folgende Zuordnung hat beispielsweise zur Folge, daß eine Spalte der Quelltabelle namens *ColName* der gleichnamigen Spalte der Zieltabelle zugeordnet wird:

```
ColName
```

Um die Spalte *SourceColName* in der Quelltabelle der Spalte *DestColName* in der Zieltabelle zuzuordnen, verwenden Sie folgende Syntax:

```
DestColName = SourceColName
```

Stimmen die Datentypen von Quell- und Zielspalte nicht überein, versucht die Batch-Move-Operation, die Unterschiede so gut wie möglich zu korrigieren. Dabei werden notfalls Daten abgeschnitten und bis zu einem bestimmten Grad Konvertierungsversuche durchgeführt. Bei dem Versuch, eine CHAR(10)-Spalte einer CHAR(5)-Spalte zuzuordnen, werden beispielsweise die letzten fünf Zeichen der Quellspalte abgeschnitten.

Hier ein Beispiel für eine Konvertierung: Wird eine Quellspalte mit einem alphanumerischen Datentyp einer Zielspalte vom Typ Integer zugeordnet, konvertiert die Batch-Move-Operation den Zeichenwert »5« in den entsprechenden Integer-Wert. Werte, die nicht konvertiert werden können, lösen einen Fehler aus. Weitere Informationen zu Fehlern finden Sie unter »Batch-Move-Fehler« auf Seite 20-26.

Wenn Daten zwischen Tabellen unterschiedlichen Typs bewegt werden, wandelt eine Batch-Move-Komponente die Datentypen auf Grundlage der Servertypen der Datenmenge um. Die aktuellen Zuordnungen zwischen verschiedenen Servertypen finden Sie in der BDE-Hilfe.

Hinweis Damit Daten im Rahmen einer Batch-Move-Operation an eine SQL-Server-Datenbank übermittelt werden können, muß der Datenbankserver und eine Delphi-Version mit der entsprechenden SQL Links-Software installiert sein. Wenn die richtigen ODBC-Treiber von Fremdherstellern installiert sind, können Sie auch ODBC verwenden.

Batch-Move-Operationen ausführen

Die Methode *Execute* führt zur Laufzeit eine zuvor festgelegte Batch-Operation durch. Wenn beispielsweise eine Batch-Move-Komponente den Namen *BatchMoveAdd* hat, wird die mit der folgenden Anweisung ausgeführt:

```
BatchMoveAdd.Execute;
```

Sie können eine Batch-Move-Operation auch zur Entwurfszeit ausführen. Klicken Sie dazu mit der rechten Maustaste auf die Batch-Move-Komponente, und wählen Sie *Ausführen* im lokalen Menü.

Die Eigenschaft *MovedCount* enthält die Anzahl der Datensätze, die im Rahmen einer Batch-Move-Operation bewegt wurden.

Mit der Eigenschaft *RecordCount* können Sie festlegen, wie viele Datensätze maximal bewegt werden sollen. Hat *RecordCount* den Wert 0, erfaßt die Operation alle Datensätze, beginnend mit dem ersten Datensatz der Quelldatenmenge. Ist *RecordCount* eine positive Zahl, wird maximal diese Anzahl von Datensätzen kopiert, beginnend mit dem aktuellen Datensatz der Quelldatenmenge. Ist *RecordCount* höher als die Anzahl der Datensätze zwischen dem aktuellen und dem letzten Datensatz der Quelldatenmenge, wird die Operation beendet, sobald das Ende der Quelldatenmenge erreicht ist. Sie können *MoveCount* abfragen, um festzustellen, wie viele Datensätze tatsächlich übertragen wurden.

Batch-Move-Fehler

Bei einer Batch-Move-Operation können zwei Arten von Fehlern auftreten: Fehler bei der Konvertierung von Daten und Integritätsverletzungen. *TBatchMove* verfügt über eine Reihe von Eigenschaften, die die Reaktion auf solche Fehler steuern.

Die Eigenschaft *AbortOnProblem* legt fest, ob eine Operation abgebrochen wird, wenn ein Fehler bei der Konvertierung von Datentypen auftritt. Wenn *AbortOnProblem* den Wert *True* hat, wird die Batch-Move-Operation abgebrochen, sobald ein Fehler auftritt. Hat sie den Wert *False*, wird die Operation fortgesetzt. Durch Überprüfen der Tabelle, die in der Eigenschaft *ProblemTableName* angegeben ist, kann festgestellt werden, welche Datensätze Fehler ausgelöst haben.

Die Eigenschaft *AbortOnKeyViol* legt fest, ob die Operation abgebrochen wird, wenn eine Paradox-Schlüsselverletzung auftritt.

Die Eigenschaft *ProblemCount* gibt an, wie viele Datensätze in der Zieltabelle nicht ohne Datenverlust bearbeitet werden konnten. Wenn *AbortOnProblem* den Wert *True* hat, steht *ProblemCount* immer auf 1, da die Operation beim ersten auftretenden Problem abgebrochen wird.

Mit Hilfe der folgenden Eigenschaften können Batch-Move-Komponenten zusätzliche Tabellen erzeugen, in denen die Batch-Move-Operation protokolliert wird:

- Wenn die Eigenschaft *ChangedTableName* einen Wert enthält, wird eine lokale Paradox-Tabelle erzeugt, die eine Kopie aller Datensätze der Zieltabelle enthält, die im Rahmen einer Aktualisierungs- oder Löschoperation verändert wurden.

- Wenn die Eigenschaft *KeyViolTableName* einen Wert enthält, wird eine lokale Paradox-Tabelle erzeugt, die alle Datensätze der Quelltabelle enthält, die bei einer Paradox-Tabelle eine Schlüsselverletzung hervorgerufen haben. Hat *AbortOnKeyViol* den Wert *True*, enthält diese Tabelle höchstens einen Eintrag, da die Operation beim ersten auftretenden Fehler abgebrochen wird.
- Wenn die Eigenschaft *ProblemTableName* einen Wert enthält, wird eine lokale Paradox-Tabelle erzeugt. Diese Tabelle enthält alle Datensätze, die wegen eines Fehlers bei der Konvertierung von Datentypen nicht in der Zieltabelle gespeichert werden konnten. Die Tabelle könnte z.B. die Datensätze der Quelltabelle enthalten, die beim Speichern in der Zieltabelle abgeschnitten wurden. Hat *AbortOnProblem* den Wert *True*, enthält diese Tabelle höchstens einen Datensatz, da die Operation beim ersten auftretenden Problem abgebrochen wird.

Hinweis Wenn *ProblemTableName* kein Wert zugewiesen wurde, werden die Daten abgeschnitten und in der Zieltabelle gespeichert.

Mit einer Datenbanktabelle verknüpfte Tabellenkomponenten synchronisieren

Sind mehrere Tabellenkomponenten durch ihre Eigenschaften *DatabaseName* und *TableName* mit derselben Datenbanktabelle verknüpft, ohne eine Datenquellenkomponente gemeinsam zu nutzen, verwendet jede Tabellenkomponente einen spezifischen Teil der Daten. Die jeweils aktuellen Datensätze werden in den Tabellenkomponenten unabhängig voneinander verwaltet. Sobald Benutzer mit den Tabellenkomponenten auf die Datensätze zugreifen, unterscheiden sich die aktuellen Datensätze der verschiedenen Komponenten.

Sie können mit der Methode *GotoCurrent* veranlassen, daß der aktuelle Datensatz in beiden Tabellenkomponenten identisch ist. *GotoCurrent* aktiviert den Datensatz in der zugehörigen Tabellenkomponente, die in der anderen Tabellenkomponente der aktuelle Datensatz ist. Die folgende Anweisung aktiviert beispielsweise in *CustomerTableOne* den aktuellen Datensatz von *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

Tip Müssen in einer Anwendung Tabellenkomponenten auf diese Weise synchronisiert werden, fügen Sie die Komponenten in ein Datenmodul und anschließend den Header für das Datenmodul in jede Unit ein, die auf die Tabellen zugreift.

Sollen Tabellenkomponenten in separaten Formularen synchronisiert werden, müssen Sie die Header-Datei des einen Formulars in den Quelltext des anderen Formulars einbinden und mindestens einen Tabellennamen mit seinem Formularnamen qualifizieren.

Ein Beispiel:

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

Haupt/Detail-Formulare erstellen

Mit den Eigenschaften *MasterSource* and *MasterFields* einer Tabellenkomponente können Sie zwischen zwei Tabellen eine 1-N-Beziehung einrichten.

Die Eigenschaft *MasterSource* gibt dabei die Datenquelle an, aus der die Daten für die Haupttabelle abgerufen werden. Verknüpfen Sie beispielsweise zwei Tabellen in einer Haupt / Detail-Beziehung, kann die Detailtabelle die in der Haupttabelle auftretenden Ereignisse überwachen, indem die Datenquellenkomponente der Haupttabelle in der Eigenschaft *MasterSource* angegeben wird.

Die Eigenschaft *MasterFields* bezeichnet die in beiden Tabellen vorhandenen Spalten, mit denen die Verknüpfung hergestellt wird. Sollen Tabellen über mehrere Spaltennamen verknüpft werden, müssen Sie eine durch Semikolons getrennte Liste verwenden:

```
Table1.MasterFields := 'OrderNo;ItemNo';
```

Mit dem Feldverbindungs-Designer können Sie Verknüpfungen zwischen zwei Tabellen einrichten. Weitere Informationen zum Feldverbindungs-Designer finden Sie im Benutzerhandbuch.

Beispiel für ein Haupt/Detail-Formular

Im folgenden Beispiel wird ein einfaches Formular erstellt, in dem der Benutzer die verschiedenen Kundendatensätze und die zugehörigen Aufträge anzeigen kann. Die Haupttabelle ist *CustomersTable,* und die Detailtabelle heißt *OrdersTable.*

1. Fügen Sie zwei *TTable*- und zwei *TDataSource*-Komponenten in ein Datenmodul ein.
2. Legen Sie die Eigenschaften der ersten *TTable*-Komponente wie folgt fest:
   - *DatabaseName*: DBDEMOS
   - *TableName*: CUSTOMER
   - *Name*: CustomersTable
3. Legen Sie die Eigenschaften der zweiten *TTable*-Komponente wie folgt fest:
   - *DatabaseName*: DBDEMOS
   - *TableName*: ORDERS
   - *Name*: OrdersTable
4. Legen Sie die Eigenschaften der ersten *TDataSource*-Komponente wie folgt fest:
   - *Name*: CustSource
   - *DataSet*: CustomersTable
5. Legen Sie die Eigenschaften der zweiten *TDataSource*-Komponente wie folgt fest:
   - *Name*: OrdersSource

- *DataSet*: OrdersTable

6 Fügen Sie zwei *TDBGrid*-Komponenten in ein Formular ein.

7 Wählen Sie *Datei* / *Unit Hdr. einfügen*, damit das Datenmodul vom Formular verwendet wird.

8 Setzen Sie die Eigenschaft *DataSource* der ersten Gitterkomponente auf *DataModule2->CustSource* und *DataSource* der zweiten Komponente auf *DataModule2->OrdersSource*.

9 Setzen Sie die Eigenschaft *MasterSource* von *OrdersTable* auf *CustSource*. Dadurch wird die Tabelle CUSTOMER (Haupttabelle) mit der Tabelle ORDERS (Detailtabelle) verknüpft.

10 Doppelklicken Sie auf die Wertespalte der Eigenschaft *MasterFields* im Objektinspektor, um den Feldverbindungs-Designer zu öffnen, und geben Sie die folgenden Eigenschaften an:

- Wählen Sie im Kombinationsfeld *Verfügbare Indexe* CUSTNO, um die beiden Tabellen über das Feld CUSTNO zu verknüpfen.
- Wählen Sie CUSTNO in den beiden Listenfeldern *Detailfelder* und *Hauptfelder* aus.
- Um die Verbindungsbedingung hinzuzufügen, klicken Sie den Schalter *Hinzufügen* an. Das Listenfeld *Verbundene Felder* enthält dann den Eintrag »CustNo -> CustNo«.
- Wählen Sie *OK*, um die Einstellungen zu bestätigen und den Feldverbindungs-Designer zu schließen.

11 Setzen Sie die Eigenschaft *Active* der Tabellenkomponenten *CustomersTable* und *OrdersTable* auf *True*, damit die Daten in den Gitterkomponenten im Formular angezeigt werden.

12 Compilieren und starten Sie die Anwendung.

Beim Ausführen der Anwendung sind die beiden Tabellen miteinander verknüpft. Sobald Sie zu einem anderen Datensatz in der Tabelle CUSTOMER wechseln, werden nur die Datensätze in ORDERS angezeigt, die zum aktuellen Kunden gehören.

Verschachtelte Tabellen

Eine verschachtelte Tabellenkomponente bietet Zugriff auf die Daten einer verschachtelten Datenmenge einer Tabelle. Die Eigenschaft *NestedDataSet* eines persistenten verschachtelten Datenmengenfeldes enthält eine Referenz auf die verschachtelte Datenmenge. Da *TNestedDataSet* ein Nachkomme von *TBDEDataSet* ist, erbt eine verschachtelte Tabelle die BDE-Funktionalität und verwendet die BDE für den Zugriff auf Daten in verschachtelten Tabellen. Eine verschachtelte Tabelle bietet ähnliche Funktionalität wie eine Tabellenkomponente. Die Daten, auf die sie zugreift, befinden sich jedoch in einer verschachtelten Tabelle.

Verschachtelte Tabellenkomponenten einrichten

In den folgenden Schritten wird die Einrichtung von verschachtelten Tabellenkomponenten zur Entwurfszeit beschrieben. Eine Tabellenkomponenten oder aktualisierbare Abfrage muß bereits zur Verfügung stehen, um auf eine Datenmenge zuzugreifen, die ein Datenmengen- oder Referenzfeld enthält. Außerdem muß ein persistentes Feld für *TDataSetField* oder *TReferenceField* existieren. Siehe »Datenmengenfelder« auf Seite 19-30.

So verwenden Sie eine verschachtelte Tabellenkomponente:

1. Plazieren Sie eine verschachtelte Tabellenkomponente aus der Registerkarte *Datenzugriff* der Komponentenpalette im Datenmodul bzw. Formular, und setzen Sie ihre Eigenschaft *Name* auf einen in der Anwendung eindeutigen Wert.
2. Setzen Sie die Eigenschaft *DataSetField* der Komponente auf den Namen des persistenten Datenmengenfeldes bzw. des Referenzfeldes, auf das zugegriffen werden soll. Sie können die Felder aus der Dropdown-Liste auswählen.
3. Fügen Sie eine Datenquellenkomponente in das Datenmodul oder Formular ein, und setzen Sie ihre Eigenschaft *DataSet* auf den Namen der verschachtelten Tabellenkomponente. Die Datenquellenkomponente wird verwendet, um eine Ereignismenge aus der Tabelle an die datensensitiven Steuerelemente zur Anzeige zu übergeben.

Kapitel

21

Abfragen

Dieses Kapitel befaßt sich mit der Datenmengenkomponente *TQuery*, die es Ihnen ermöglicht, mit SQL-Anweisungen auf Daten zuzugreifen. Die Erläuterungen basieren auf der allgemeinen Beschreibung von Datenmengen und Datenquellen in Kapitel 18, »Datenmengen«.

Eine Abfragekomponente kapselt eine SQL-Anweisung, die in einer Client-Anwendung für das Lesen, Einfügen, Aktualisieren und Löschen von Daten aus einer oder mehreren Datenbanktabellen verantwortlich ist. SQL ist eine standardisierte Abfragesprache für relationale Datenbanken, die von den meisten serverbasierten Remote-Datenbanken eingesetzt wird, beispielsweise von Sybase, Oracle, InterBase und Microsoft SQL Server. Die Abfragekomponenten können mit Remote-Datenbankservern (sofern Ihre Delphi-Version SQL Links enthält), Paradox, dBASE, FoxPro und Access sowie mit OBDC-kompatiblen Datenbanken verwendet werden.

Abfragen effektiv einsetzen

Damit Sie die Abfragekomponente effektiv nutzen können, benötigen Sie Kenntnisse über

- SQL und die SQL-Implementierung Ihres Servers, einschließlich der Beschränkungen und Erweiterungen des Standards SQL-92,
- die Borland Database Engine (BDE).

Wenn Sie bereits Desktop-Datenbankanwendungen entwickelt haben und sich nun Serveranwendungen zuwenden wollen, finden Sie unter »Abfragen für Desktop-Entwickler« auf Seite 21-2 weitere Informationen. Wenn Sie mit SQL noch keine Erfahrung haben, sollten Sie eines der vielen guten Bücher über SQL lesen. Eines der besten ist *Understanding the New SQL: A Complete Guide* von Jim Melton und Alan R. Simpson, Morgan Kaufmann Publishers.

Als erfahrener Entwickler von Datenbank-Servern kennen Sie zwar SQL und Ihren Server, aber möglicherweise nicht die BDE und den Aufbau von Delphi-Clients. Im

Abschnitt »Abfragen für Server-Entwickler« auf Seite 21-3 finden Sie eine Einführung zum Thema Abfragen und BDE. Danach können Sie sich dem Rest dieses Kapitels zuwenden.

Abfragen für Desktop-Entwickler

Als Desktop-Entwickler sind Sie bereits mit den Zusammenhängen zwischen Tabellen, Datensätzen und Feldern in Delphi und der BDE vertraut. Sie wissen, wie einfach mit einer *TTable*-Komponente der Zugriff auf alle Felder aller Datensätze einer Datenmenge vor sich geht. Außerdem wissen Sie, daß Sie über die Eigenschaft *DataSet* die Datenbanktabelle angeben können, auf die zugegriffen werden soll.

Unter Umständen haben Sie schon die Bereichs- und Filtereigenschaften und die Methoden einer *TTable*-Komponente eingesetzt, um die Anzahl der zu einem bestimmten Zeitpunkt verfügbaren Datensätze einzuschränken. Durch die Verwendung von Bereichen können Sie den Datenzugriff vorübergehend auf einen Block mit fortlaufend indizierten Datensätzen einschränken, die die vorher festgelegten Bedingungen erfüllen. Sie könnten beispielsweise die Datensätze aller Angestellten abrufen, deren Nachnamen größer oder gleich »Kaiser« und kleiner oder gleich »Schmitt« sind. Wird ein temporärer Filter gesetzt, läßt sich der Datenzugriff auf eine Menge von Datensätzen einschränken, die normalerweise nicht unmittelbar aufeinanderfolgen, aber den Filterkriterien entsprechen. Ein Beispiel hierfür wären die Datensätze aller Kunden, deren Anschrift in einem bestimmten Vertriebsgebiet liegt.

Eine Abfrage verhält sich in vielerlei Hinsicht wie ein Tabellenfilter, mit der Ausnahme, daß sie die Eigenschaft *SQL* (und manchmal die Eigenschaft *Params*) der Abfragekomponente verwendet. Mit dieser Eigenschaft werden die Datensätze festgelegt, die gelesen, eingefügt, gelöscht oder aktualisiert werden sollen. In gewissem Sinn ist eine Abfrage sogar leistungsfähiger als ein Filter, weil sich mit ihr folgende Aufgaben ausführen lassen:

- Sie können auf mehrere Tabellen gleichzeitig zugreifen (in SQL ein sogenannter »Join«).
- Sie können auf eine Untermenge von Zeilen *und* Spalten in der bzw. den zugrundeliegenden Tabellen zugreifen, d.h., es müssen nicht immer alle Zeilen und Spalten abgerufen werden. Durch dieses Vorgehen erhöhen sich Ausführungsgeschwindigkeit und Sicherheit. Außerdem wird der Speicher nicht mit unnötigen Daten belegt und der Zugriff auf Felder verhindert, die der Benutzer nicht anzeigen oder ändern darf.

Abfragen können »wörtlich« vorliegen oder austauschbare Parameter enthalten. Abfragen, die Parameter verwenden, heißen *parametrisierte Abfragen*. Wenn Sie mit Abfragen dieser Art arbeiten, müssen die Werte, die den Parametern aktuell zugewiesen werden sollen, von der BDE vor der Ausführung in die Abfrage eingefügt werden. Der Einsatz von parametrisierten Abfragen bringt eine hohe Flexibilität mit sich, weil Sie die Benutzeransicht der Daten und den Zugriff darauf zur Laufzeit schnell ändern können, ohne die Eigenschaft *SQL* ändern zu müssen.

Am häufigsten werden Abfragen verwendet, um die Daten auszuwählen, die der Benutzer in der Anwendung sehen soll. Dies entspricht dem Einsatz einer Tabellenkom-

ponente. Mit Abfragen lassen sich aber nicht nur Daten für die Anzeige abrufen, es können auch Aktualisierungs-, Einfüge- und Löschoperationen ausgeführt werden. Wenn Sie eine Abfrage für derartige Operationen verwenden, werden normalerweise keine Datensätze für die Anzeige zurückgegeben. In diesem Punkt unterschiedet sich eine Abfrage von einer Tabelle.

Im Abschnitt »Die Eigenschaft SQL setzen« auf Seite 21-6 erfahren Sie mehr über die Formulierung einer SQL-Anweisung für die Eigenschaft SQL. Detaillierte Informationen über Parameter in SQL-Anweisungen enthält der Abschnitt »Parameter setzen« auf Seite 21-9. Wie Abfragen ausgeführt werden, erfahren Sie im Abschnitt »Abfragen ausführen« auf Seite 21-13.

Abfragen für Server-Entwickler

Als Server-Entwickler sind Sie bereits mit SQL und mit den Funktionen Ihres Datenbank-Servers vertraut. Für Sie stellt sich eine Abfrage als SQL-Anweisung dar, die dem Zugriff auf Daten dient. Sie wissen, wie diese Anweisung eingesetzt und bearbeitet wird und wie optionale Parameter eingefügt werden können.

Die SQL-Anweisung und ihre Parameter sind die wichtigsten Bestandteile einer Abfragekomponente. Mit Hilfe der Eigenschaft *SQL* der Abfragekomponente wird die SQL-Anweisung bereitgestellt, über die der Zugriff auf die Daten erfolgt. Die Eigenschaft *Params* der Komponente ist ein optionales Array von Parametern, die in die Abfrage eingebunden werden können. Eine Abfragekomponente ist jedoch mehr als eine SQL-Anweisung mit Parametern. Sie stellt zugleich die Schnittstelle zwischen der Client-Anwendung und der BDE dar.

Eine Client-Anwendung verwendet die Eigenschaften und Methoden einer Abfragekomponente, um eine SQL-Anweisung zu bearbeiten, die abzufragende Datenbank festzulegen, Abfragen mit Parametern vorzubereiten und am Ende die Abfrage auszuführen. Die Methoden einer Abfragekomponente rufen die BDE auf, die ihrerseits Ihre Abfrageanforderungen bearbeitet und mit dem Datenbank-Server normalerweise über einen SQL-Links-Treiber kommuniziert. Der Server gibt im Erfolgsfall eine Ergebnismenge an die BDE zurück. Die BDE gibt diese Menge ihrerseits über die Abfragekomponente an die Anwendung weiter.

Wenn Sie mit einer Abfragekomponente arbeiten, erscheint Ihnen möglicherweise die Terminologie, die zur Beschreibung von BDE-Funktionen verwendet wird, etwas verwirrend. Die verwendeten Begriffe haben für SQL-Programmierer normalerweise eine andere Bedeutung. Beispielsweise dient in der BDE der Begriff »Alias« zur Bezeichnung eines Kurznamens für den Pfad des Datenbank-Servers. Der BDE-Alias wird in der Konfigurationsdatei gespeichert und der Eigenschaft *DatabaseName* der Abfragekomponente zugewiesen. (Sie können Tabellen-Aliase oder Tabellenkorrelationsnamen aber auch in SQL-Anweisungen verwenden.)

Ähnliches gilt für die BDE-Hilfe in \BORLAND\GEMEINSAME DATEIEN\BDE\BDE32.HLP. Dort werden Abfragen mit Parametern häufig als »parametrisierte Abfragen« bezeichnet. Sie werden bei diesem Begriff spontan eher an SQL-Anweisungen mit gebundenen Variablen oder Parameter-Bindungen denken.

Hinweis In diesem Kapitel wird die BDE-Terminologie verwendet, da Ihnen diese in allen Borland-Handbüchern begegnet. Stellen, an denen es zu Mißverständnissen kommen könnte, sind mit entsprechenden Anmerkungen versehen.

Im Abschnitt »Die Eigenschaft SQL setzen« auf Seite 21-6 erfahren Sie mehr über die Formulierung einer SQL-Anweisung für die Eigenschaft *SQL*. Wenn Sie detaillierte Informationen über die Verwendung von Parametern in SQL-Anweisungen benötigen, lesen Sie den Abschnitt »Parameter setzen« auf Seite 21-9. Die Vorbereitung einer Abfrage wird im Abschnitt »Abfragen vorbereiten« auf Seite 21-15, die Ausführung von Abfragen im Abschnitt »Abfragen ausführen« auf Seite 21-13 beschrieben.

Datenbanken, auf die mit einer Abfragekomponente zugegriffen werden kann

Eine *TQuery*-Komponente kann auf Daten in folgenden Datenbanken zugreifen:

- Paradox- oder dBASE-Tabellen, die Local SQL verwenden (Bestandteil der BDE). Local SQL unterstützt den größten Teil der DML und weite Bereiche der DDL-Syntax, um mit diesen Tabellentypen arbeiten zu können. Details zur unterstützten SQL-Syntax finden Sie in der Hilfedatei LOCALSQL.HLP.
- Lokale InterBase Server-Datenbanken, die die InterBase-Engine verwenden. Informationen über die SQL-Syntaxunterstützung und den Standard SQL-92 für InterBase finden Sie in der InterBase-Sprachreferenz.
- Datenbanken auf Remote-Datenbankservern wie Oracle, Sybase, MS SQL Server, Informix, DB2 und InterBase (je nachdem, über welche Delphi-Version Sie verfügen). Damit der Zugriff auf den Remote-Server möglich ist, muß der entsprechende SQL Links-Treiber und die entsprechende Client-Software (vom Hersteller) für den Datenbank-Server installiert sein. Jede SQL-Standardsyntax, die vom benutzten Server unterstützt wird, ist zulässig. Informationen über die SQL-Syntax, Einschränkungen und Erweiterungen finden Sie in der Dokumentation Ihres Servers.

Delphi unterstützt außerdem heterogene Abfragen über mehrere Server- oder Tabellentypen (z.B. über Daten aus einer Oracle- und einer Paradox-Tabelle). Wenn Sie eine heterogene Abfrage erstellen, verwendet die BDE zur Bearbeitung dieser Abfrage Local SQL. Weitere Informationen finden Sie im Abschnitt »Heterogene Abfragen erstellen« auf Seite 21-16.

Abfragekomponenten im Überblick

Zum Einsatz einer Abfragekomponente in einer Anwendung führen Sie zur Entwurfszeit die folgenden Schritte durch:

1. Fügen Sie eine Abfragekomponente aus der Registerkarte *Datenzugriff* der Komponentenpalette in ein Datenmodul ein, und setzen Sie ihre Eigenschaft *Name*.
2. Weisen Sie der Eigenschaft *DatabaseName* der Komponente den Namen der abzufragenden Datenbank zu. *DatabaseName* kann ein BDE-Alias oder ein expliziter

Verzeichnispfad (für lokale Tabellen) oder der Wert der Eigenschaft *DatabaseName* einer *TDatabase*-Komponente in der Anwendung sein.

3. Legen Sie in der Eigenschaft *SQL* dieser Komponente eine SQL-Anweisung und optional in der Eigenschaft *Params* die Parameter für die Anweisung fest. Weitere Informationen finden Sie im Abschnitt »Die Eigenschaft SQL zur Entwurfszeit setzen« auf Seite 21-7.
4. Um die Abfragedaten in visuellen Steuerelementen anzuzeigen, fügen Sie eine Datenquellenkomponente aus der Registerkarte *Datenzugriff* der Komponentenpalette in das Datenmodul ein und weisen ihrer Eigenschaft *DataSet* den Namen der Abfragekomponente zu. Die Datenquellenkomponente wird verwendet, um die Abfrageergebnisse (die sogenannte *Ergebnismenge*) zum Zweck der Anzeige an die datensensitiven Komponenten zurückzugeben. Verbinden Sie die datensensitiven Komponenten über ihre Eigenschaften *DataSource* und *DataField* mit der Datenquelle.
5. Aktivieren Sie die Abfragekomponente. Verwenden Sie die Eigenschaft *Active* oder die Methode *Open* für Abfragen, die Ergebnismengen zurückliefern. Verwenden Sie die Methode *ExecSQL* für Abfragen, die in Tabellen nur bestimmte Aktionen durchführen und keine Ergebnismenge zurückgeben.

Führen Sie folgende Schritte aus, um eine Abfrage zur Laufzeit zum ersten Mal auszuführen:

1. Schließen Sie die Abfragekomponente.
2. Weisen Sie der Eigenschaft *SQL* eine SQL-Anweisung zu. Damit ersparen Sie sich das Setzen der Eigenschaft *SQL* zur Entwurfszeit. Sie können auf diese Weise auch eine bereits vorhandene SQL-Anweisung ändern. Wenn die zur Entwurfszeit erstellten Anweisungen unverändert verwendet werden sollen, übergehen Sie diesen Schritt. Weitere Informationen über das Setzen der Eigenschaft *SQL* finden Sie im Abschnitt »Die Eigenschaft SQL setzen« auf Seite 21-6.
3. Belegen Sie die Parameter und Parameterwerte in der Eigenschaft *Params* entweder direkt oder mit Hilfe der Methode *ParamByName*. Führen Sie diesen Schritt nicht aus, wenn die Abfrage keine Parameter enthält oder wenn die zur Entwurfszeit gesetzten Parameter noch gültig sind. Informationen über das Setzen von Parametern finden Sie im Abschnitt »Parameter setzen« auf Seite 21-9.
4. Rufen Sie *Prepare* auf, um die BDE zu initialisieren und die Parameter in die Abfrage einzubinden. Der Aufruf von *Prepare* ist optional, aber empfehlenswert. Weitere Informationen über die Vorbereitung einer Abfrage finden Sie im Abschnitt »Abfragen vorbereiten« auf Seite 21-15.
5. Rufen Sie für Abfragen, die eine Ergebnismenge zurückliefern, *Open* auf. Für Abfragen, die keine Ergebnisse zurückliefern, rufen Sie *ExecSQL* auf. Weitere Informationen über das Öffnen und Ausführen von Abfragen finden Sie im Abschnitt »Abfragen ausführen« auf Seite 21-13.

Nach der ersten Ausführung kann eine Anwendung die Abfrage wiederholt schließen und öffnen bzw. erneut ausführen, ohne daß diese neu vorbereitet werden muß. Dies gilt, solange Sie die SQL-Anweisung nicht ändern. Weitere Informationen über

die wiederholte Verwendung von Abfragen finden Sie im Abschnitt »Abfragen ausführen« auf Seite 21-13.

Die Eigenschaft SQL setzen

Mit Hilfe der Eigenschaft *SQL* können Sie die auszuführende SQL-Anweisung definieren. Zur Entwurfszeit wird eine Abfrage vorbereitet und automatisch ausgeführt, wenn Sie die Eigenschaft *Active* der Abfragekomponente auf *True* setzen. Zur Laufzeit wird die Abfrage mit einem Aufruf von *Prepare* vorbereitet und ausgeführt, sobald die Anwendung die Komponentenmethode *Open* oder *ExecSQL* aufruft.

Die Eigenschaft *SQL* ist ein *TStrings*-Objekt, das ein Array aus Text-Strings und eine Reihe von Eigenschaften, Ereignissen und Methoden enthält, welche die Strings bearbeiten. Die Strings werden in SQL automatisch verkettet und bilden so die auszuführende SQL-Anweisung. Sie können eine Anweisung in beliebig viele Strings aufteilen. Ein Vorteil bei der Verwendung mehrerer Strings liegt darin, daß sich die SQL-Anweisung in logische Einheiten aufteilen läßt (z.B. kann die WHERE-Klausel einer SELECT-Anweisung in einem eigenen String eingefügt werden). Dies erleichtert das Ändern und Testen der Abfrage.

Bei der SQL-Anweisung kann es sich sowohl um eine Abfrage handeln, die fest vergebene Feldnamen und Werte enthält, als auch um eine parametrisierte Abfrage mit auswechselbaren Parametern. Diese repräsentieren Feldwerte, die vor dem Ausführen der Anweisung eingebunden werden müssen. Die folgende Anweisung ist beispielsweise hart codiert:

```
SELECT * FROM Customer WHERE CustNo = 1231
```

Hart codierte Anweisungen sind von Nutzen, wenn Anwendungen jedesmal genau bekannte Abfragen ausführen. Sie können eine derartige Abfrage zur Entwurfs- oder Laufzeit nach Bedarf durch eine andere hart codierte oder parametrisierte Abfrage ersetzen. Bei jeder Änderung der Eigenschaft *SQL* wird die Abfrage automatisch geschlossen und zurückgesetzt.

Hinweis Enthalten Spaltennamen in einer BDE-Abfrage mit Local SQL Leerzeichen oder Sonderzeichen, sind diese Namen in Anführungszeichen einzuschließen. Außerdem muß vor jedem Spaltennamen eine Tabellenreferenz und ein Punkt stehen, z.B. BIOLIFE."Gattung". Weitere Informationen über gültige Spaltennamen finden Sie in der Server-Dokumentation.

Eine parametrisierte Abfrage enthält einen oder mehrere Platzhalterparameter. Es handelt sich dabei um Anwendungsvariablen, die für Vergleichswerte, wie z.B. diejenigen in der WHERE-Klausel einer SELECT-Anweisung, stehen. Bei parametrisierten Abfragen können Sie den Wert ändern, ohne die Anwendung umschreiben zu müssen. Die Parameterwerte müssen vor der ersten Ausführung der SQL-Anweisung eingebunden werden. Abfragekomponenten führen dies automatisch durch, es sei denn, Sie rufen vor der Ausführung der Abfrage explizit die Methode *Prepare* auf.

Die folgende Anweisung ist eine parametrisierte Abfrage:

```
SELECT * FROM Customer WHERE CustNo = :Anzahl
```

Die Variable *Anzahl*, gekennzeichnet durch den vorangestellten Doppelpunkt, ist ein Parameter für einen Vergleichswert, der zur Laufzeit angegeben werden muß. Bei jeder Ausführung der Anweisung kann ein anderer Wert angegeben werden. Der jeweilige Wert für *Anzahl* wird in der Eigenschaft *Params* der Abfragekomponente zur Verfügung gestellt.

Tip Im Sinne eines guten Programmierstils sollten Sie für Parameter Variablennamen vergeben, die dem Namen der Spalte entsprechen, die über den betreffenden Parameter angesprochen wird. Wenn ein Spaltenname beispielsweise »Anzahl« lautet, sollte der entsprechende Parameter »:Anzahl« heißen. Übereinstimmende Namen stellen sicher, daß die Variablennamen gültigen Feldnamen entsprechen, wenn eine Abfrage über die Eigenschaft *DataSource* Parameterwerte vergibt.

Die Eigenschaft SQL zur Entwurfszeit setzen

Sie können der Eigenschaft *SQL* zur Entwurfszeit mit Hilfe des Stringlisten-Editors einen Wert zuweisen. Dafür gibt es zwei Möglichkeiten:

- Doppelklicken Sie auf die Wertespalte der Eigenschaft *SQL*.
- Klicken Sie auf die Ellipsen-Schaltfläche der Eigenschaft *SQL*.

Eine SQL-Anweisung darf beliebig viele Zeilen umfassen. Die Gliederung von Anweisungen in mehrere Zeilen erleichtert das Lesen, Ändern und Testen. Klicken Sie auf *OK*, wenn Sie den eingegebenen Text der Eigenschaft *SQL* zuweisen wollen.

Normalerweise kann die Eigenschaft *SQL* nur eine einzige vollständige SQL-Anweisung enthalten, wobei die Anweisung jedoch sehr komplex sein kann (z.B. eine SELECT-Anweisung mit einer WHERE-Klausel, die ihrerseits mehrere logische Operatoren wie AND und OR enthält). Einige Server unterstützen auch eine »Batch«-Syntax, die die Verarbeitung mehrerer Anweisungen ermöglicht. Wenn Ihr Server diese Syntax unterstützt, kann die Eigenschaft *SQL* mehrere Anweisungen enthalten.

Hinweis Mit einigen Versionen von Delphi können Sie auch den SQL Builder zum Erstellen einer Abfrage einsetzen. Der visuelle Abfragegenerator unterstützt Sie durch die Anzeige der benötigten Datenbanktabellen und -felder. Um dieses Modul zu aktivieren, wählen Sie eine Abfragekomponente aus, klicken mit der rechten Maustaste und wählen im lokalen Menü den entsprechenden Befehl. Details über den SQL Builder finden Sie in der Online-Hilfe.

Die Eigenschaft SQL zur Laufzeit setzen

Es gibt drei Möglichkeiten, die Eigenschaft *SQL* zur Laufzeit zu setzen. Eine Anwendung kann der Eigenschaft *SQL* direkt einen Wert zuweisen, sie kann die Methode *LoadFromFile* der Eigenschaft *SQL* aufrufen, um die SQL-Anweisung aus einer Datei zu lesen, und sie kann der Eigenschaft *SQL* eine SQL-Anweisung in einem Stringlisten-Objekt zuweisen.

Die Eigenschaft SQL direkt setzen

So weisen Sie der Eigenschaft *SQL* zur Laufzeit direkt einen Wert zu:

1. Rufen Sie die Methode *Close* auf, um die Abfrage zu deaktivieren. Ein Versuch, die Eigenschaft *SQL* zu ändern, deaktiviert die Abfrage automatisch, es ist jedoch eine gute Sicherheitsmaßnahme, dies explizit zu tun.
2. Wenn Sie die gesamte SQL-Anweisung ersetzen, rufen Sie die Methode *Clear* der Eigenschaft *SQL* auf, um die aktuelle SQL-Anweisung zu löschen.
3. Wenn Sie die gesamte SQL-Anweisung neu aufbauen oder an eine vorhandene Anweisung eine Zeile anhängen, rufen Sie die Methode *Add* der Eigenschaft *SQL* auf, um einen oder mehrere Strings einzufügen oder anzuhängen. Dadurch wird eine neue SQL-Anweisung erstellt. Wenn Sie eine vorhandene Zeile ändern, verwenden Sie die Eigenschaft *SQL* mit einem Index, der die betreffende Zeile angibt, und weisen Sie den neuen Wert zu.
4. Rufen Sie *Open* oder *ExecSQL* auf, um die Abfrage auszuführen.

Der folgende Quelltext zeigt die Erstellung einer vollständig neuen SQL-Anweisung:

```
with CustomerQuery do begin
   Close; { Abfrage schließen, wenn sie aktiv ist }
   with SQL do begin
    Clear; { Aktuelle SQL-Anweisung löschen, falls vorhanden }
    Add('SELECT * FROM Customer'); { Erste SQL-Zeile hinzufügen... }
    Add('WHERE Company = "Sight Diver"'); { ... zweite Zeile hinzufügen... }
   end;
   Open; { ...und Abfrage aktivieren }
end;
```

Der folgende Quelltext zeigt, wie in einer vorhandenen SQL-Anweisung eine einzelne Zeile geändert wird. In diesem Fall ist in der zweiten Zeile der Anweisung die WHERE-Klausel bereits vorhanden. Sie wird über die Eigenschaft *SQL* mit dem Index 1 referenziert.

```
CustomerQuery.SQL[1] := 'WHERE Company = "Kauai Dive Shoppe"';
```

Hinweis Wenn eine Abfrage Parameter verwendet, müssen Sie deren Anfangswerte setzen und die Methode *Prepare* aufrufen, bevor die Abfrage geöffnet oder ausgeführt wird. Ein expliziter Aufruf von *Prepare* ist sinnvoll, wenn dieselbe SQL-Anweisung wiederholt verwendet wird. Andernfalls wird diese Methode von der Abfragekomponente automatisch aufgerufen.

Die Eigenschaft SQL aus einer Datei laden

Mit der Methode *LoadFromFile* der Eigenschaft *SQL* können Sie eine SQL-Anweisung auch aus einer Textdatei laden. Die Methode *LoadFromFile* löscht automatisch den aktuellen Inhalt der Eigenschaft *SQL*, bevor die neue Anweisung aus einer Datei geladen wird. Beispiel:

```
CustomerQuery.Close;
CustomerQuery.SQL.LoadFromFile('c:\orders.txt');
CustomerQuery.Open;
```

Hinweis Wenn die SQL-Anweisung in der Datei eine parametrisierte Abfrage ist, initialisieren Sie vor dem Öffnen oder Ausführen der Abfrage die Parameter und rufen *Prepare* auf. Ein expliziter Aufruf von *Prepare* ist sinnvoll, wenn dieselbe SQL-Anweisung wiederholt verwendet wird. Andernfalls wird diese Methode von der Abfragekomponente automatisch aufgerufen.

Die Eigenschaft SQL aus einem Stringlisten-Objekt laden

Sie können auch die Methode *Assign* der Eigenschaft *SQL* verwenden, um den Inhalt eines Stringlisten-Objekts in die Eigenschaft *SQL* zu kopieren. Diese Methode löscht automatisch den aktuellen Inhalt der Eigenschaft *SQL*, bevor die neue Anweisung kopiert wird. Im folgenden Beispiel wird eine SQL-Anweisung aus einer *TMemo*-Komponente kopiert:

```
CustomerQuery.Close;
CustomerQuery.SQL.Assign(Memo1.Lines);
CustomerQuery.Open;
```

Hinweis Wenn die SQL-Anweisung eine parametrisierte Abfrage ist, initialisieren Sie vor dem Öffnen oder Ausführen der Abfrage die Parameter und rufen *Prepare* auf. Ein expliziter Aufruf von *Prepare* ist sinnvoll, wenn dieselbe SQL-Anweisung wiederholt verwendet wird. Andernfalls wird diese Methode von der Abfragekomponente automatisch aufgerufen.

Parameter setzen

Eine parametrisierte SQL-Anweisung enthält Parameter oder Variablen, deren Werte sich zur Entwurfs- oder zur Laufzeit ändern können. Parameter können für Datenwerte stehen. Ein Beispiel dafür sind die Werte in der WHERE-Klausel einer SQL-Anweisung. Normalerweise stehen Parameter für Datenwerte, die an die Anweisung übergeben werden. In der folgenden INSERT-Anweisung werden die einzufügenden Werte als Parameter übergeben:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In dieser SQL-Anweisung sind *:Name*, *:Capital* und *:Population* Platzhalter für die tatsächlichen Werte, die zur Laufzeit an die Anweisung übergeben werden. Bevor eine parametrisierte Abfrage zum ersten Mal ausgeführt wird, müssen Sie die Methode *Prepare* aufrufen, um die aktuellen Werte für die Parameter in die SQL-Anweisung einzubinden. Einbinden bedeutet in diesem Fall, daß die BDE und der Server Ressourcen für die Anweisung und ihre Parameter bereitstellen, wodurch die Ausführungsgeschwindigkeit der Abfrage erhöht wird.

```
with Query1 do begin
   Close;
   Unprepare;
   ParamByName('Name').AsString := 'Belize';
   ParamByName('Capital').AsString := 'Belmopan';
   ParamByName('Population').AsInteger := '240000';
   Prepare;
   Open;
```

```
end;
```

Parameter zur Entwurfszeit setzen

Die Parameter in der SQL-Anweisung werden zur Entwurfszeit im Parameter-Editor angezeigt. Um auf die *TParam*-Objekte der Parameter zuzugreifen, rufen Sie den Parameter-Editor auf, wählen einen Parameter aus und greifen im Objektinspektor auf die *TParam*-Eigenschaften zu. Wenn die SQL-Anweisung keine Parameter enthält, sind im Parameter-Editor keine *TParam*-Objekte aufgeführt. Parameter können nur hinzugefügt werden, indem sie in die SQL-Anweisung geschrieben werden.

So rufen Sie den Parameter-Editor auf:

1. Wählen Sie die Abfragekomponente aus.
2. Klicken Sie im Objektinspektor auf die Ellipsen-Schaltfläche der Eigenschaft *Params*.
3. Wählen Sie im Parameter-Editor einen Parameter aus.
4. Das *TParam*-Objekt für den ausgewählten Parameter wird im Objektinspektor angezeigt.
5. Überprüfen und ändern Sie die Eigenschaften des *TParam*-Objekts im Objektinspektor.

Bei Abfragen, für die keine Parameter definiert wurden (die Eigenschaft *SQL* ist leer oder die SQL-Anweisung enthält keine Parameter), ist die Parameterliste im Parameter-Editor leer. Wurden für eine Abfrage bereits Parameter definiert, werden im Parameter-Editor alle vorhandenen Parameter angezeigt.

Hinweis Die *TQuery*-Komponente greift auf das *TParam*-Objekt und den Parameter-Editor gemeinsam mit verschiedenen anderen Komponenten zu. Das lokale Menü des Parameter-Editors enthält zwar immer die Optionen *Hinzufügen* und *Löschen*, diese sind jedoch für *TQuery*-Parameter niemals aktiviert. Sie können *TQuery*-Parameter nur hinzufügen oder löschen, indem Sie dies in der SQL-Anweisung selber vornehmen.

Die Auswahl eines Parameters im Parameter-Editor bewirkt, daß im Objektinspektor die entsprechenden Eigenschaften und Ereignisse angezeigt werden. Legen Sie die Werte für Eigenschaften und Methoden von Parametern im Objektinspektor fest.

Für die Eigenschaft *DataType* wird der BDE-Datentyp für den im Dialogfeld ausgewählten Parameter aufgeführt. Anfangs ist dies der Datentyp *ftUnknown*. Sie müssen für jeden Parameter einen Datentyp vergeben. Im allgemeinen entsprechen die Datentypen der BDE denjenigen des Servers. Eine Gegenüberstellung der BDE- und Server-Datentypen finden Sie in der BDE-Hilfe in \Borland\Gemeinsame Dateien\BDE\BDE32.HLP.

Für die Eigenschaft *ParamType* wird der Typ des im Dialogfeld ausgewählten Parameters aufgeführt. Anfangs ist dies der Typ *ptUnknown*. Sie müssen für jeden Parameter einen Typ vergeben.

Verwenden Sie die Eigenschaft *Value*, um zur Entwurfszeit einen Wert für den ausgewählten Parameter festzulegen. Dies ist nicht erforderlich, wenn die Werte der Parameter zur Laufzeit zur Verfügung gestellt werden. In diesem Fall lassen Sie *Value* leer.

Parameter zur Laufzeit setzen

Für die Definition von Parametern zur Laufzeit stehen folgende Möglichkeiten zur Verfügung:

- Weisen Sie den Parametern mit der Methode *ParamByName* Werte über den Parameternamen zu.
- Verwenden Sie die Eigenschaft *Params*, um den Parametern Werte über ihre Position in der SQL-Anweisung zuzuweisen.
- Verwenden Sie die Eigenschaft *Params.ParamValues*, um einem oder mehreren Parametern in einer einzelnen Befehlszeile Werte über den Namen der Parametergruppe zuzuweisen. Diese Methode verwendet Varianten, so daß für die Werte keine Typumwandlung durchgeführt werden muß.

Alle nachfolgenden Beispiele setzen voraus, daß die Eigenschaft *SQL* die folgende SQL-Anweisung enthält. Alle drei verwendeten Parameter haben den Datentyp *ftString*.

```
INSERT INTO "COUNTRY.DB"
(Name, Capital, Continent)
VALUES (:Name, :Capital, :Continent)
```

Im folgenden Quelltext wird mit der Methode *ParamByName* dem Parameter *Capital* der Text eines Eingabefeldes zugewiesen:

```
Query1.ParamByName('Capital').AsString := Edit1.Text;
```

Derselbe Quelltext würde bei Verwendung der Eigenschaft *Params* mit dem Index 1 (der Parameter *Capital* ist der zweite Parameter in der SQL-Anweisung) folgendermaßen aussehen:

```
Query1.Params[1].AsString := Edit1.Text;
```

Die folgende Befehlszeile weist über die Eigenschaft *Params.ParamValues* allen drei Parametern gleichzeitig Werte zu:

```
Query1.Params.ParamValues['Country;Capital;Continent'] :=
   VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Datenquellen für die Parameterbindung

Wenn Parameterwerte für eine parametrisierte Abfrage nicht zur Entwurfszeit gebunden oder zur Laufzeit angegeben werden, versucht die Abfragekomponente, den Parametern über die Eigenschaft *DataSource* Werte zuzuweisen. *DataSource* gibt eine andere Tabellen- oder Abfragekomponente an, in der die Abfragekomponente nach Feldnamen suchen kann, die mit den Namen der nicht gebundenen Parameter übereinstimmen. Die Suchdatenmenge muß erstellt und gefüllt werden, bevor die Abfragekomponente, die sich darauf bezieht, angelegt wird. Werden in der Suchdatenmen-

ge Übereinstimmungen gefunden, bindet die Abfragekomponente die Parameterwerte an die Feldwerte im aktuellen Datensatz der Tabelle oder Abfrage, auf welche die Eigenschaft *DataSource* zeigt.

An einer einfachen Anwendung läßt sich zeigen, wie mit der Eigenschaft *DataSource* eine Abfrage in einem Haupt/Detailformular verknüpft werden kann. Angenommen, das Datenmodul für diese Anwendung trägt den Namen *LinkModule* und enthält eine Abfragekomponente namens *OrdersQuery*, deren Eigenschaft *SQL* folgenden Inhalt hat:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = :CustNo
```

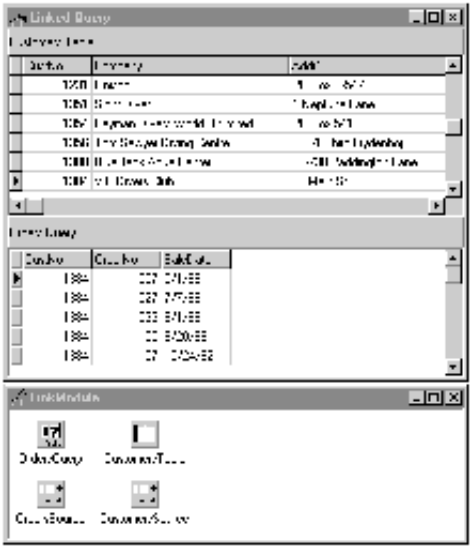
Das Datenmodul *LinkModule* enthält außerdem

- eine *TTable*-Datenmengenkomponente namens *CustomersTable*, die mit der Tabelle CUSTOMER.DB verknüpft ist.
- eine *TDataSource*-Komponente namens *OrdersSource*. Die Eigenschaft *DataSet* von *OrdersSource* zeigt auf *OrdersQuery*.
- eine *TDataSource*-Komponente namens *CustomersSource*, die mit *CustomersTable* verknüpft ist. Die Eigenschaft *DataSource* der *OrdersQuery*-Komponente zeigt auf *CustomersSource*. *OrdersQuery* ist also eine verknüpfte Abfrage.

Nehmen wir weiter an, daß diese Anwendung über ein Formular namens *LinkedQuery* verfügt, das zwei Datengitter enthält: ein Gitter für *CustomersTable*, das mit *CustomersSource* verknüpft ist, und ein Gitter für *OrdersQuery*, das mit *OrdersSource* verknüpft ist.

Die folgende Abbildung zeigt, wie diese Anwendung zur Entwurfszeit aussieht.

Abbildung 21.1 Beispiel für ein Haupt/Detail-Abfrageformular und ein Datenmodul zur Entwurfszeit



Hinweis Legen Sie vor der Erstellung der Abfragekomponente eine Tabellenkomponente und ihre Datenquelle an.

Bei der Compilierung dieser Anwendung wird dem Parameter *:CustNo* in der SQL-Anweisung zur Laufzeit kein Wert zugewiesen. *OrdersQuery* versucht deshalb, in der Tabelle, auf die *CustomersSource* zeigt, einen Spaltennamen zu finden, der dem Parameternamen entspricht. *CustomersSource* erhält seine Daten aus der Komponente *CustomersTable*, die ihrerseits Daten aus der Tabelle CUSTOMER.DB abruft. Da CUSTOMER.DB eine Spalte namens *CustNo* enthält, wird der Wert aus dem Feld *CustNo* im aktuellen Datensatz der Datenmenge *CustomersTable* dem Parameter *:CustNo* der SQL-Anweisung von *OrdersQuery* zugewiesen. Die Gitter sind in einer Haupt/Detail-Beziehung miteinander verknüpft. Wenn Sie zur Laufzeit im Gitter von *CustomersTable* den aktuellen Datensatz wechseln, ruft die SELECT-Anweisung von *OrdersQuery* erneut alle Aufträge gemäß der aktuellen Kundennummer ab.

Abfragen ausführen

Sobald Sie eine SQL-Anweisung in der Eigenschaft *SQL* angegeben und die Parameter für die Abfrage bereitgestellt haben, können Sie die Abfrage ausführen. Dabei empfängt und bearbeitet die BDE die SQL-Anweisungen Ihrer Anwendung. Wird die Abfrage in einer lokalen Tabelle (dBASE und Paradox) durchgeführt, verarbeitet die SQL-Engine der BDE die SQL-Anweisung und gibt bei einer SELECT-Abfrage Daten an die Anwendung zurück. Wenn die Abfrage in der Tabelle eines SQL-Servers durchgeführt wird und die Eigenschaft *TQuery.RequestLive* den Wert *False* hat, wird die SQL-Anweisung nicht von der BDE verarbeitet oder interpretiert, sondern direkt an das Datenbanksystem übergeben. Hat die Eigenschaft *RequestLive* den Wert *True*, muß die SQL-Anweisung lokale SQL-Standards verwenden, da die BDE diese benötigt, um Datenänderungen in die Tabelle zu übertragen.

Hinweis Bevor Sie eine Abfrage das erste Mal ausführen, können Sie die Methode *Prepare* aufrufen. Die Abfrage wird dann schneller ausgeführt. *Prepare* initialisiert die BDE und den Datenbank-Server. Die BDE und der Server weisen der Abfrage im voraus Systemressourcen zu. Weitere Informationen über die Vorbereitung einer Abfrage finden Sie im Abschnitt »Abfragen vorbereiten« auf Seite 21-15.

In den folgenden Abschnitten wird die Ausführung statischer und dynamischer SQL-Anweisungen zur Entwurfs- und zur Laufzeit erläutert.

Abfragen zur Entwurfszeit ausführen

Um eine Abfrage zur Entwurfszeit auszuführen, setzen Sie im Objektinspektor ihre Eigenschaft *Active* auf *True*.

Die Ergebnisse der Abfrage werden (falls vorhanden) in den datensensitiven Steuerelementen angezeigt, die mit der Abfragekomponente verknüpft sind.

Hinweis Die Eigenschaft *Active* kann nur für Abfragen verwendet werden, die eine Ergebnismenge zurückgeben, wie z.B. bei einer SELECT-Anweisung.

Abfragen zur Laufzeit ausführen

Zur Ausführung einer Abfrage zur Laufzeit stehen Ihnen folgende Methoden zur Verfügung:

Die Methode *Open* führt eine Abfrage aus, die eine Ergebnismenge zurückliefert, wie bei einer SELECT-Anweisung.

Die Methode *ExecSQL* führt eine Abfrage aus, die keine Ergebnismenge zurückliefert, wie bei den Anweisungen INSERT, UPDATE oder DELETE.

Hinweis Wenn Sie zur Entwurfszeit nicht wissen, ob eine Abfrage zur Laufzeit eine Ergebnismenge zurückgibt, geben Sie beide Typen der Abfrageausführung in einem **try..except** -Block an. Rufen Sie die Methode *Open* im der **try**-Block auf. Dies ermöglicht Ihnen, die Fehlermeldung zu unterdrücken, die aufgrund einer nicht dem Typ der SQL-Anweisung entsprechenden Aktivierung auftreten würde. Überprüfen Sie den Typ der aufgetretenen Exception. Handelt es sich nicht um eine *ENoResult*-Exception, ist sie aus einem anderen Grund aufgetreten und muß verarbeitet werden. Dieses Verfahren funktioniert, weil die Abfrage ausgeführt wird, wenn sie mit der Methode *Open* aktiviert wird, und die Exception zusätzlich auftritt.

```
try
   Query2.Open;
except
   on E: Exception do
    if not (E is ENoResultSet) then
      raise;
end;
```

Abfragen ausführen, die Ergebnismengen liefern

Zur Ausführung einer Abfrage, die eine Ergebnismenge zurückliefert (weil sie eine SELECT-Anweisung verwendet), führen Sie folgende Schritte aus:

1. Rufen Sie die Methode *Close* auf, um sicherzustellen, daß die Abfrage nicht bereits geöffnet ist. Wenn eine Abfrage bereits geöffnet ist, müssen Sie sie erst schließen, bevor sie erneut geöffnet werden kann. Das Schließen und erneute Öffnen einer Abfrage ruft vom Server die aktuelle Version der Daten ab.
2. Rufen Sie die Methode *Open* auf, um die Abfrage auszuführen.

Beispiel:

```
CustomerQuery.Close;
CustomerQuery.Open; { Die Abfrage gibt eine Ergebnismenge zurück }
```

Informationen zur Navigation innerhalb einer Ergebnismenge finden Sie im Abschnitt »Bidirektionale Cursor deaktivieren« auf Seite 21-17. Die Bearbeitung und Aktualisierung von Ergebnismengen wird im Abschnitt »Mit Ergebnismengen arbeiten« auf Seite 21-18 beschrieben.

Abfragen ausführen, die keine Ergebnismengen liefern

Zur Ausführung einer Abfrage, die keine Ergebnismenge zurückliefert (eine Abfrage mit einer SQL-Anweisung wie INSERT, UPDATE oder DELETE), rufen Sie *ExecSQL* auf.

Beispiel:

```
CustomerQuery.ExecSQL;  { Die Abfrage liefert keine Ergebnismenge zurück }
```

Abfragen vorbereiten

Das Vorbereiten einer Abfrage ist ein optionaler Schritt, der der Ausführung einer Abfrage vorangeht. Bei der Vorbereitung einer Abfrage werden die SQL-Anweisung und ihre Parameter (falls vorhanden) zur Analyse, zur Zuweisung von Ressourcen und zur Optimierung an die BDE übergeben. Die BDE benachrichtigt den Datenbank-Server über die Vorbereitung der Abfrage, so daß dieser Ressourcen bereitstellen kann. Diese Operationen beschleunigen die Abfrage und damit auch die Anwendung, besonders wenn aktualisierbare Abfragen verwendet werden.

In einer Anwendung kann eine Abfrage mit der Methode *Prepare* vorbereitet werden. Wenn Sie eine Abfrage nicht vorbereiten, bevor sie ausgeführt wird, führt Delphi die Vorbereitung immer dann aus, wenn Sie die Methode *Open* oder *ExecSQL* aufrufen. Obwohl Delphi alle Abfragen für Sie vorbereitet, ist es empfehlenswert, sie explizit vorzubereiten. Das Programm wird dadurch besser lesbar. Beispiel:

```
CustomerQuery.Close;
if not (CustomerQuery.Prepared) then
   CustomerQuery.Prepare;
CustomerQuery.Open;
```

In diesem Beispiel wird durch die Überprüfung der Eigenschaft *Prepared* der Abfragekomponente festgestellt, ob die Abfrage bereits vorbereitet ist. *Prepared* ist ein Boolescher Wert (*True*, wenn die Abfrage bereits vorbereitet ist). Wenn die Abfrage noch nicht vorbereitet ist, wird in diesem Beispiel vor *Open* die Methode *Prepare* aufgerufen.

Abfragen zur Freigabe von Ressourcen zurücksetzen

Die Methode *UnPrepare* setzt die Eigenschaft *Prepared* auf *False*. *UnPrepare* führt folgende Aufgaben aus:

- Sie stellt sicher, daß die Eigenschaft *SQL* vor ihrer erneuten Ausführung neu vorbereitet wird.
- Sie benachrichtigt die BDE darüber, daß die internen, der Anweisung zugewiesenen Ressourcen freigegeben werden sollen.
- Sie benachrichtigt den Server darüber, daß die der Anweisung zugewiesenen Ressourcen freigegeben werden sollen.

So setzen Sie eine Abfrage zurück:

```
CustomerQuery.UnPrepare;
```

Hinweis Wenn Sie den Text der Eigenschaft *SQL* einer Abfrage ändern, schließt die Abfragekomponente die Abfrage automatisch und setzt sie zurück.

Heterogene Abfragen erstellen

Delphi unterstützt *heterogene Abfragen*, also Abfragen, die sich auf Tabellen in mehreren Datenbanken beziehen. Eine heterogene Abfrage kann Tabellen auf verschiedenen Servern miteinander verknüpfen, sogar auf Servern unterschiedlichen Typs. Eine heterogene Abfrage könnte beispielsweise eine Tabelle in einer Oracle-Datenbank, eine Tabelle in einer Sybase-Datenbank und eine lokale dBASE-Tabelle einbeziehen. Wenn Sie eine heterogene Abfrage ausführen, analysiert und bearbeitet die BDE die Abfrage mit Hilfe von Local SQL. Da die BDE die Sprache Local SQL verwendet, wird eine erweiterte, serverspezifische SQL-Syntax nicht unterstützt.

So realisieren Sie eine heterogene Abfrage:

1. Definieren Sie separate BDE-Aliase für jede Datenbank, auf die in der Abfrage zugegriffen wird. Geben Sie keinen Wert für die Eigenschaft *DatabaseName* von *TQuery* an. Die Namen der beiden Datenbanken werden in der SQL-Anweisung definiert.
2. Geben Sie die SQL-Anweisung an, die in der Eigenschaft *SQL* ausgeführt werden soll. Fügen Sie vor jeden Tabellennamen in der SQL-Anweisung den BDE-Alias für die Datenbank ein, in der sich die genannte Tabelle befindet. Der Tabellenreferenz geht der Name des BDE-Alias, eingeschlossen in Doppelpunkten, voraus. Die gesamte Referenz wird in Anführungszeichen eingeschlossen.
3. Setzen Sie in der Eigenschaft *Params* die Parameter für die Abfrage.
4. Rufen Sie die Methode *Prepare* auf, um die Abfrage vor ihrer ersten Ausführung vorzubereiten.
5. Rufen Sie die Methode *Open* oder *ExecSQL* auf, je nachdem, von welchem Typ die auszuführende Abfrage ist.

Angenommen, Sie definieren einen Alias namens *Oracle1* für eine Oracle-Datenbank, in der die Tabelle CUSTOMER enthalten ist, und einen Alias namens *Sybase1* für eine Sybase-Datenbank, in der sich eine Tabelle mit dem Namen ORDERS befindet. Eine einfache Abfrage über diese beiden Tabellen könnte wie folgt aussehen:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
   JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

In einer heterogenen Abfrage können Sie zur Angabe der Datenbank anstelle eines BDE-Alias auch eine *TDatabase*-Komponente verwenden. Konfigurieren Sie die *TDatabase*-Komponente wie gewöhnlich so, daß sie auf die Datenbank zeigt, setzen

Sie *TDatabase.DatabaseName* auf einen beliebigen, jedoch eindeutigen Wert, und verwenden Sie diesen Wert anstelle des Namens des BDE-Alias in der SQL-Anweisung.

Die Ausführungsgeschwindigkeit von Abfragen erhöhen

Durch folgende Maßnahmen können Sie die Ausführungsgeschwindigkeit von Abfragen erhöhen:

- Setzen Sie die Eigenschaft *UniDirectional* der Abfrage auf *True*, wenn keine Rückwärtsbewegungen in einer Ergebnismenge erforderlich sind (SQL-92 erlaubt keine Rückwärtsbewegungen in der Ergebnismenge). Per Voreinstellung ist die Eigenschaft *UniDirectional* auf *False* gesetzt, weil die BDE per Voreinstellung bidirektionale Cursor unterstützt.
- Bereiten Sie die Abfrage vor der Ausführung vor. Dies ist besonders dann von Nutzen, wenn Sie eine einfache Abfrage mehrere Male ausführen wollen. Sie müssen die Abfrage nur ein einziges Mal vor ihrer ersten Ausführung vorbereiten. Weitere Informationen über die Vorbereitung von Abfragen finden Sie im Abschnitt »Abfragen vorbereiten« auf Seite 21-15

Bidirektionale Cursor deaktivieren

Die Eigenschaft *UniDirectional* legt fest, ob in der BDE bidirektionale Cursor für eine Abfrage aktiviert sind. Wenn eine Abfrage eine Ergebnismenge zurückliefert, erhält sie einen Cursor, d.h. einen Zeiger auf den ersten Datensatz der Ergebnismenge. Der Datensatz, auf den der Cursor zeigt, ist der momentan aktive Datensatz. Der aktuelle Datensatz dagegen ist derjenige, dessen Feldwerte in datensensitiven Komponenten angezeigt werden, die mit der Ergebnismenge der Datenquelle verknüpft sind.

Die Eigenschaft *UniDirectional* ist per Voreinstellung *False*. Das bedeutet, daß der Cursor für eine Ergebnismenge vor- und rückwärts durch die Datensätze bewegt werden kann. Die Unterstützung bidirektionaler Cursor erfordert zusätzlichen Verwaltungsaufwand, was dazu führen kann, daß bestimmte Abfragen langsamer ausgeführt werden. Um die Ausführungsgeschwindigkeit zu erhöhen, haben Sie die Möglichkeit, die Eigenschaft *UniDirectional* auf *True* zu setzen, um in der Ergebnismenge nur Vorwärtsbewegungen des Cursors zuzulassen.

Wenn in einer Ergebnismenge keine Rückwärtsbewegungen erforderlich sind, setzen Sie die Eigenschaft *UniDirectional* für die Abfrage auf *True*, und zwar vor der Vorbereitung und Ausführung der Abfrage. Beispiel:

```
if not (CustomerQuery.Prepared) then begin
   CustomerQuery.UniDirectional := True;
   CustomerQuery.Prepare;
end;
CustomerQuery.Open;  { Liefert eine Ergebnismenge, in der die Bewegung nur in eine Richtung
                       möglich ist }
```

Mit Ergebnismengen arbeiten

Per Voreinstellung hat die von einer Abfrage zurückgelieferte Ergebnismenge das Attribut »Nur Lesen«. Die Feldwerte der Ergebnismenge werden in der Anwendung in datensensitiven Steuerelementen angezeigt und können vom Benutzer nicht bearbeitet werden. Wenn Ergebnismengen bearbeitet werden sollen, muß die Anwendung eine aktualisierbare Ergebnismenge anfordern.

Die Bearbeitung einer Ergebnismenge ermöglichen

Für die Anforderung einer Ergebnismenge, die der Benutzer in datensensitiven Steuerelementen bearbeiten kann, setzen Sie die Eigenschaft *RequestLive* der Abfragekomponente auf *True*. Dies ist zwar noch keine Garantie für eine aktualisierbare Ergebnismenge, die BDE versucht jedoch, die Anforderung nach Möglichkeit zu erfüllen. Es gibt einige Einschränkungen in bezug auf die Anforderung von solchen Ergebnismengen, die davon abhängen, ob eine Abfrage den lokalen SQL-Parser oder den SQL-Parser des Servers verwendet. Heterogene Joins und Abfragen über Paradox- oder dBASE-Tabellen werden von der BDE mit Local SQL analysiert, während Abfragen über Tabellen auf einem Remote-Datenbankserver von dem betreffenden Server analysiert werden.

Wenn eine Anwendung eine aktualisierbare Ergebnismenge anfordert und erhält, wird die Eigenschaft *CanModify* für die Abfragekomponente auf *True* gesetzt.

Wenn eine Anwendung eine aktualisierbare Ergebnismenge anfordert, aber die Syntax der SELECT-Anweisung dies nicht zuläßt, liefert die BDE entweder

- eine Nur-Lesen-Ergebnismenge für Abfragen über Paradox und dBASE, oder
- einen Fehlercode für Passthrough-SQL-Abfragen auf einem Remote-Server.

Local SQL-Syntax für aktualisierbare Ergebnismengen

Bei Abfragen, die den lokalen SQL-Parser verwenden, bietet die BDE eine erweiterte Unterstützung für aktualisierbare Ergebnismengen. Dies gilt sowohl für Abfragen über einzelne als auch für solche über mehrere Tabellen. Der lokale SQL-Parser wird eingesetzt, wenn eine Abfrage über eine oder mehrere dBASE- oder Paradox-Tabellen bzw. über eine oder mehrere Tabellen des Remote-Servers ausgeführt wird und wenn den betreffenden Tabellennamen in der Abfrage ein BDE-Datenbank-Alias vorangeht. Die folgenden Abschnitte beschreiben die Einschränkungen, die berücksichtigt werden müssen, damit eine aktualisierbare Ergebnismenge für Local SQL zurückgeliefert wird.

Einschränkungen für Abfragen mit aktualisierbaren Ergebnismengen

Bei einer Abfrage über eine einzelne Tabelle oder Ansicht wird nur dann eine aktualisierbare Ergebnismenge zurückgeliefert, wenn die Abfrage keine der folgenden Klauseln, Anweisungen usw. enthält:

- Eine DISTINCT-Klausel in der SELECT-Anweisung
- Joins (Inner, Outer oder UNION-Klausel)
- Zusammenfassungsfunktionen mit oder ohne GROUP BY- oder HAVING-Klauseln
- Basistabellen oder Ansichten, die nicht aktualisiert werden können
- Unterabfragen
- ORDER BY-Klauseln, die nicht auf einem Index basieren

Remote-Server-SQL für aktualisierbare Ergebnismengen

Bei Abfragen, die mit Passthrough-SQL-Anweisungen arbeiten (dazu gehören alle Abfragen, die nur auf Remote-Datenbankservern ausgeführt werden), sind aktualisierbare Ergebnismengen auf die Standards beschränkt, die durch SQL-92 und weitere vom Server auferlegte Einschränkungen festgelegt werden.

Bei einer Abfrage über eine einzelne Tabelle oder Ansicht wird nur dann eine aktualisierbare Ergebnismenge zurückgeliefert, wenn die Abfrage keine der folgenden Klauseln, Anweisungen usw. enthält:

- Eine DISTINCT-Klausel in der SELECT-Anweisung
- Zusammenfassungsfunktionen mit oder ohne GROUP BY- oder HAVING-Klauseln
- Referenzen auf mehr als eine Basistabelle oder aktualisierbare Ansicht (Joins)
- Unterabfragen, die die Tabelle in der FROM-Klausel referenzieren

Einschränkungen bei der Bearbeitung von aktualisierbaren Ergebnismengen

Wenn eine Abfrage eine aktualisierbare Ergebnismenge zurückliefert, die verknüpfte Felder enthält, kann sie unter Umständen nicht direkt aktualisiert werden. Dies ist möglicherweise auch dann der Fall, wenn vor dem Aktualisierungsversuch die Indizes gewechselt werden. Sollte eine Aktualisierung aus diesen Gründen nicht möglich sein, können Sie die Ergebnismenge als Nur-Lesen-Ergebnismenge behandeln und versuchen, sie entsprechend zu aktualisieren.

Nur-Lesen-Ergebnismengen aktualisieren

Anwendungen können Daten, die in einer Nur-Lesen-Ergebnismenge zurückgeliefert werden, mit Hilfe von zwischengespeicherten Aktualisierungen modifizieren. Führen Sie zur Aktualisierung einer Nur-Lesen-Ergebnismenge, die mit einer Abfragekomponente verknüpft ist, folgende Schritte aus:

1. Fügen Sie eine *TUpdateSQL*-Komponente in das Datenmodul der Anwendung ein. Damit wird es möglich, Aktualisierungen an einer Nur-Lesen-Datenmenge vorzunehmen.
2. Weisen Sie die SQL-Anweisung, welche die Ergebnismenge aktualisieren soll, den Eigenschaften *ModifySQL*, *InsertSQL* oder *DeleteSQL* der Aktualisierungskomponente zu.
3. Setzen Sie die Eigenschaft *CachedUpdate* der Abfragekomponente auf *True*.

Informationen über die Verwendung zwischengespeicherter Aktualisierungen finden Sie in Kapitel 25, “Zwischengespeicherte Aktualisierungen”.

Kapitel

22

Stored Procedures

Dieses Kapitel beschreibt den Einsatz von Stored Procedures in Datenbankanwendungen. Eine Stored Procedure ist ein eigenständiges Programm, das in der datenbankspezifischen Sprache für Prozeduren und Trigger geschrieben ist. Es gibt zwei Typen von Stored Procedures. Der erste Typ ruft Daten ab (wie eine SELECT-Abfrage). Die abgerufenen Daten können in Form einer Datenmenge vorliegen, die aus einer oder mehreren Zeilen besteht, die in eine oder mehrere Spalten aufgeteilt sind. Sie können aber auch in Form von einzelnen Informationspaketen vorliegen. Der zweite Typ von Stored Procedures liefert keine Daten zurück, sondern führt eine Aktion mit Daten in der Datenbank durch (wie eine DELETE-Anweisung). Einige Datenbank-Server unterstützen beide Typen von Operationen in derselben Stored Procedure.

Stored Procedures, die Daten zurückgeben, tun dies auf unterschiedliche Weise. Dies hängt davon ab, wie die Stored Procedure aufgebaut ist und welches Datenbanksystem verwendet wird. Einige Datenbanksysteme, beispielsweise InterBase, geben alle Daten (Datenmengen und einzelne Informationspakete) ausschließlich mit Ausgabeparametern zurück. Andere können einen Cursor auf Daten zurückgeben. Wieder andere, beispielsweise Microsoft SQL Server und Sybase, können Datenmengen und einzelne Informationspakete zurückliefern.

In Delphi-Anwendungen erfolgt der Zugriff auf Stored Procedures über die Komponenten *TStoredProc* und *TQuery*. Welche der Komponenten für den Zugriff verwendet werden soll, hängt davon ab, wie die Stored Procedure codiert ist, ob und wie Daten zurückgegeben werden und welches Datenbanksystem verwendet wird. Die Komponenten *TStoredProc* und *TQuery* sind Nachkommen von *TDataSet* und erben ihr fundamentales Verhalten. Weitere Informationen zu *TDataSet* finden Sie in Kapitel 18, »Datenmengen«.

Eine Stored-Procedure-Komponente wird eingesetzt, um Stored Procedures auszuführen, die keine Daten zurückgeben, um einzelne Informationspakete in Form von Ausgabeparametern abzurufen und um eine zurückgegeben Datenmenge an eine verknüpfte Datenquellenkomponente weiterzugeben (letzteres ist datenbankspezifisch). Stored-Procedure-Komponenten erlauben die Übergabe von Werten als Para-

meter. Diese Parameter werden über die Eigenschaft *Params* definiert. Stored-Procedure-Komponenten stellen auch die Methode *GetResults* zur Verfügung, die bewirkt, daß die Stored Procedure eine Datenmenge zurückgibt. Dies ist bei einigen Datenbank-Servern erforderlich, bevor eine Ergebnismenge ausgegeben werden kann. Eine Stored-Procedure-Komponente sollte für Stored Procedures verwendet werden, die entweder keine Daten oder Daten nur über Ausgabeparameter zurückgeben.

Eine Abfragekomponente wird in erster Linie eingesetzt, um Stored Procedures auszuführen, die Datenmengen zurückgeben. Dazu gehören auch Stored Procedures von InterBase, die Datenmengen nur über Ausgabeparameter zurückgeben. Die Abfragekomponente kann auch zur Ausführung einer Stored Procedure verwendet werden, die keine Datenmengen oder Ausgabeparameterwerte zurückgibt.

Um einzelne Werte an eine Stored Procedure zu übergeben oder aus ihr abzurufen, verwenden Sie Parameter. Die Werte der Eingabeparameter werden in einer Stored Procedure an den entsprechenden Positionen als WHERE-Klausel einer SELECT-Anweisung verwendet. Über einen Ausgabeparameter kann eine Stored Procedure einen einzelnen Wert an die aufrufende Anwendung übergeben. Einige Stored Procedures liefern außerdem einen Ergebnisparameter. In der Dokumentation des verwendeten Datenbank-Servers finden Sie Details zu den unterstützten Parametertypen und ihrer Verwendung in der serverspezifischen Sprache für Stored Procedures.

Stored Procedures verwenden

Wenn Ihr Server Stored Procedures definiert, sollten Sie diese verwenden, wenn sie auf die Bedürfnisse Ihrer Anwendung abgestimmt sind. Der Entwickler eines Datenbank-Servers erstellt Stored Procedures für häufig wiederkehrende, datenbankspezifische Aufgaben. Typische Anwendungsfälle sind Operationen mit vielen Tabellenzeilen oder statistische sowie mathematische Funktionen. Wenn auf dem Remote-Datenbankserver, auf den Ihre Anwendung zugreift, Stored Procedures vorhanden sind, sollten Sie diese nutzen. Sie verbessern den Durchsatz der Datenbankanwendung:

- Die meist höhere Rechenleistung des Servers wird ausgenutzt.
- Der Datenverkehr im Netzwerk wird verringert, weil der Prozeß dort abläuft, wo sich auch die Daten befinden.

Betrachten wir eine Anwendung, die einen einzigen Wert berechnen muß: die Standardabweichung von Werten in einer großen Zahl von Datensätzen. Um dies innerhalb einer Anwendung zu erledigen, muß jeder einzelne Wert, der in die Berechnung eingeht, vom Server geholt werden, wodurch die Netzwerkbelastung steigt. Anschließend führt die Anwendung die Berechnung durch. Letztendlich ist aber nur ein einziger Wert verlangt – die Standardabweichung. Es wäre also wesentlich ökonomischer, auf dem Server eine Stored Procedure vorzusehen, welche die dort gespeicherten Daten liest, die Berechnung ausführt und nur das angeforderte Ergebnis an die Anwendung übergibt.

In der Dokumentation Ihrer Server-Datenbank finden Sie Details zur Unterstützung von Stored Procedures.

Mit Stored Procedures arbeiten

Wie eine Stored Procedure in einer Delphi-Anwendung eingesetzt wird, hängt davon ab, wie die Stored Procedure codiert ist, ob und wie Daten zurückgegeben werden und welcher Datenbank-Server verwendet wird.

Für den Zugriff auf eine Stored Procedure auf dem Server muß die Anwendung folgende Schritte ausführen:

1. Eine *TStoredProc*-Komponente muß instantiiert und optional mit einer Stored Procedure auf dem Server verknüpft werden. Sie können auch eine *TQuery*-Komponente instantiieren und den Inhalt ihrer Eigenschaft *SQL* so definieren, daß entweder eine SELECT-Abfrage in der Stored Procedure oder ein EXECUTE-Befehl ausgeführt wird, abhängig davon, ob die Stored Procedure eine Ergebnismenge zurückliefert. Weitere Informationen zum Erstellen einer *TStoredProc*-Komponente finden Sie unter »Stored Procedures erstellen« auf Seite 22-4. Weitere Informationen zum Erstellen einer *TQuery*-Komponente finden Sie in Kapitel 21, »Abfragen«.
2. Falls nötig, müssen Eingabeparameter für die Stored Procedure zur Verfügung gestellt werden. Wenn die Prozedurkomponente nicht mit einer Stored Procedure des Servers verknüpft ist, müssen Sie zusätzliche Informationen zu Eingabeparametern, wie z.B. Parameternamen und Datentypen, bereitstellen. Details zu Eingabeparametern finden Sie unter »Parameter während des Entwurfs einstellen« auf Seite 22-14.
3. Die Stored Procedure muß ausgeführt werden.
4. Die Ergebnis- und Ausgabeparameter müssen verarbeitet werden. Wie bei jeder anderen Datenmengenkomponente können Sie die Ergebnisdatenmenge überprüfen, die vom Server zurückgegeben wurde. Weitere Informationen zu Ergebnis- und Ausgabeparametern finden Sie unter »Ausgabeparameter verwenden« auf Seite 22-12 und »Ergebnisparameter verwenden« auf Seite 22-14. Details zur Anzeige von Datensätzen einer Datenmenge finden Sie unter »Stored Procedures, die Ergebnismengen zurückgeben« auf Seite 22-6.

Stored-Procedure-Komponenten erstellen

So erstellen Sie eine Komponente für eine Stored Procedure auf einem Datenbank-Server:

1. Die Komponenten für Stored Procedures befinden sich in der Registerkarte *Datenzugriff* der Komponentenpalette. Plazieren Sie eine solche Komponente in einem Datenmodul.
2. Setzen Sie die Eigenschaft *DatabaseName* der Komponente auf den Namen der Datenbank, in der die Stored Procedure definiert ist. *DatabaseName* muß ein BDE-Alias oder der Wert der Eigenschaft *DatabaseName* einer *TDatabase*-Komponente sein, die eine Verbindung zum Server herstellen kann.

   Normalerweise sollten Sie die Eigenschaft *DatabaseName* mit einem Wert belegen. Wenn jedoch die Server-Datenbank, auf der die Anwendung ausgeführt wird,

nicht verfügbar ist, können Sie dennoch eine Stored-Procedure-Komponente erstellen und einrichten, indem Sie die Eigenschaft *DatabaseName* weglassen und während des Entwurfs den Namen einer Stored Procedure sowie Eingabe-, Ausgabe- und Ergebnisparameter angeben. Weitere Informationen zu Eingabeparametern finden Sie unter »Ergebnisparameter verwenden« auf Seite 22-14. Details zu Ausgabeparametern finden Sie unter »Ausgabeparameter verwenden« auf Seite 22-12. Informationen zu Ergebnisparametern finden Sie unter »Ergebnisparameter verwenden« auf Seite 22-14.

3. Setzen Sie für die Eigenschaft *StoredProcName* den Namen der gewünschten Stored Procedure ein. Wenn für die Eigenschaft *DatabaseName* ein Wert angegeben wurde, können Sie den Namen der Stored Procedure aus der Dropdown-Liste der Eigenschaft wählen. Eine einzelne *TStoredProc*-Komponente kann verwendet werden, um eine beliebige Anzahl von Stored Procedures auszuführen, indem die Eigenschaft *StoredProcName* auf einen gültigen Namen gesetzt wird. Es ist nicht immer wünschenswert, *StoredProcName* zur Entwurfszeit einen Wert zuzuweisen.
4. Doppelklicken Sie in der Wertespalte der Eigenschaft *Params*, um den Parameter-Editor der Eigenschaft *StoredProc* aufzurufen, mit dessen Hilfe Sie die Ein- und Ausgabeparameter der Stored Procedure überprüfen können. Wenn unter Schritt 3 für die Stored Procedure kein Name angegeben wurde (oder ein Name, der auf dem unter Schritt 2 in der Eigenschaft *DatabaseName* angegebenen Server nicht vorhanden ist), ist der Parameter-Editor leer.

Nicht von allen Servern werden Parameter oder Parameterinformationen zurückgegeben. In der Dokumentation Ihres Servers können Sie nachlesen, welche Informationen mit Stored Procedures an die Client-Anwendung übergeben werden.

Hinweis Wenn Sie der Eigenschaft *DatabaseName* in Schritt 2 keinen Wert zuweisen, müssen Sie den Parameter-Editor von *StoredProc* verwenden, um Parameter zur Entwurfszeit einzurichten. Weitere Informationen zum Einstellen der Parameter zur Entwurfszeit finden Sie unter »Parameter während des Entwurfs einstellen« auf Seite 22-14.

Stored Procedures erstellen

Normalerweise werden Stored Procedures erstellt, wenn die Anwendung und die Datenbank erstellt werden. Dazu werden die vom Hersteller des Datenbanksystems mitgelieferten Tools verwendet. Sie können jedoch Stored Procedures auch zur Laufzeit erstellen. Die zu verwendende spezifische SQL-Anweisung hängt vom Datenbanksystem ab, da sich die einzelnen Datenbanksysteme hinsichtlich der Prozedursprache stark unterscheiden. In der Dokumentation Ihres Datenbanksystems können Sie nachlesen, welche Sprache unterstützt wird.

Eine Stored Procedure kann von einer Anwendung zur Laufzeit mit einer SQL-Anweisung erstellt werden (normalerweise CREATE PROCEDURE), die von einer *TQuery*-Komponente ausgegeben wird. Wenn in der Stored Procedure Parameter verwendet werden, sollten Sie der Eigenschaft *ParamCheck* der *TQuery*-Komponente den Wert *False* zuweisen. Dadurch wird verhindert, daß *TQuery* die Parameter der neuen Stored Procedure fälschlicherweise für ihre eigenen Parameter hält.

Hinweis Zum Überprüfen, Bearbeiten und Erstellen von Stored Procedures auf dem Server kann auch der SQL-Explorer verwendet werden.

Nachdem Sie der Eigenschaft *SQL* die Anweisung zum Erstellen der Stored Procedure zugewiesen haben, führen Sie die Anweisung aus, indem Sie die Methode *ExecSQL* aufrufen.

```
with Query1 do begin
   ParamCheck := False;
   with SQL do begin
    Clear;
    Add('CREATE PROCEDURE GET_MAX_EMP_NAME');
    Add('RETURNS (Max_Name CHAR(15))');
    Add('AS');
    Add('BEGIN');
    Add('  SELECT MAX(LAST_NAME)');
    Add('  FROM EMPLOYEE');
    Add('  INTO :Max_Name;');
    Add('  SUSPEND;');
    Add('END');
   end;
   ExecSQL;
end;
```

Stored Procedures vorbereiten und ausführen

Um eine Stored Procedure verwenden zu können, muß sie vorbereitet und ausgeführt werden.

Zur Vorbereitung einer Stored Procedure haben Sie folgende Möglichkeiten:

- Zur Entwurfszeit klicken Sie im Parameter-Editor auf *OK*.
- Zur Laufzeit rufen Sie die Methode *Prepare* der Prozedurkomponente auf.

Das folgende Beispiel bereitet eine Stored Procedure auf die Ausführung vor:

```
StoredProc1.Prepare;
```

Hinweis Eine Stored Procedure muß neu vorbereitet werden, wenn die Anwendung Parameterinformationen zur Laufzeit ändert, weil beispielsweise überladene Oracle-Prozeduren verwendet werden.

Wenn eine vorbereitete Stored Procedure schließlich ausgeführt werden soll, rufen Sie die Methode *ExecProc* auf. Das folgende Beispiel bereitet eine Stored Procedure vor und führt sie aus:

```
StoredProc1.Params[0].AsString := Edit1.Text;
StoredProc1.Prepare;
StoredProc1.ExecProc;
```

Hinweis Beim Versuch, eine Stored Procedure auszuführen, bevor sie vorbereitet wurde, bereitet die Komponente diese automatisch für Sie vor und setzt die Werte nach der Ausführung wieder zurück. Wenn eine Stored Procedure mehrmals ausgeführt werden soll, ist es sinnvoller, zuerst die Methode *Prepare* aufzurufen und nach Beenden der Stored Procedure die Methode *UnPrepare* einmal aufzurufen.

Beim Ausführen einer Stored Procedure können eines oder alle der folgenden Elemente zurückgegeben werden:

- Eine Datenmenge, die aus einem oder mehreren Datensätzen besteht. Die Datensätze können in datensensitiven Steuerelementen angezeigt werden, die über eine Datenquelle mit der Stored Procedure verbunden sind.
- Ausgabeparameter.
- Ein Ergebnisparameter, der Statusinformationen über die Ausführung der Stored Procedure enthält.

In der Dokumentation des Servers werden die Rückgabeelemente einer Stored Procedure näher erläutert.

Stored Procedures, die Ergebnismengen zurückgeben

Stored Procedures, die Datenmengen (Daten in Zeilen und Spalten) zurückgeben, sollten normalerweise zusammen mit einer Abfragekomponente verwendet werden. Wenn jedoch Datenbank-Server das Zurückgeben von Datenmengen von einer Stored Procedure unterstützen, kann auch eine Stored Procedure-Komponente verwendet werden.

Ergebnismengen mit einer TQuery-Komponente abrufen

So rufen Sie mit einer *TQuery*-Komponente eine Datenmenge von einer Stored Procedure ab:

1. Instantiieren Sie eine Abfragekomponente.
2. Weisen Sie der Eigenschaft *TQuery.SQL* eine SELECT-Abfrage zu, die anstelle des Tabellennamens den Namen der Stored Procedure verwendet.
3. Benötigt die Stored Procedure Eingabeparameter, geben Sie die Parameterwerte durch Kommas getrennt und in Klammern eingeschlossen nach dem Prozedurnamen an.
4. Weisen Sie der Eigenschaft *Active* den Wert *True* zu, oder rufen Sie die Methode *Open* auf.

Die nachfolgende InterBase-Implementierung der Stored Procedure GET_EMP_PROJ nimmt über den Eingabeparameter EMP_NO einen Wert entgegen und gibt über den Ausgabeparameter PROJ_ID eine Datenmenge zurück.

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
   FOR SELECT PROJ_ID
   FROM EMPLOYEE_PROJECT
   WHERE EMP_NO = :EMP_NO
   INTO :PROJ_ID
   DO
    SUSPEND;
```

```
END
```

Die SQL-Anweisung einer *TQuery*-Komponente, die diese Stored Procedure verwendet, würde folgendermaßen aussehen:

```
SELECT *
FROM GET_EMP_PROJ(52)
```

Ergebnismengen mit einer TStoredProc-Komponente abrufen

So rufen Sie mit einer *TStoredProc*-Komponente eine Datenmenge von einer Stored Procedure ab:

1. Instantiieren Sie eine Stored Procedure-Komponente.
2. Weisen Sie der Eigenschaft *StoredProcName* den Namen der Stored Procedure zu.
3. Benötigt die Stored Procedure Eingabeparameter, geben Sie die Parameterwerte mit der Eigenschaft *Params* oder der Methode *ParamByName* an.
4. Weisen Sie der Eigenschaft *Active* den Wert *True* zu, oder rufen Sie die Methode *Open* auf.

Die nachfolgende Sybase-Implementierung der Stored Procedure GET_EMPLOYEES nimmt den Eingabeparameter @EMP_NO entgegen und gibt, basierend auf diesem Wert, eine Ergebnismenge zurück.

```
CREATE PROCEDURE GET_EMPLOYEES @EMP_NO SMALLINT
AS SELECT EMP_NAME, EMPLOYEE_NO FROM EMPLOYEE_TABLE
WHERE (EMPLOYEE_NO = @EMP_NO)
```

Der Delphi-Quelltext, der dem Parameter einen Wert zuweist und die Stored-Procedure-Komponente aufruft, lautet folgendermaßen:

```
with StoredProc1 do begin
   Close;
   ParamByName('EMP_NO').AsSmallInt := 52;
   Active := True;
end;
```

Stored Procedures, die Daten mit Hilfe von Parametern zurückliefern

Stored Procedures können so aufgebaut sein, daß sie einzelne Informationspakete (nicht gesamte Datenzeilen) über Parameter abrufen können. Beispielsweise kann eine Stored Procedure den höchsten Wert einer Spalte abrufen, diesen um 1 erhöhen, und dann den Wert an die Anwendung zurückgeben. Mit *TQuery*- oder *TStoredProc*-Komponenten können diese Stored Procedures eingesetzt und ihre Werte überprüft werden. *TStoredProc*-Komponenten verdienen jedoch beim Abrufen von Parameterwerten den Vorzug.

Einzelne Werte mit einer TQuery-Komponente abrufen

Parameterwerte, die über eine *TQuery*-Komponente abgerufen werden, entsprechen einer Datenmenge mit einem Datensatz, auch wenn von der Stored Procedure nur

ein Parameter zurückgegeben wird. So rufen Sie mit einer *TQuery*-Komponente einzelne Werte aus Stored-Procedure-Parametern ab:

1. Instantiieren Sie eine Abfragekomponente.
2. Weisen Sie der Eigenschaft *TQuery.SQL* eine SELECT-Abfrage zu, die den Namen der Stored Procedure anstelle des Tabellennamens verwendet. Die SELECT-Klausel dieser Abfrage kann den Namen des Parameters angeben wie eine Tabellenspalte, oder den Operator * verwenden, um alle Parameterwerte abzurufen.
3. Benötigt die Stored Procedure Eingabeparameter, geben Sie die Parameterwerte durch Kommas getrennt und in Klammern eingeschlossen nach dem Prozedurnamen an.
4. Weisen Sie der Eigenschaft *Active* den Wert *True* zu, oder rufen Sie die Methode *Open* auf.

Die nachfolgende InterBase-Implementierung der Stored Procedure GET_HIGH_EMP_NAME ruft den höchsten Wert der Spalte LAST_NAME der Tabelle EMPLOYEE ab und gibt ihn über den Ausgabeparameter High_Last_Name zurück.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
   SELECT MAX(LAST_NAME)
   FROM EMPLOYEE
   INTO :High_Last_Name;
   SUSPEND;
END
```

Die SQL-Anweisung, die von einer *TQuery*-Komponente ausgegeben werden müßte, um diese Stored Procedure zu verwenden, sieht folgendermaßen aus:

```
SELECT High_Last_Name
FROM GET_HIGH_EMP_NAME
```

Einzelne Werte mit einer TStoredProc-Komponente abrufen

So rufen Sie mit einer *TStoredProc*-Komponente einzelne Werte aus den Ausgabeparametern einer Stored Procedure ab:

1. Instantiieren Sie eine Stored-Procedure-Komponente.
2. Geben Sie in der Eigenschaft *StoredProcName* den Namen der Stored Procedure an.
3. Benötigt die Stored Procedure Eingabeparameter, geben Sie die Parameterwerte mit der Eigenschaft *Params* oder der Methode *ParamByName* an.
4. Rufen Sie die Methode *ExecProc* auf.
5. Untersuchen Sie die Werte einzelner Ausgabeparameter mit der Eigenschaft *Params* oder der Methode *ParamByName.*

Die nachfolgend wiedergegebene InterBase-Implementierung der Stored Procedure GET_HIGH_EMP_NAME ruft den höchsten Wert der Spalte LAST_NAME der Tabel-

le EMPLOYEE ab und gibt ihn über den Ausgabeparameter High_Last_Name zurück.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
   SELECT MAX(LAST_NAME)
   FROM EMPLOYEE
   INTO :High_Last_Name;
   SUSPEND;
END
```

Der Delphi-Quelltext, der den Wert im Ausgabeparameter High_Last_Name ermittelt und ihn der Eigenschaft *Text* einer *TEdit*-Komponente zuweist, lautet folgendermaßen:

```
with StoredProc1 do begin
   StoredProcName := 'GET_HIGH_EMP_NAME';
   ExecProc;
   Edit1.Text := ParamByName('High_Last_Name').AsString;
end;
```

Stored Procedures, die Aktionen an Daten vornehmen

Stored Procedures können so codiert werden, daß sie keine Daten zurückgeben, sondern nur Aktionen in der Datenbank durchführen. Gute Beispiele hierfür sind SQL-Operationen, die INSERT- und DELETE-Anweisungen enthalten. Anstatt beispielsweise einem Benutzer das Löschen einer Tabellenzeile direkt zu erlauben, können Sie für diesen Vorgang eine Stored Procedure verwenden. Die Stored Procedure kann die Löschoperation überwachen und dabei die referentielle Integrität berücksichtigen, beispielsweise bei verschachtelten Löschvorgängen in voneinander abhängigen Tabellen.

Stored Procedures, die Aktionen ausführen, mit einer TQuery-Komponente aufrufen

So rufen Sie mit einer TQuery-Komponente eine Stored Procedure auf, die Aktionen ausführt:

1. Instantiieren Sie eine Abfragekomponente.
2. Weisen Sie der Eigenschaft *TQuery.SQL* den für die Ausführung der Stored Procedure erforderlichen Befehl und den Namen der Stored Procedure zu. (Der Befehl zur Ausführung einer Stored Procedure hängt vom Datenbanksystem ab. Bei einer InterBase-Datenbank lautet er EXECUTE PROCEDURE.)
3. Benötigt die Stored Procedure Eingabeparameter, geben Sie die Parameterwerte durch Kommas getrennt und in Klammern eingeschlossen nach dem Prozedurnamen an.
4. Rufen Sie die Methode *TQuery.ExecSQL* auf.

Die nachfolgende InterBase-Implementierung der Stored Procedure ADD_EMP_PROJ fügt eine neue Zeile in die Tabelle EMPLOYEE_PROJECT ein. Sie gibt weder eine Datenmenge noch einzelne Werte über Ausgabeparameter zurück.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
   BEGIN
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
    VALUES (:EMP_NO, :PROJ_ID);
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
   END
   SUSPEND;
END
```

Die SQL-Anweisung einer *TQuery*-Komponente, die diese Stored Procedure ausführen soll, sieht folgendermaßen aus:

```
EXECUTE PROCEDURE ADD_EMP_PROJ(20, "GUIDE")
```

Stored Procedures, die Aktionen ausführen, mit einer TStoredProc-Komponente aufrufen

So rufen Sie mit einer *TStoredProc*-Komponente einzelne Werte aus den Ausgabeparametern einer Stored Procedure ab:

1. Instantiieren Sie eine Stored-Procedure-Komponente.
2. Weisen Sie der Eigenschaft *StoredProcName* den Namen der Stored Procedure zu.
3. Benötigt die Stored Procedure Eingabeparameter, geben Sie die Parameterwerte mit der Eigenschaft *Params* oder der Methode *ParamByName* an.
4. Rufen Sie die Methode *ExecProc* auf.

Die nachfolgende InterBase-Implementierung der Stored Procedure ADD_EMP_PROJ fügt eine neue Zeile in die Tabelle EMPLOYEE_PROJECT ein. Sie gibt weder eine Datenmenge noch einzelne Werte über Ausgabeparameter zurück.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
   BEGIN
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
    VALUES (:EMP_NO, :PROJ_ID);
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
   END
   SUSPEND;
END
```

Der Delphi-Quelltext zur Ausführung der Stored Procedure ADD_EMP_PROJ lautet folgendermaßen:

```
with StoredProc1 do begin
   StoredProcName := 'ADD_EMP_PROJ';
```

```
    ExecProc;
end;
```

Parameter für Stored Procedures

Stored Procedures verwenden die folgenden vier Parametertypen:

- *Eingabeparameter* für die Übergabe von Werten an eine Stored Procedure.
- *Ausgabeparameter*, in denen die Stored Procedure Rückgabewerte an eine Anwendung liefert.
- *Eingabe-/Ausgabeparameter*, die dazu dienen, Werte zur Bearbeitung an eine Stored Procedure zu übergeben, und die von dieser zur Rückgabe von Werten an die Anwendung verwendet werden.
- *Ergebnisparameter*, mit denen einige Stored Procedures Fehler- oder Statuswerte an die Anwendung übergeben. Eine Stored Procedure kann immer nur einen einzigen Ergebnisparameter zurückgeben.

Ob eine Stored Procedure einen bestimmten Parametertyp verwendet, hängt sowohl von der allgemeinen Sprachimplementierung von Stored Procedures auf Ihrem Datenbank-Server als auch von bestimmten Instanzen einer Stored Procedure ab. Einzelne Stored Procedures auf einem Server können z.B. mit Hilfe von Eingabeparametern implementiert werden, was jedoch nicht erforderlich ist. Andererseits sind einige Parameter auf bestimmte Server abgestimmt. Stored Procedures von MS-SQL- und Sybase-Servern geben immer einen Ergebnisparameter zurück, die InterBase-Implementierung einer Stored Procedure dagegen nie.

Der Zugriff auf die Parameter von Stored Procedures erfolgt über die *TParam*-Objekte in der Eigenschaft *TStoredProc.Params*. Wenn Sie den Namen der Stored Procedure zur Entwurfszeit der Eigenschaft *StoredProcName* zuweisen, wird automatisch für jeden Parameter ein *TParam*-Objekt erzeugt und der Eigenschaft *Params* hinzugefügt. Wird der Name der Stored Procedure erst zur Laufzeit zugewiesen, muß das *TParam*-Objekt zur Laufzeit im Programm erzeugt werden. Wenn Sie die Stored Procedure nicht angeben und die *TParam*-Objekte manuell erzeugen, kann eine einzelne *TStoredProc*-Komponente für eine beliebige Anzahl verfügbarer Stored Procedures verwendet werden.

Hinweis Einige Stored Procedures geben zusätzlich zu Ausgabe- und Ergebnisparametern eine Datenmenge zurück. Anwendungen können deren Datensätze in datensensitiven Steuerelementen anzeigen, müssen jedoch Ausgabe- und Ergebnisparameter getrennt bearbeiten. Weitere Informationen zur Anzeige von Datensätzen in datensensitiven Steuerelementen finden Sie unter »Stored Procedures, die Ergebnismengen zurückgeben« auf Seite 22-6.

Eingabeparameter verwenden

Anwendungen verwenden Eingabeparameter, um einzelne Werte an eine Stored Procedure zu übergeben. Diese Werte werden dann in SQL-Anweisungen innerhalb der Stored Procedure eingesetzt, beispielsweise als Vergleichswerte in einer WHERE-Klausel. Wenn eine Stored Procedure Eingabeparameter benötigt, müssen Sie ihnen Werte zuweisen, bevor die Stored Procedure ausgeführt wird.

Wenn eine Stored Procedure eine Datenmenge zurückliefert und von einer SELECT-Abfrage in einer *TQuery*-Komponente verwendet wird, geben Sie die Parameterwerte durch Kommas getrennt und in Klammern eingeschlossen nach dem Namen der Stored Procedure an. Die folgende SQL-Anweisung ruft Daten von der Stored Procedure GET_EMP_PROJ ab und stellt als Eingabeparameter den Wert 52 zur Verfügung.

```
SELECT PROJ_ID
FROM GET_EMP_PROJ(52)
```

Wird eine Stored Procedure mit einer *TStoredProc*-Komponente ausgeführt, verwenden Sie die Eigenschaft *Params* oder die Methode *ParamByName* zum Setzen der einzelnen Eingabeparameter. Verwenden Sie die *TParam*-Eigenschaft, die dem Datentyp des Parameters entspricht, beispielsweise *TParam.AsString* für einen Parameter vom Typ CHAR. Weisen Sie den Eingabeparametern Werte zu, bevor die *TStoredProc*-Komponente aktiviert oder ausgeführt wird. Im folgenden Beispiel wird dem Parameter EMP_NO (vom Typ Smallint) der Stored Procedure GET_EMP_PROJ der Wert 52 zugewiesen.

```
with StoredProc1 do begin
   ParamByName('EMP_NO').AsSmallInt := 52;
   ExecProc;
end;
```

Ausgabeparameter verwenden

Stored Procedures verwenden Ausgabeparameter, um einzelne Werte an die Anwendung zurückzugeben, die die Stored Procedure aufruft. Ausgabeparametern können keine Werte zugewiesen werden, außer von Stored Procedures nach ihrer Ausführung. Um den Wert eines Ausgabeparameters abzurufen, überprüfen Sie ihn in der Anwendung, nachdem Sie die Methode *TStoredProc.ExecProc* aufgerufen haben.

Verwenden Sie die Eigenschaft *TStoredProc.Params* oder die Methode *TStoredProc.ParamByName*, um das *TParam*-Objekt zu referenzieren, das einen Parameter repräsentiert, und um den Wert des Parameters zu überprüfen. Das folgende Beispiel ruft einen Parameterwert ab und speichert ihn in der Eigenschaft *Text* einer *TEdit*-Komponente:

```
with StoredProc1 do begin
   ExecProc;
   Edit1.Text := Params[0].AsString;
end;
```

Fast alle Stored Procedures geben einen oder mehrere Ausgabeparameter zurück. Ausgabeparameter können die alleinigen Rückgabewerte einer Stored Procedure darstellen, wenn diese nicht gleichzeitig eine Datenmenge zurückgibt, sie können aber auch eine Menge von Werten sein, wenn die Stored Procedure gleichzeitig eine Datenmenge zurückliefert. Es kann sich aber auch um Werte handeln, die keinen direkten Bezug zu einem bestimmten Datensatz in der zurückgegebenen Datenmenge haben. In diesem Punkt ist die Implementierung von Stored Procedures bei jedem Server unterschiedlich.

Hinweis Der Quelltext einer Stored Procedure auf einem Informix-Server kann zu dem Schluß führen, daß Ausgabeparameter zurückgegeben werden, obwohl im Parameter-Editor keine diesbezüglichen Informationen angezeigt werden. Informix übersetzt Ausgabeparameter in eine einzelne Datensatzmenge, die in den datensensitiven Steuerelementen Ihrer Anwendung angezeigt werden kann.

Eingabe-/Ausgabeparameter verwenden

Eingabe-/Ausgabeparameter fassen die Funktionalität von Eingabe- und Ausgabeparametern zusammen. Sie werden von Anwendungen zur Übergabe von einzelnen Datenwerten an Stored Procedures und von diesen zur Rückgabe einzelner Werte an die aufrufende Anwendung verwendet. Wie bei Eingabeparametern müssen auch hier die Eingabewerte zugewiesen werden, bevor die Stored Procedure- oder Abfragekomponente aktiviert wird. Ebenso steht der Ausgabewert erst nach der Ausführung der Stored Procedure zur Verfügung.

Im folgenden Beispiel der Oracle-Implementierung einer Stored Procedure ist der Parameter IN_OUTVAR ein Eingabe-/Ausgabeparameter.

```
CREATE OR REPLACE PROCEDURE UPDATE_THE_TABLE (IN_OUTVAR IN OUT INTEGER)
AS
BEGIN
   UPDATE ALLTYPETABLE
   SET NUMBER82FLD = IN_OUTVAR
   WHERE KEYFIELD = 0;
   IN_OUTVAR:=1;
END UPDATE_THE_TABLE;
```

Im folgenden Delphi-Quelltext wird IN_OUTVAR ein Eingabewert zugewiesen und anschließend die Stored Procedure ausgeführt. Anschließend wird der Ausgabewert überprüft und in einer Variablen gespeichert.

```
with StoredProc1 do begin
   ParamByName('IN_OUTVAR').AsInteger := 103;
   ExecProc;
   IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Ergebnisparameter verwenden

Zusätzlich zu Ausgabeparametern und einer Datenmenge geben einige Stored Procedures auch einen einzelnen Ergebnisparameter zurück. Dieser wird normalerweise dazu verwendet, einen Fehlerstatus oder die Anzahl der bearbeiteten Datensätze zu signalisieren. In der Dokumentation Ihres Datenbank-Servers finden Sie Informationen, ob und auf welche Weise Ihr Server Ergebnisparameter unterstützt. Ergebnisparametern können keine Werte zugewiesen werden, außer von Stored Procedures, nachdem diese ausgeführt wurden. Um den Wert von Ergebnisparametern abzurufen, überprüfen Sie diese von einer Anwendung aus, nachdem Sie die Methode *TStoredProc.ExecProc* aufgerufen haben.

Verwenden Sie die Eigenschaft *TStoredProc.Params* oder die Methode *TStoredProc.ParamByName*, um das *TParam*-Objekt zu referenzieren, das einen Ergebnisparameter repräsentiert, und um seinen Wert zu überprüfen.

```
DateVar := StoredProc1.ParamByName('MyOutputParam').AsDate;
```

Zur Entwurfszeit auf Parameter zugreifen

Wenn Sie durch Setzen der Eigenschaften *DatabaseName* und *StoredProcName* zur Entwurfszeit eine Verbindung zu einem Remote-Datenbankserver herstellen, können Sie mit Hilfe des Parameter-Editors die Namen und Datentypen jedes Eingabeparameters überprüfen und die Werte der Eingabeparameter festlegen, die beim Ausführen der Stored Procedure an den Server übergeben werden.

Wichtig Ändern Sie die Namen oder Datentypen der vom Server angegebenen Eingabeparameter nicht, da ansonsten beim Ausführen der Stored Procedure eine Exception ausgelöst wird.

Einige Server, wie z.B. Informix, geben grundsätzlich keine Informationen über Parameternamen oder Datentypen aus. Untersuchen Sie in diesen Fällen mit Hilfe des SQL-Explorers oder mit herstellerspezifischen Server-Dienstprogrammen den Quelltext der Stored Procedure des Servers, um die Eingabeparameter und Datentypen zu ermitteln. Weitere Informationen finden Sie in der Online-Hilfe des SQL-Explorers.

Wenn während des Entwurfs keine Parameterliste von der Stored Procedure auf dem Remote-Server empfangen wird (weil z.B. keine Server-Verbindung besteht), müssen Sie den Parameter-Editor aufrufen, die erforderlichen Eingabeparameter angeben und jedem einen Datentyp und einen Wert zuweisen. Weitere Informationen zur Verwendung des Parameter-Editors finden Sie unter »Parameter während des Entwurfs einstellen« auf Seite 22-14.

Parameter während des Entwurfs einstellen

Sie können zur Entwurfszeit den Parameter-Editor aufrufen, um Parameter und ihre Werte festzulegen.

Mit diesem Editor können Sie Parameter für Stored Procedures einrichten. Wenn Sie die Eigenschaften *DatabaseName* und *StoredProcName* der *TStoredProc*-Komponente

zur Entwurfszeit setzen, werden alle vorhandenen Parameter im Parameter-Editor angezeigt. Wenn Sie diese Eigenschaften nicht setzen, werden keine Parameter angezeigt, und Sie müssen diese manuell hinzufügen. Außerdem geben einige Datenbanktypen nicht alle Parameterinformationen (z.B. Datentypen) zurück. Untersuchen Sie in diesen Fällen die Stored Procedures mit dem SQL-Explorer, um die Datentypen zu bestimmen. Konfigurieren Sie dann die Parameter mit dem Parameter-Editor und dem Objektinspektor. Gehen Sie folgendermaßen vor, um Parameter für Stored Procedures zur Entwurfszeit zu setzen:

1. Weisen Sie den Eigenschaften *DatabaseName* und *StoredProcName* Werte zu (optional).
2. Klicken Sie im Objektinspektor auf die Ellipsen-Schaltfläche im Feld *Params*, um den Parameter-Editor aufzurufen.
3. Wenn die Eigenschaften *DatabaseName* und *StoredProcName* nicht gesetzt sind, werden im Editor keine Parameter angezeigt. Fügen Sie die Parameterdefinitionen manuell hinzu, indem Sie im Editor mit der rechten Maustaste klicken und *Hinzufügen* im lokalen Menü wählen.
4. Markieren Sie einzelne Parameter im Parameter-Editor, um ihre Eigenschaften im Objektinspektor anzuzeigen.
5. Wird für die Eigenschaft *ParamType* nicht automatisch ein Typ angezeigt, wählen Sie einen Parametertyp (*Input*, *Output*, *Input/Output* oder *Result*) aus der Dropdown-Liste der Eigenschaft.
6. Wird für die Eigenschaft *DataType* nicht automatisch ein Datentyp angezeigt, wählen Sie einen Datentyp aus der Dropdown-Liste der Eigenschaft. (Um bei der Oracle-Implementierung einer Stored Procedure eine Ergebnismenge zurückzuliefern, setzen Sie den Feldtyp auf *Cursor*.)
7. Verwenden Sie optional die Eigenschaft *Value*, um für einen Eingabe- oder Eingabe-/Ausgabeparameter einen Anfangswert festzulegen.

Wenn Sie mit der rechten Maustaste im Parameter-Editor klicken, wird ein lokales Menü mit Optionen zur Bearbeitung der Parameterdefinitionen angezeigt. Dieses Menü enthält in Abhängigkeit von den angezeigten und ausgewählten Parametern Optionen zum Hinzufügen und Löschen von Parametern, zum Verschieben von Parametern in der Liste und zum Markieren aller angezeigten Parameter.

Sie können die Definition für jedes *TParam*-Objekt ändern, das Sie hinzufügen. Die Attribute dieses Objekts müssen jedoch denen der Parameter für die Stored Procedure auf dem Server entsprechen. Sie bearbeiten das *TParam*-Objekt für einen Parameter, indem Sie es im Parameter-Editor auswählen und die Eigenschaftswerte im Objektinspektor verändern.

Hinweis Sybase-, MS-SQL- und Informix-Server geben grundsätzlich keine Informationen über Parametertypen aus. Mit Hilfe des SQL-Explorers können Sie diese Informationen abrufen.

Hinweis Informix-Server geben grundsätzlich keine Informationen über Datentypen zurück. Mit Hilfe des SQL-Explorers können Sie diese Informationen abrufen.

Hinweis Ausgabe- und Ergebnisparametern können grundsätzlich keine Werte zugewiesen werden. Die Werte für diese Parametertypen werden bei der Ausführung der Stored Procedure zugewiesen.

Parameter zur Laufzeit erstellen

Wenn der Name der Stored Procedure in der Eigenschaft *StoredProcName* erst zur Laufzeit festgelegt wird, werden *TParam*-Objekte für Parameter nicht automatisch erzeugt, sondern müssen zur Laufzeit im Programm erstellt werden. Sie können dafür die Methoden *TParam.Create* oder *TParams.AddParam* verwenden.

Im folgenden Beispiel benötigt die InterBase-Implementierung der Stored Procedure GET_EMP_PROJ einen Eingabeparameter (EMP_NO) und einen Ausgabeparameter (PROJ_ID).

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
   FOR SELECT PROJ_ID
   FROM EMPLOYEE_PROJECT
   WHERE EMP_NO = :EMP_NO
   INTO :PROJ_ID
   DO
    SUSPEND;
END
```

Der folgende Delphi-Quelltext verknüpft diese Stored Procedure mit der *TStoredProc*-Komponente *StoredProc1* und erzeugt mit der Methode *TParam.Create* zwei *TParam*-Objekte für die beiden Parameter:

```
var
   P1, P2: TParam;
begin
   ...
   with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[1].Name := 'PROJ_ID';
      ParamByname('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
   end;
   ...
end;
```

Parameterbindung

Beim Vorbereiten und Ausführen einer Stored Procedure werden ihre Eingabeparameter automatisch an Parameter auf dem Server gebunden.

Mit Hilfe der Eigenschaft *ParamBindMode* können Sie angeben, wie die Parameter der Stored Procedure an Parameter auf dem Server gebunden werden. Standardmäßig hat die Eigenschaft *ParamBindMode* den Wert *pbByName*, was bedeutet, daß die Parameter der Stored Procedure über den Namen mit den Parametern auf dem Server verglichen werden. Dies ist die einfachste Methode, Parameter zu binden.

Einige Server unterstützen das Binden von Parametern über ordinale Werte. In diesem Fall ist jedoch die Reihenfolge von Bedeutung, in der die Parameter im Editor angegeben werden. Der erste definierte Parameter wird dem ersten Eingabeparameter auf dem Server zugeordnet, der zweite Parameter dem zweiten Eingabeparameter usw. Wenn Ihr Server die Parameterbindung über Ordinalwerte unterstützt, können Sie die Eigenschaft *ParamBindMode* auf den Wert *pbByNumber* setzen.

Hinweis Wenn Sie der Eigenschaft *ParamBindMode* den Wert *pbByNumber* zuweisen wollen, müssen die Parametertypen in der richtigen Reihenfolge angegeben werden. Mit Hilfe des SQL-Explorers kann der Quelltext einer Stored Procedure auf dem Server angezeigt werden, um die richtige Reihenfolge und die Parametertypen zu ermitteln.

Parameterinformationen während des Entwurfs anzeigen

Wenn Sie zur Entwurfszeit Zugriff auf einen Datenbank-Server haben, können Sie die Informationen über die von der Stored Procedure verwendeten Parameter auf zwei Arten anzeigen:

- Starten Sie den SQL-Explorer, um den Quelltext einer Stored Procedure auf dem Server zu untersuchen. Im Quelltext sind die Parameterdeklarationen enthalten, die den Datentyp und den Namen jedes Parameters beschreiben.
- Verwenden Sie den Objektinspektor, um Eigenschaftseinstellungen einzelner *TParam*-Objekte anzuzeigen.

Mit Hilfe des SQL-Explorers können Sie die auf Ihrem Datenbank-Server verfügbaren Stored Procedures untersuchen, wenn BDE-Treiber verwendet werden. Bei Verwendung von ODBC-Treibern ist dies nicht der Fall. Obwohl der SQL-Explorer nicht immer zur Verfügung steht, bietet er unter Umständen mehr Informationen über *TParam*-Objekte als der Objektinspektor. Der Umfang der Informationen, die der Objektinspektor über eine Stored Procedure liefert, hängt vom Datenbank-Server ab.

So zeigen Sie einzelne Parameterdefinitionen im Objektinspektor an:

1. Markieren Sie die Prozedurkomponente.
2. Weisen Sie der Eigenschaft *DatabaseName* einer Stored-Procedure-Komponente den BDE-Alias Ihres Datenbank-Servers zu (oder den Wert der Eigenschaft *DatabaseName* einer *TDatabase*-Komponente).

3. Weisen Sie der Eigenschaft *StoredProcName* den Namen der gewünschten Stored Procedure zu.
4. Klicken Sie im Objektinspektor auf die Ellipsen-Schaltfläche der Eigenschaft *TStoredProc.Params*.
5. Markieren Sie im Parameter-Editor einzelne Parameter, um ihre Eigenschaften im Objektinspektor anzuzeigen.

Einige Server geben keine bzw. nicht alle Parameterinformationen aus.

Wenn Sie im Objektinspektor einzelne *TParam*-Objekte anzeigen, können Sie der Eigenschaft *ParamType* entnehmen, ob es sich bei dem betreffenden Parameter um einen Eingabe-, Ausgabe-, Eingabe-/Ausgabe- oder um einen Ergebnisparameter handelt. Die Eigenschaft *DataType* zeigt den Datentyp des Wertes an, den der Parameter enthält, z.B. String, Integer oder Date. Im Eingabefeld *Value* können Sie für den Eingabeparameter einen Wert festlegen.

Hinweis Sybase-, MS-SQL- und Informix-Server geben grundsätzlich keine Informationen über Parametertypen aus. Mit Hilfe des SQL-Explorers können Sie diese Informationen abrufen.

Hinweis Informix-Server geben grundsätzlich keine Informationen über Datentypen zurück. Mit Hilfe des SQL-Explorers können Sie diese Informationen abrufen.

Weitere Informationen hierzu finden Sie unter »Parameter während des Entwurfs einstellen« auf Seite 22-14.

Hinweis Ausgabe- und Ergebnisparametern können keine Werte zugewiesen werden. Die Werte für diese Parametertypen werden bei der Ausführung der Stored Procedure zugewiesen.

Überladene Stored Procedures in Oracle

Oracle-Server ermöglichen das Überladen von Stored Procedures. Der Begriff »überladen« bezeichnet verschiedene Stored Procedures, die alle denselben Namen haben. Anwendungen können diejenige Stored Procedure bestimmen, die ausgeführt werden soll. Dazu dient die Eigenschaft *Overload* der betreffenden Komponente.

Wenn *Overload* den Vorgabewert Null (0) hat, wird kein Überladen vorausgesetzt. Beim Wert Eins (1) führt die Komponente die erste Stored Procedure mit dem überladenen Namen aus, beim Wert Zwei (2) die zweite usw.

Hinweis Überladene Stored Procedures können unterschiedliche Eingabe- und Ausgabeparameter erfordern. Details hierzu finden Sie in der Dokumentation des Oracle-Servers.

Kapitel

23

Mit ADO-Komponenten arbeiten

Die ADO-Komponenten kapseln die ADO-Funktionalität. ADO (Microsoft ActiveX Data Objects) ist eine Gruppe von Datenobjekten, die es einer Anwendung ermöglichen, über einen OLE-DB-Provider auf Daten zuzugreifen. Die ADO-Komponenten von Delphi kapseln die Funktionalität dieser ADO-Objekte und machen sie für Delphi-Komponenten verfügbar. Die am häufigsten auftretenden ADO-Objekte sind die Verbindungs-, Befehls- und Datensatzmengenobjekte. Diese ADO-Objekte sind eine direkte Entsprechung der Komponenten *TADOConnection*, *TADOCommand* und der ADO-Datenmengenkomponenten. Daneben gibt es »Hilfsobjekte« wie *Field* und *Properties*. Diese werden jedoch von Delphi-Programmierern üblicherweise nicht direkt eingesetzt und nicht von speziellen Komponenten repräsentiert.

Die Verwendung von ADO und den ADO-Komponenten ermöglicht Delphi-Programmierern die Erstellung von Datenbankanwendungen, die die Borland Database Engine (BDE) nicht benötigen. Der Zugriff auf die Daten erfolgt statt dessen über ADO.

In diesem Kapitel werden die einzelnen ADO-Komponenten vorgestellt. Weiterhin wird erläutert, inwieweit sie sich von ihren BDE-basierten Gegenstücken unterscheiden. Sie finden außerdem Verweise auf BDE-basierten Verbindungs- und Datenmengenkomponenten, die den ADO-Komponenten entsprechen.

In diesem Kapitel werden die folgenden allgemeinen Bereiche behandelt:

- ADO-Komponenten im Überblick
- Verbindungen zu ADO-Datenspeichern einrichten
- ADO-Datenmengen verwenden
- Befehle ausführen

ADO-Komponenten im Überblick

Delphi verfügt neben den auf der Borland Database Engine (BDE) basierenden Elementen auch über eine Reihe von Komponenten zur Verwendung mit ADO. Diese Komponenten ermöglichen es Programmierern, eine Verbindung zu einem ADO-Datenspeicher einzurichten und dann Befehle aufzurufen sowie Daten aus Tabellen in Datenbanken abzurufen.

Diese ADO-bezogenen Datenzugriffskomponenten benötigen für den Zugriff auf ADO-Datenspeicher und die Arbeit mit Daten lediglich das ADO-Gerüst, die BDE ist hierfür nicht erforderlich. Verwenden Sie die ADO-Komponenten, wenn Ihnen ADO zur Verfügung steht und Sie die BDE nicht einsetzen wollen. ADO 2.1 (oder höher) muß auf dem Host-Computer installiert sein. Zusätzlich muß für das Zieldatenbanksystem (beispielsweise Microsoft SQL Server) Client-Software sowie ein für das Datenbanksystem spezifischer OLE-DB-Treiber oder ODBC-Treiber installiert sein.

Die meisten ADO-Verbindungs- und Datenmengenkomponenten entsprechen den Verbindungs- oder Datenmengenkomponenten der BDE. Die Funktionsweise der Komponente *TADOConnection* entspricht der der Komponente *TDatabase* in BDE-basierten Anwendungen. *TADOTable* ist äquivalent mit *TTable*, *TADOQuery* mit *TQuery* und *TADOStoredProc* mit *TStoredProc*. Sie setzen diese ADO-Komponenten auf die gleiche Weise und im gleichen Kontext wie deren BDE-Gegenstücke ein. Für *TADODataSet* ist keine direkte BDE-Entsprechung vorhanden, die Komponente bietet jedoch viele der Funktionen von *TTable* und *TQuery*. Entsprechend gibt es keine BDE-Komponente, die mit *TADOCommand* vergleichbar ist, da diese in einer Delphi/ADO-Umgebung einem besonderen Zweck dient.

Folgende Klassen bilden die ADO-Komponenten:

Tabelle 23.1 ADO-Komponenten

Komponente	Verwendungszweck
TADOConnection	Diese Komponente wird zur Einrichtung einer Verbindung mit einem ADO-Datenspeicher verwendet. Auf diese Verbindung können gleichzeitig mehrere ADO-Datenmengen- und Befehlskomponenten zur Ausführung von Befehlen, zum Abrufen von Daten und zur Verarbeitung von Metadaten zugreifen.
TADODataSet	Dieses ist die Hauptkomponente für den Zugriff auf und das Verarbeiten von Daten. Sie kann Daten aus einer einzelnen oder mehreren Tabellen abrufen und eine Verbindung zu einem Datenspeicher direkt oder über *TADOConnection* einrichten.
TADOTable	Diese Komponente wird für den Zugriff auf und die Bearbeitung von Datenmengen verwendet, die Daten aus einer einzelnen Tabelle enthalten. Sie kann direkt oder über *TADOConnection* eine Verbindung zu einem Datenspeicher einrichten.
TADOQuery	Diese Komponente wird für den Zugriff auf und die Bearbeitung von Datenmengen verwendet, die durch eine gültige SQL-Anweisung erzeugt wurden. Sie kann auch DDL-SQL-Anweisungen wie CREATE TABLE ausführen. Sie kann direkt oder über *TADOConnection* eine Verbindung zu einem Datenspeicher einrichten.

Tabelle 23.1 ADO-Komponenten (Fortsetzung)

Komponente	Verwendungszweck
TADOStoredProc	Diese Komponente wird zur Ausführung von Stored Procedures verwendet. Sie kann Stored Procedures ausführen, die Daten abrufen oder DDL-Anweisungen ausführen. Sie kann direkt oder über *TADOConnection* eine Verbindung zu einem Datenspeicher einrichten.
TADOCommand	Diese Komponente wird hauptsächlich zur Ausführung von Befehlen (SQL-Anweisungen, die keine Ergebnismengen zurückgeben) eingesetzt. Sie wird zusammen mit einer unterstützenden Datenmengenkomponente verwendet und kann auch eine Datenmenge aus einer Tabelle abrufen. Sie kann direkt oder über *TADOConnection* eine Verbindung zu einem Datenspeicher einrichten.

Verbindungen zu ADO-Datenspeichern einrichten

Eine Anwendung muß zunächst eine Verbindung zu einem Datenspeicher herstellen, bevor Befehle ausgeführt oder Daten abgerufen werden können. Jede ADO-Befehls- und Datenmengenkomponente in einer Anwendung kann ihre eigene Verbindung einrichten, durch den Einsatz von *TADOConnection* kann jedoch eine einzelne Verbindung von mehreren ADO-Komponenten genutzt werden.

Beim Aufbau einer Verbindung von ADO-Befehls- und Datenmengenkomponenten zu einem Datenspeicher können diese eine Verbindung gemeinsam nutzen oder jeweils eine eigene Verbindung einrichten. Jedes dieser Verfahren hat bestimmte Vor- und Nachteile.

In diesem Abschnitt werden die Schritte beschrieben, die zur Einrichtung und Verwendung einer Verbindung zu einem ADO-Datenspeicher durchgeführt werden müssen.

Mit TADOConnection eine Verbindung zu einem Datenspeicher einrichten

ADO-Datenmengen- und Befehlskomponenten können gemeinsam auf eine einzelne Verbindung zu einem Datenspeicher zugreifen. Um dies zu ermöglichen, muß in der Anwendung eine *TADOConnection*-Komponente für jede Datenspeicherverbindung vorhanden sein. Die Datenmengen- und Befehlskomponenten werden dann über ihre Eigenschaft *Connection* der Verbindungskomponente zugeordnet.

Neben den Mitteln, die Datensatz- und Befehlskomponenten die Einrichtung von Verbindungen zu Datenspeichern ermöglichen, enthalten Verbindungskomponenten Eigenschaften und Methoden zum Aktivieren und Deaktivieren der Verbindung, für den direkten Zugriff auf das ADO-Verbindungsobjekt und zur Ermittlung der Aktivität (falls vorhanden) einer Verbindungskomponente zu einem bestimmten Zeitpunkt.

TADOConnection im Vergleich mit der Eigenschaft ConnectionString einer Datenmenge

Jede ADO-Befehls- und Datenmengenkomponente in einer Anwendung kann eine direkte Verbindung zu einem Datenspeicher einrichten. Sind jedoch zahlreiche Befehls- und Datenmengenkomponenten vorhanden, ist es häufig einfacher, mit Hilfe von *TADOConnection* nur eine Verbindung einzurichten, auf die die Befehls- und Datenmengenkomponenten dann gemeinsam zugreifen können. Weitere Informationen zur direkten Verbindung einzelner Befehls- und Datenmengenkomponenten mit einem Datenspeicher finden Sie im Abschnitt »Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen« auf Seite 23-15.

Die Verwendung einer *TADOConnection*-Komponente gibt Ihnen, im Vergleich zur Einrichtung einzelner Verbindungen für Befehls- und Datenmengenkomponenten, mehr Steuerungsmöglichkeiten für die Verbindung. Diese wird durch die Eigenschaften, Methoden und Ereignisse von *TADOConnection* ermöglicht, deren Funktionalität andernfalls nicht verfügbar ist.

Verbindungen festlegen

Bevor eine *TADOConnection*-Komponente zur Bereitstellung einer gemeinsam genutzten Verbindung für ADO-Datenmengen und -Befehlskomponenten eingesetzt werden kann, muß zunächst die Verbindung eingerichtet werden. Geben Sie hierzu die spezifischen Verbindungsinformationen in der Eigenschaft *ConnectionString* der Verbindungskomponente an. Zur Entwurfszeit öffnen Sie das Dialogfeld des Verbindungsstringeditors, indem Sie im Objektinspektor auf die Ellipsenschaltfläche neben der Eigenschaft ConnectionString klicken. In diesem vom ADO-System bereitgestellten Dialogfeld können Sie einen Verbindungsstring interaktiv erstellen, indem Sie Verbindungselemente (wie Provider und Server) in Listen auswählen. Zur Laufzeit weisen Sie der Eigenschaft *ConnectionString* einen String mit den Verbindungsinformationen als Wert zu. Wird der Eigenschaft *Connected* der Verbindungskomponente der Wert *True* zugewiesen, wird die Verbindung aktiviert. Dies ist jedoch zu diesem Zeitpunkt noch nicht erforderlich. Zur Entwurfszeit kann dieses Verfahren als Verbindungstest genutzt werden.

```
ADOConnection1.ConnectionString := 'Provider=ProviderName;Remote Server=ServerReferenz';
```

Die Eigenschaft *ConnectionString* kann mehrere Verbindungsparameter enthalten, die jeweils durch ein Semikolon voneinander getrennt sind. Diese Parameter können den Namen des Providers, einen Benutzernamen und ein Kennwort (für Anmeldezwekke) sowie einen Verweis auf einen Remote-Server enthalten.

Die Eigenschaft *ConnectionString* kann auch den Namen einer Datei enthalten, in der die Verbindungsparameter definiert sind. Eine solche Datei hat denselben Inhalt wie die Eigenschaft *ConnectionString*: einen oder mehrere Parameter, denen jeweils ein Wert zugewiesen ist und die durch ein Semikolon voneinander getrennt sind. Eine Liste der von ADO unterstützten Parameter finden Sie in der VCL-Hilfe zur Eigenschaft *ConnectionString*.

Sobald die Verbindungsinformationen in der Verbindungskomponente vorhanden sind, verknüpfen Sie die Datenmengen- und Befehlskomponenten mit der Verbindungskomponente. Sie erreichen dies, indem Sie der Eigenschaft *Connection* jeder Da-

tenmengen- oder Befehlskomponente eine Referenz auf die Verbindungskomponente hinzufügen. Zur Entwurfszeit können Sie die gewünschte Verbindungskomponente in der Dropdown-Liste der Eigenschaft *Connection* im Objektinspektor auswählen. Zur Laufzeit weisen Sie die Referenz der Eigenschaft *Connection* zu. Der nachfolgende Befehl beispielsweise verknüpft eine *TADODataSet*-Komponente mit einer *TADOConnection*-Komponente.

```
ADODataSet1.Connection := ADOConnection1;
```

Sie können die Verbindung explizit aktivieren, indem Sie der Eigenschaft *Connected* der Verbindungskomponente den Wert *True* zuweisen. Sie wird jedoch automatisch aktiviert, wenn die erste Datenmengenkomponente aktiviert oder der erste Befehl mit einer Befehlskomponente ausgeführt wird.

Auf das Verbindungsobjekt zugreifen

Verwenden Sie die Eigenschaft *ConnectionObject* von *TADOConnection*, um direkt auf das zugrundeliegende ADO-Verbindungsobjekt zuzugreifen. Auf diese Weise können Sie von einer Anwendung aus auf Eigenschaften des zugrundeliegenden ADO-Verbindungsobjekts zugreifen und dessen Methoden aufrufen.

Die Verwendung von *ConnectionObject* für den direkten Zugriff auf das zugrundeliegende ADO-Verbindungsobjekt erfordert gute Kenntnisse hinsichtlich der Funktionsweise von ADO-Objekten im allgemeinen und von ADO-Verbindungsobjekten im besonderen. Sie sollten nur dann direkt auf das Verbindungsobjekt zugreifen, wenn Sie mit dessen Funktionen vertraut sind. Informationen zum Einsatz von ADO-Verbindungsobjekten finden Sie in der Hilfe von Microsoft Data Access SDK.

Verbindungen aktivieren und deaktivieren

Zum Aktivieren einer ADO-Verbindungskomponente setzen Sie die Eigenschaft *TADOConnection.Active* auf *True*, oder rufen die Methode *TADOConnection.Open* auf.

```
ADOConnection1.Active := True;
```

Damit die Verbindung erfolgreich eingerichtet werden kann, muß über die in der Eigenschaft *TADOConnection.ConnectionString* angegebenen Informationen eine gültige Verbindung definiert sein. Weitere Informationen zur Bereitstellung von Verbindungsinformationen finden Sie im Abschnitt »Verbindungen festlegen« auf Seite 23-4.

Das Aktivieren einer ADO-Verbindungskomponente löst die Ereignisse *OnWillConnect* und *OnConnectComplete* der ADO-Verbindungskomponente aus und führt die Behandlungsroutinen für diese Ereignisse aus, falls diese zugewiesen wurden.

War eine Verbindungskomponente zuvor noch nicht aktiviert, geschieht dies automatisch, wenn auf eine zugeordnete Datenmengen- oder Befehlskomponente zugegriffen wird. Datenmengenkomponenten lösen diesen Vorgang aus, wenn sie aktiviert werden, Befehlskomponenten, wenn ein Befehl ausgeführt wird. Weitere Informationen zum Verbinden von Datenmengenkomponenten und Verbindungskomponenten finden Sie im Abschnitt »Mit TADOConnection eine Verbindung zu einem Datenspeicher einrichten« auf Seite 23-3.

Setzen Sie zum Deaktivieren einer ADO-Verbindungskomponente entweder ihre Eigenschaft *Active* auf *False*, oder rufen Sie ihre Methode *Close* auf.

```
ADOConnection1.Close;
```

Das Deaktivieren einer Verbindungskomponente mit Hilfe der Eigenschaft *Active* oder der Methode *Close* löst die folgenden vier Aktionen aus:

1. Das Ereignis *TADOConnection.OnDisconnect* wird ausgelöst.
2. Die Ereignisbehandlungsroutine für *OnDisconnect* wird ausgeführt (falls sie definiert wurde).
3. Die Komponente *TADOConnection* wird deaktiviert.
4. Alle zugehörigen ADO-Befehls- oder Datenmengenkomponenten werden deaktiviert.

Den Status einer Verbindungskomponente abrufen

Solange eine *TADOConnection*-Komponente existiert, können Sie jederzeit über deren Eigenschaft *State* ermitteln, ob eine bzw. welche Aktivität aktuell von der Verbindungskomponente durchgeführt wird.

Der Wert *stClosed* für *TObjectStates* in der Eigenschaft *TADOConnection.State* gibt an, daß das Verbindungsobjekt aktuell nicht aktiv ist. Die Eigenschaft *TADOConnection.Active* enthält den Wert *False*, und auch alle zugeordneten Befehls- oder Datenmengenkomponenten sind nicht aktiv.

Der Wert *stOpen* zeigt an, daß die Verbindungskomponente aktiv ist, d. h. der Aufbau einer Verbindung mit einem ADO-Datenspeicher erfolgreich war. Die Eigenschaft *Active* enthält den Wert *True*, und zugeordnete Befehls- oder Datenmengenkomponenten sind möglicherweise aktiv.

Der Wert *stConnecting* gibt an, daß die Verbindungskomponente aktuell versucht, eine Verbindung zu dem in der Eigenschaft *TADOConnection.ConnectionString* definierten ADO-Datenspeicher einzurichten. Solange dieser Status besteht, kann die Methode *Cancel* aufgerufen werden, um den Verbindungsversuch abzubrechen.

Verbindungen optimieren

Wenn für die ADO-Befehls- und Datenmengenkomponenten einer Anwendung eine *TADOConnection*-Komponente zur Einrichtung einer Verbindung mit einem Datenspeicher verwendet wird, können Sie erweiterte Steuerungsmöglichkeiten hinsichtlich verschiedener Bedingungen und Attribute der Verbindung nutzen. Diese Aspekte werden durch Eigenschaften und Ereignisbehandlungsroutinen von *TADOConnection* zur Optimierung einer Verbindung implementiert.

Verbindungsattribute angeben

Verwenden Sie die Eigenschaft *TADOConnection.ConnectOption*s, um optional festzulegen, daß eine asynchrone Verbindung eingerichtet werden soll. Die Standardeinstellung für *ConnectionOptions* ist *coConnectUnspecified*. Diese Einstellung überläßt dem Server die Entscheidung über den besten Verbindungstyp. Um explizit eine asynchrone Verbindung einzurichten, weisen Sie *ConnectOptions* den Wert *coAsyncConnect* zu.

Um eine asynchrone Verbindung einzurichten oder die Entscheidung an den Server zu übertragen, weisen Sie der Eigenschaft *ConnectOptions* der Verbindungskomponente eine der *TConnectOption*-Konstanten zu. Aktivieren Sie dann die Verbindungskomponente, indem Sie deren Methode *Open* aufrufen, der Eigenschaft *Connected* den Wert *True* zuweisen oder indem Sie eine der zugeordneten Befehls- oder Datenmengenkomponenten aktivieren. Die folgenden Beispielroutinen aktivieren bzw. deaktivieren asynchrone Verbindungen für die angegebene Verbindungskomponente.

```
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
  end;
end;

procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
```

Verwenden Sie die Eigenschaft *TADOConnection.Attribute*s, um die Verarbeitung ausstehender *Commit*- und *Abort*-Operationen für die Verbindungskomponente zu steuern. Attributes kann eine, beide oder keine der Konstanten *xaCommitRetaining* und *xaAbortRetaining* enthalten. Durch die Verwendung derselben Eigenschaft schließt sich die Steuerung von ausstehenden *Commit*-Operationen und ausstehenden Abort-Operationen wechselseitig aus.

Durch Verwendung des Operators *in* mit einer der Konstanten können Sie überprüfen, welche der Optionen aktiviert ist. Sie aktivieren eine Option, indem Sie die Konstante in die Eigenschaft *Attributes* einfügen. Zum Deaktivieren entfernen Sie die Konstante. Die folgenden Beispielroutinen aktivieren bzw. deaktivieren ausstehende Commit-Operationen für die angegebene Verbindungskomponente.

```
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
```

```
  end;
end;

procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
```

Timeouts steuern

Sie steuern die Zeitspanne, nach deren Ablauf der Versuch, Befehle auszuführen oder Verbindungen einzurichten, als fehlgeschlagen interpretiert und somit abgebrochen wird, über die Eigenschaften *TADOConnection.ConnectionTimeout* und *TADOConnection.CommandTimeout*.

ConnectionTimeout legt den Zeitraum fest, nach dessen Ablauf der Versuch abgebrochen wird, eine Verbindung zu einem Datenspeicher einzurichten. Kann die von der Methode *Open* aufgerufene Verbindung vor Ablauf der in *ConnectionTimeout* festgelegten Zeitspanne nicht hergestellt werden, wird die Einrichtung der Verbindung abgebrochen. Legen Sie für *ConnectionTimeout* die Anzahl von Sekunden fest, nach deren Ablauf die Einrichtung der Verbindung abgebrochen werden soll.

```
with ADOConnection1 do begin
  ConnectionTimeout = 10 {Sekunden};
  Open;
end;
```

CommandTimeout gibt den Zeitraum an, nach dessen Ablauf der Versuch abgebrochen wird, einen Befehl auszuführen. Kann der von der Methode *Execute* aufgerufene Befehl nicht vor dem Ende des in *CommandTimeout* angegebenen Zeitraums erfolgreich ausgeführt werden, wird der Befehl verworfen, und ADO generiert eine Exception. Legen Sie für *CommandTimeout* die Anzahl von Sekunden fest, nach deren Ablauf der Versuch abgebrochen werden soll, einen Befehl auszuführen.

```
with ADOConnection1 do begin
   CommandTimeout = 10 {Sekunden};
   Execute('DROP TABLE Employee1997', []);
end;
```

Die Anmeldung für eine Verbindung steuern

Der Versuch, eine Verbindung zu einem Datenspeicher mit einer Verbindungskomponente herzustellen, löst das Ereignis *OnLogin* für die sichere Anmeldung aus. Ein Hinweis auf dieses Ereignis ist die Anzeige eines Dialogfelds, in dem der Benutzer zur Anmeldung einen Benutzernamen und ein Kennwort angeben muß. Die Anzeige dieses Dialogfelds kann, falls gewünscht, umgangen werden, indem die Angabe von Benutzernamen und Kennworten im Quelltext erfolgt.

Um die Anzeige des Standarddialogfelds zu unterdrücken, weisen Sie zunächst der Eigenschaft *LoginPrompt* der Verbindungskomponente den Wert *False* zu. Geben Sie dann vor dem Aktivieren der Verbindungskomponente alle für die Anmeldung erforderlichen Informationen über ein für die Übertragung geeignetes Medium an, beispielsweise über die Eigenschaft *ConnectionString*.

```
with ADOConnection1 do begin
  Close;
  LoginPrompt := False;
  ConnectionString := 'Provider=NameDesProviders;Remote Server=NameDesServers;' +
    'User Name=JaneDoe;Password='Geheim';
  Connected := True;
end;
```

Die Anmeldungsinformationen können auch als Parameter der Methode *Open* der Verbindungskomponente an den Datenspeicher übertragen werden.

```
with ADOConnection1 do begin
  Close;
  LoginPrompt := False;
  ConnectionString := 'Provider=NameDesProviders;Remote Server=NameDesProviders';
  Open('JaneDoe', 'Geheim');
end;
```

Die Funktion der beiden dargestellten Routinen ist identisch. Der Unterschied besteht darin, daß in der zweiten Routine der Benutzername und das Kennwort nicht in der Eigenschaft *ConnectionString* ausgedrückt werden. Sie werden vielmehr als Parameter der Methode *Open* übertragen. Dieses Verfahren ist insbesondere dann sinnvoll, wenn Verbindungsinformationen (wie Provider und Server) für alle Benutzer gleich und nur die benutzerspezifischen Informationen unterschiedlich sind. Die Anwendung kann beispielsweise die Benutzerdaten über ein benutzerdefiniertes Anmeldedialogfeld abrufen und die Provider- und Serverinformationen einer statischen Quelle entnehmen, beispielsweise der Registrierung von Windows.

Schlägt die Anmeldung nach der Übertragung der Anmeldungsinformationen fehl, wird eine Exception des Typs *EOleException* ausgelöst.

Tabellen und Stored Procedures abrufen

Die Komponente *TADOConnection* verfügt über Eigenschaften, mit deren Hilfe Sie Listen mit den über diese Verbindung zur Verfügung stehenden Tabellen und Stored Procedures abrufen können. Weitere Eigenschaften dieser Komponente ermöglichen Ihnen den Zugriff auf die Datenmengen- und Befehlskomponenten, die mit der Verbindungskomponente verknüpft sind.

Auf die Datenmengen einer Verbindung zugreifen

Die Eigenschaften *DataSets* und *DataSetCoun*t von *TADOConnection* ermöglichen einem Programm, jede Datenmengenkomponente, die mit einer Verbindungskomponente verknüpft ist, sequentiell zu referenzieren. Datenmengenkomponenten mit den Eigenschaften *DataSets* und *DataSetCount* sind beispielsweise *TADODataSet*,

TADOQuery und *TADOStoredProc*. Arbeiten Sie mit den Befehlskomponenten einer Verbindung, verwenden Sie die Eigenschaften *Commands* und *CommandCount*.

DataSets ist ein Array mit Referenzen auf ADO-Datenmengenkomponenten, dessen Zählung bei Null beginnt. Verwenden Sie für *DataSets* eine Indexzahl, die die Position einer bestimmten Datenmenge innerhalb des Arrays angibt. Verwenden Sie beispielsweise die Indexzahl 3, um auf die vierte Datenmengenkomponente in *DataSets* zu verweisen.

```
ShowMessage(ADOConnection1.DataSets[3].Name);
```

Da *DataSets* eine Referenz vom Typ *TCustomADODataSet* enthält, können Sie für diese Referenz eine Typumwandlung durchführen. Definieren Sie diese Referenz als Typ einer untergeordneten Klasse, um auf die Eigenschaften und Methoden zugreifen zu können, die nur in dieser zur Verfügung stehen. So verfügt beispielsweise *TCustomADODataSet* nicht über die Eigenschaft *SQL*, die untergeordnete Klasse *TADOQuery* hingegen schon. Wollen Sie also auf die Eigenschaft *SQL* der Datenmenge zugreifen, auf die in der Eigenschaft *DataSets* verwiesen wird, weisen Sie dieser den Typ *TADOQuery* zu.

```
with (ADOConnection1.DataSets[10] as TADOQuery do begin
  SQL.Clear;
  SQL.Add('SELECT * FROM Species');
  Open;
end;
```

Die Eigenschaft *DataSetCount* stellt die Gesamtanzahl aller mit einer Verbindungskomponente verknüpften Datenmengen zur Verfügung. Sie können diese Eigenschaft als Grundlage einer Schleife verwenden, die sequentiell alle mit einer Verbindung verknüpften Datenmengenkomponenten aufruft.

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection4.DataSetCount) do
    ADOConnection4.DataSets[i].Open;
end;
```

Auf die Befehle einer Verbindung zugreifen

Die Funktionsweise der Eigenschaften *Commands* und *CommandCount* von *TADOConnection* entspricht in weiten Teilen der Funktionsweise der Eigenschaften *DataSets* und *DataSetCount*. Der Unterschied besteht darin, daß *Commands* und *CommandCount* Referenzen auf alle mit der Verbindungskomponente verknüpften *TADOCommand*-Komponenten bereitstellen. Um mit allen Datenmengenkomponenten einer Verbindung zu arbeiten, verwenden Sie die Eigenschaften *DataSets* und *DataSetCount*.

Commands ist ein Array mit Referenzen auf ADO-Befehlskomponenten, dessen Zählung bei Null beginnt. Verwenden Sie für *Commands* eine Indexzahl, die die Position eines bestimmten Befehls innerhalb des Arrays angibt. Verwenden Sie beispielsweise die Indexzahl 1, um auf die zweite Befehlskomponente in *Commands* zu verweisen.

```
Memo1.Lines.Text := ADOConnection1.Commands[1].CommandText;
```

Die Eigenschaft *CommandCount* stellt die Gesamtanzahl aller mit einer Verbindungskomponente verknüpften Befehle zur Verfügung. Sie können diese Eigenschaft als Grundlage einer Schleife verwenden, die sequentiell alle mit einer Verbindung verknüpften Befehlskomponenten aufruft.

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount) do
    ADOConnection1.Commands[i].Execute;
end;
```

Verfügbare Tabellen abrufen

Verwenden Sie die Methode *GetTableNames*, um eine Liste mit allen in der Datenbank enthaltenen Tabellen zu erstellen, auf die über das Verbindungsobjekt zugegriffen werden soll. Diese Methode kopiert eine Liste der Tabellennamen in ein bereits vorhandenes Stringlistenobjekt. Sie können einzelne Elemente beispielsweise als Wert für die Eigenschaft *TableName* einer *TADOTable*-Komponente oder als Name einer Tabelle in einer von *TADOQuery* ausgeführten SQL-Anweisung verwenden.

```
ADOConnection1.GetTableNames(ListBox1.Items, False);
```

Das nachfolgende Beispiel durchläuft eine Liste mit Tabellennamen, die mit der Methode *GetTableNames* erstellt wurde. Die Routine nimmt für jede Tabelle einen Eintrag in einer anderen Tabelle vor, der aus dem Namen der Tabelle und der Anzahl der enthaltenen Datensätze besteht.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SL: TStrings;
  index: Integer;
begin
  SL := TStringList.Create;
  try
    ADOConnection1.GetTableNames(SL, False);
    for index := 0 to (SL.Count - 1) do begin
      Table1.Insert;
      Table1.FieldByName('Name').AsString := SL[index];
      ADOTable1.TableName := SL[index];
      ADOTable1.Open;
      Table1.FieldByName('Records').AsInteger :=
        ADOTable1.RecordCount;
      Table1.Post;
    end;
  finally
    SL.Free;
    ADOTable1.Close;
  end;
end;
```

Verfügbare Stored Procedures abrufen

Verwenden Sie die Methode *GetProcedureNames*, um eine Liste mit allen Stored Procedures in der Datenbank zu erstellen, auf die über das Verbindungsobjekt zugegriffen wird. Diese Methode kopiert eine Liste der Stored Procedures in ein bereits vorhandenes Stringlistenobjekt. Einzelne Elemente dieser Liste können beispielsweise als Wert für die Eigenschaft *ProcedureName* einer *TADOStoredProc*-Komponente verwendet werden.

```
ADOConnection1.GetProcedureNames(ListBox1.Items);
```

Im nachfolgenden Beispiel wird eine über *GetProcedureNames* abgerufene Liste zur Ausführung aller Stored Procedures in der zugeordneten Datenbank verwendet.

```
procedure TDataForm.ExecuteProcsButtonClick(Sender: TObject);
var
  SL: TStrings;
  index: Integer;
begin
  SL := TStringList.Create;
  try
    ADOConnection1.GetProcedureNames(SL);
    if (SL.Count > 0) then
      for index := 0 to (SL.Count - 1) do begin
        ADOStoredProc1.ProcedureName := SL[index];
        ADOStoredProc1.ExecProc;
      end;
  finally
    SL.Free;
  end;
end;
```

Mit Verbindungstransaktionen arbeiten

Die Komponente *TADOConnection* enthält eine Reihe von Methoden und Ereignissen für die Arbeit mit Transaktionen. Auf diese können die Befehls- und Datensatzkomponenten gemeinsam zugreifen, die auch die Datenspeicherverbindung gemeinsam nutzen.

Transaktionsmethoden verwenden

Verwenden Sie die Methoden *BeginTrans*, *CommitTrans* und *RollbackTrans* zur Durchführung von Transaktionsverarbeitungen. *BeginTrans* beginnt eine Transaktion in dem der ADO-Verbindungskomponente zugeordneten Datenspeicher. CommitTrans überträgt eine aktuell aktive Transaktion, speichert Änderungen in der Datenbank und beendet die Transaktion. *RollbackTrans* bricht eine aktuell aktive Transaktion ab, verwirft alle im Rahmen der Transaktion durchgeführten Änderungen und beendet die Transaktion. Sie können der Eigenschaft *InTransaction* zu jedem Zeitpunkt entnehmen, ob für die Verbindungskomponente eine nicht abgeschlossene Transaktion vorliegt.

Eine von der Verbindungskomponente aufgerufene Transaktion können alle Befehls- und Datenmengenkomponenten, die auf die von der *TADOConnection*-Komponente eingerichtete Verbindung zugreifen, gemeinsam nutzen.

Transaktionsereignisse verwenden

Die ADO-Verbindungskomponente stellt eine Reihe von Ereignissen zur Verfügung, mit deren Hilfe ermittelt werden kann, wann transaktionsbezogene Prozesse abgeschlossen sind. Diese Ereignisse geben an, wann eine von den Methoden *BeginTrans*, *CommitTrans* oder *RollbackTrans* ausgelöste Transaktion erfolgreich im Datenspeicher abgeschlossen ist.

Das Ereignis *OnBeginTransComplete* wird ausgelöst, wenn der Datenspeicher eine Transaktion erfolgreich begonnen hat, nachdem die Methode *BeginTrans* der Verbindungskomponente aufgerufen wurde. Das Ereignis *OnCommitTransComplete* wird ausgelöst, wenn eine Transaktion nach dem Aufruf von *CommitTrans* erfolgreich eingetragen wurde. *OnRollbackTransComplete* wird ausgelöst, sobald eine Transaktion nach dem Aufruf von *RollbackTrans* erfolgreich eingetragen wurde.

ADO-Datenmengen verwenden

Die in Delphi zur Verfügung stehenden ADO-Datenmengenkomponenten entsprechen den BDE-Datenmengenkomponenten. So entspricht beispielsweise die Funktionsweise der Komponente *TADOTable* der der Komponente *TTable*. Der wesentliche Unterschied besteht darin, daß die ADO-Datenmengenkomponenten für den Datenzugriff zugrundeliegende ADO-Objekte verwenden und hierfür die Borland Database Engine nicht benötigen.

Die Klasse *TDataSet* ist der gemeinsame Vorfahr der ADO- und der BDE-Datenmengenkomponenten. Aus diesem Grund verfügen sie über gleiche Funktionsweisen hinsichtlich der geerbten oder ähnlichen Eigenschaften, Methoden und Ereignisse. In diesem Abschnitt werden speziell die Bereiche der ADO-Datenmengenkomponenten erläutert, die sich von den entsprechenden allgemeinen Datenmengenkomponenten unterscheiden. Weitere Informationen zur gemeinsamen Funktionalität beider Gruppen von Datenmengenkomponenten finden Sie in den Beschreibungen der allgemeinen und BDE-basierten Datenmengenkomponenten:

- Kapitel 18, »Datenmengen«
- Kapitel 20, »Tabellen«
- Kapitel 21, »Abfragen«
- Kapitel 22, »Stored Procedures«

Dieser Abschnitt enthält Informationen zu den Aspekten, in denen sich die Funktionsweise der ADO-Versionen der Datenmengenkomponenten von der ihrer allgemeinen Gegenstücke unterscheidet. Diese Informationen sind in die folgenden Bereiche aufgeteilt:

- Gemeinsame Merkmale aller ADO-Datenmengenkomponenten

- TADODataSet verwenden
- TADOTable verwenden
- TADOQuery verwenden
- TADOStoredProc verwenden

Gemeinsame Merkmale aller ADO-Datenmengenkomponenten

Bestimmte Aspekte der ADO-Datenmengenkomponenten haben in allen unterschiedlichen Komponenten dieselbe Funktionsweise. Außer bei den gesondert aufgeführten werden diese Funktionsbereiche auf genau dieselbe Weise eingesetzt, unabhängig davon, welche ADO-Datenmengenkomponente verwendet wird.

Daten ändern

Der Zugriff auf Spalten in ADO-Datenmengenkomponenten und die Änderung von Daten erfolgt auf die gleiche Weise wie bei allgemeinen Datenmengenkomponenten. So verwenden Sie beispielsweise die Datenmengenmethoden *Edit* und *Inser*t, um den Bearbeitungsmodus für die Datenmenge zu aktivieren, bevor Sie Daten ändern. Sie setzen die Methode *Post* ein, um Datenänderungen abzuschließen.

Verwenden Sie die dynamischen *TField*-Referenzen, die über die Eigenschaft *Fields* und die Methode *FieldByname* der Datenmengenkomponenten bereitgestellt werden. Arbeiten Sie von dort aus mit den Eigenschaften und Methoden der Klasse *TField* und deren untergeordneten Klassen, um beispielsweise einer Spalte einen Wert zuzuweisen bzw. einen Wert aus ihr abzurufen, die Gültigkeit von Daten zu prüfen oder den Datentyp einer Spalte zu ermitteln.

Informationen zum Ändern von Daten mit Hilfe von Datenmengenkomponenten finden Sie im Abschnitt »Modifying data« auf Seite 18-31. Informationen zur Verwendung von Tabellenspalten und persistenten Feldobjekten finden Sie im Abschnitt »Working with field components« auf Seite 19-1.

In Datenmengen navigieren

Sie gelangen in ADO-Datenmengen auf die gleiche Weise von einem Datensatz zu einem anderen wie in allgemeinen Datenmengenkomponenten. Verwenden Sie Methoden wie *First, Next, Last* und *Prior*, um den Datensatzzeiger in der Datenmengenkomponente auf einen anderen Datensatz zu verschieben. Für Schleifen können die Eigenschaften *Eof* und *Bof* verwendet werden, so daß alle Zeilen einer Datenmenge verarbeitet werden.

```
ADOTable1.First;
while not ADOTable1.Eof do begin
  { Jeden Datensatz hier bearbeiten }
  ⋮
  ADOTable1.Next;
end;
```

Weitere Informationen zum Navigieren in Datenmengenkomponenten finden Sie im Abschnitt »Navigating datasets« auf Seite 18-15.

Visuelle datensensitive Steuerelemente verwenden

Die Datenmenge einer ADO-Datenmengenkomponente kann in einer Anwendung mit Hilfe von datensensitiven Steuerelementen verfügbar gemacht werden. Diese Datenmengen enthalten beispielsweise die von einer *TADOTable*-Komponente zurückgegebenen Datensätze, die von einer SELECT-Anweisung in einer *TADOQuery*-Komponente zurückgegebene Ergebnismenge oder von einer *TADOStoredProc*-Komponente ausgeführte Stored Procedures, die eine Ergebnismenge zurückgeben.

So machen Sie diese Datenmengen über datensensitive Steuerelemente verfügbar:

1. Verwenden Sie eine *TDataSource*-Standardkomponente.
2. Legen Sie in der Eigenschaft *DataSet* eine aktive ADO-Datenmengenkomponente fest.
3. Verwenden Sie datensensitive Standardsteuerelemente wie *TDBEdit* und *TDBGrid.*
4. Legen Sie in der Eigenschaft *DataSource* des datensensitiven Steuerelements die Komponente *TDataSource* fest.

Im Quelltext kann eine solche Verbindung zwischen ADO-Datenmengenkomponente, Datenquellenkomponente und datensensitivem Steuerelement beispielsweise so hergestellt werden:

```
DBGrid1.DataSource := DataSource1;
DataSource1.DataSet := ADOQuery1;
ADOQuery1.Open;
```

Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen

ADO-Datenmengenkomponenten können entweder zusammen oder einzeln mit einem ADO-Datenspeicher verbunden werden.

Um eine gemeinsame Verbindung für Datenmengenkomponenten einzurichten, legen Sie für die Eigenschaft *Connection* jeder Datenmengenkomponente eine *TADOConnection*-Komponente fest. In diesem Fall greifen alle Datenmengenkomponenten über die von dieser Verbindungskomponente eingerichteten Verbindung auf den Datenspeicher zu.

```
ADODataSet1.Connection := ADOConnection1;
ADODataSet2.Connection := ADOConnection1;
...
```

Einige der Vorteile des gemeinsamen Verbindens von Datenmengenkomponenten:

- Die Datenmengenkomponenten können auf die Attribute des Verbindungsobjekts zugreifen.
- Es muß nur eine Verbindung eingerichtet werden: die von *TADOConnection.*

- Die Datenmengenkomponenten können in Transaktionen genutzt werden.

Wenn Sie einzelne Verbindungen für Datenmengenkomponenten einrichten, müssen Sie der Eigenschaft *ConnectionString* jeder Datenmengenkomponente einen entsprechenden Wert zuweisen. Die für die Einrichtung einer Verbindung zu einem Datenspeicher erforderlichen Informationen müssen für jede Datenmengenkomponente gesondert festgelegt werden. Jede Datenmengenkomponente richtet dann ihre eigene Verbindung zu dem Datenspeicher ein, völlig unabhängig von anderen Datenmengenverbindungen in der Anwendung.

```
ADODataSet1.ConnectionString := 'Provider=IhrProvider;Password=Geheim;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=Geheim;' +
  'Initial Catalog=Employee';
ADODataSet2.ConnectionString := 'Provider=IhrProvider;Password=Geheim;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=Geheim;' +
  'Initial Catalog=Employee';
...
```

Weitere Informationen zur Verwendung von *TADOConnection* zur Einrichtung einer Verbindung zu einem Datenspeicher finden Sie im Abschnitt »Mit TADOConnection eine Verbindung zu einem Datenspeicher einrichten« auf Seite 23-3.

Mit Datensatzmengen arbeiten

Neben den Mitteln zum Navigieren in Datensätzen und zum Ändern von Daten, die von allen ADO-Datenmengenkomponenten verwendet werden können, gibt es zusätzliche Eigenschaften und Methoden zur Arbeit mit Datensatzmengen.

Verwenden Sie die Eigenschaft *RecordSet* für den direkten Zugriff auf das der Datenmengenkomponente zugrundeliegende ADO-Datensatzmengenobjekt. Auf diese Weise können Sie von einer Anwendung aus auf Eigenschaften und Aufrufmethoden des Datensatzmengenobjekts zugreifen. Die Verwendung von *RecordSet* für den direkten Zugriff auf das zugrundeliegende ADO-Datensatzmengenobjekt setzt umfassende Kenntnisse hinsichtlich der Funktionsweise von ADO-Objekten im allgemeinen und von ADO-Datensatzmengenobjekten im besonderen voraus. Sie sollten nur dann direkt auf das Datensatzmengenobjekt zugreifen, wenn Sie mit dessen Funktionen vertraut sind. Informationen zum Einsatz von ADO-Datensatzmengenobjekten finden Sie in der Hilfe von Microsoft Data Access SDK.

Verwenden Sie die Eigenschaft *RecordSetState*, um den aktuellen Status der Datenmengenkomponente zu ermitteln. *RecordSetState* implementiert die Eigenschaft *State* des ADO-Datensatzmengenobjekts und gibt so den aktuellen Status des zugrundeliegenden Datensatzmengenobjekts an. Die Eigenschaft *RecordSetState* enthält einen der Werte *stExecuting* oder *stFetching*. Der Wert *stExecuting* gibt an, daß die Datenmengenkomponente zur Zeit einen Befehl ausführt. Der Wert *stFetching* zeigt an, daß die Datenmengenkomponente gerade Zeilen aus einer oder mehreren verbundenen Tabelle(n) abruft.

Verwenden Sie diese Werte zur Durchführung von Aktionen, die vom aktuellen Status der Datenmenge abhängen. Beispielsweise kann eine Routine zur Aktualisierung von Daten auf die Eigenschaft *RecordSetState* zugreifen, um zu ermitteln, ob die Datenmenge aktiv ist und ob sie gerade andere Aktivitäten ausführt, wie das Herstellen einer Verbindung oder das Abrufen von Daten.

Batch-Aktualisierungen verwenden

ADO-Datenmengenkomponenten bieten die Möglichkeit, an der Datenmenge vorgenommene Änderungen zwischenzuspeichern und sie anschließend als Batch-Operation in die Datenbank einzutragen bzw. einzelne oder alle Änderungen zu verwerfen. Batch-Aktualisierungen können auf diese Weise auf Datenmengenkomponentenebene als Transaktionssteuerung eingesetzt werden. (Normalerweise werden Transaktionen mit Methoden der ADO-Verbindungskomponente verarbeitet.)

Die Funktionen zur Batch-Aktualisierung von ADO-Datenmengenkomponenten umfassen die folgenden Bereiche:

- Datenmengen im Batch-Aktualisierungsmodus öffnen
- Den Aktualisierungsstatus einzelner Datensätze prüfen
- Mehrere Datensätze auf Grundlage des Aktualisierungsstatus filtern
- Batch-Aktualisierungen in Datenbanktabellen eintragen
- Batch-Aktualisierungen verwerfen

Datenmengen im Batch-Aktualisierungsmodus öffnen

Eine ADO-Datenmengenkomponente muß folgende Kriterien erfüllen, damit sie im Batch-Aktualisierungsmodus geöffnet werden kann:

1. Die Eigenschaft *CursorType* der Komponente muß den Wert *ctKeySet* (Vorgabewert) oder *ctStatic* haben.
2. Die Eigenschaft *LockType* muß den Wert *ltBatchOptimistic* enthalten.
3. Der Befehl muß eine SELECT-Abfrage sein.

Vor dem Aktivieren der Datenmengenkomponente weisen Sie den Eigenschaften *CursorType* und *LockType* die oben genannten Werte zu. Fügen Sie in die Eigenschaft *CommandText* (für *TADODataSet*) oder in die Eigenschaft *SQL* (für *TADOQuery*) eine SELECT-Anweisung ein. Bei *TADOStoredProc*-Komponenten geben Sie in *ProcedureName* den Namen einer Stored Procedure an, die eine Ergebnismenge zurückgibt. Die Werte dieser Eigenschaften können zur Entwurfszeit mit Hilfe des Objektinspektors oder zur Laufzeit vom Programm zugewiesen werden. Das nachfolgende Beispiel zeigt die Vorbereitung einer *TADODataSet*-Komponente für den Batch-Aktualisierungsmodus.

```
with ADODataSet1 do begin
  CursorLocation := clUseServer;
  CursorType := ctKeyset;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

Nachdem eine Datenmenge im Batch-Aktualisierungsmodus geöffnet wurde, werden alle an den Daten vorgenommenen Änderungen nicht mehr direkt in die Datenbanktabellen eingetragen sondern zwischengespeichert.

Den Aktualisierungsstatus einzelner Datensätze prüfen

Sie ermitteln den Aktualisierungsstatus eines bestimmten Datensatzes, indem Sie den Datensatzzeiger auf diese Zeile setzen und dann die Eigenschaft *RecordStatus* der ADO-Datenkomponente prüfen. Diese Eigenschaft gibt den Aktualisierungsstatus des aktuellen Datensatzes an.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unveränderter Datensatz';
  rsModified:   StatusBar1.Panels[0].Text := 'Geänderter Datensatz';
  rsDeleted:    StatusBar1.Panels[0].Text := 'Gelöschter Datensatz';
  rsNew:        StatusBar1.Panels[0].Text := 'Neuer Datensatz';
end;
```

Mehrere Datensätze auf Grundlage des Aktualisierungsstatus filtern

Mit Hilfe der Eigenschaft *FilterGroup* können Sie eine Datensatzmenge filter, um nur die Datensätze anzuzeigen, die zu einer Gruppe mit demselben Aktualisierungsstatus gehören. Weisen Sie *FilterGroup* die *TFilterGroup*-Konstante zu, die dem Aktualisierungsstatus der anzuzeigenden Datensätze entspricht. Der Wert *fgNone* (Vorgabewert für diese Eigenschaft) gibt an, daß kein Filterprozeß durchgeführt wird und daß alle Datensätze, ohne Berücksichtigung des Aktualisierungsstatus, angezeigt werden (ausgenommen zum Löschen markierte Datensätze). Das nachfolgende Beispiel bewirkt, daß nur Datensätze mit ausstehenden Batch-Aktualisierungen angezeigt werden.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

Damit die Eigenschaft *FilterGroup* wirksam werden kann, muß die Eigenschaft *Filtered* der ADO-Datenmengenkomponente den Wert *True* besitzen.

Batch-Aktualisierungen in Datenbanktabellen eintragen

Ausstehende Datenänderungen, die noch nicht eingetragen oder verworfen wurden, werden durch Aufruf der Methode *UpdateBatch* in Datenbanktabellen eingetragen. Für Datensätze, die geändert wurden und übertragen werden, werden die Änderungen in die Datenbanktabelle eingetragen, auf der die Datensatzmenge basiert. Ein zum Löschen markierter Datensatz im Zwischenspeicher führt dazu, daß der entsprechende Datensatz in der Datenbanktabelle gelöscht wird. Ein eingefügter Datensatz (der im Zwischenspeicher, aber nicht in der Datenbanktabelle vorhanden ist) wird der Datenbanktabelle hinzugefügt. Für geänderte Zeilen werden die Spalten in den entsprechenden Datensätzen der Datenbanktabellen mit den neuen Werten der Spalten im Zwischenspeicher aktualisiert.

Wird *UpdateBatch* ohne Parameter aufgerufen, werden alle ausstehenden Aktualisierungen in die Datenbank eingetragen. Optional kann ein *TUpdateBatchOptions*-Wert als Parameter für *UpdateBatch* übertragen werden. Wird ein anderer Wert als *ubAffectAll* übergeben, wird immer nur eine Teilmenge der ausstehenden Änderungen eingetragen. Die Übertragung von *ubAffectAll* führt zu dem gleichen Resultat wie ein Aufruf ohne Parameter: alle ausstehenden Aktualisierungen werden eingetragen. Im nachfolgenden Beispiel wird nur der Datensatz in die Datenbank eingetragen, auf dem sich der Datensatzzeiger befindet:

```
ADODataSet1.UpdateBatch(ubAffectCurrent);
```

Batch-Aktualisierungen verwerfen

Ausstehende Datenänderungen, die noch nicht eingetragen oder verworfen wurden, werden durch Aufruf der Methode *CancelBatch* verworfen. Für Datensätze, die geändert wurden und verworfen werden, werden die Spaltenwerte wieder auf die Werte zurückgesetzt, die vor dem letzten Aufruf von *CancelBatch* oder *UpdateBatch* bzw. vor der aktuell ausstehenden Batch-Aktualisierung vorhanden waren.

Wird *CancelBatch* ohne Parameter aufgerufen, werden alle ausstehenden Aktualisierungen verworfen. Optional kann ein *TUpdateBatchOptions*-Wert als Parameter für *CancelBatch* übertragen werden. Wird ein anderer Wert als *ubAffectAll* übergeben, wird immer nur eine Teilmenge der ausstehenden Änderungen verworfen. Die Übertragung von *ubAffectAll* führt zu dem gleichen Resultat wie ein Aufruf ohne Parameter: alle ausstehenden Aktualisierungen werden verworfen. Im nachfolgenden Beispiel werden alle ausstehenden Änderungen verworfen:

```
ADODataSet1.Cancel;
```

Daten aus Dateien laden und in Dateien speichern

Die aus einer ADO-Datensatzmengenkomponente abgerufenen Daten können in einer Datei gespeichert werden, so daß zu einem späteren Zeitpunkt von demselben oder einem anderen Computer darauf zugegriffen werden kann. Verwenden Sie die Methode *SaveToFile*, um Daten in einer Datei zu speichern. Der Zugriff auf diese Daten erfolgt dann mit der Methode *LoadFromFile*. Die Daten können in einem der proprietären Formate ADTG oder XML gespeichert werden. Sie legen das zu verwendende Format über eine der Konstanten *pfADTG* oder *pfXML* von *TPersistFormat* im Parameter Format der Methode *SaveToFile* fest.

Die erste Prozedur im nachfolgenden Beispiel speichert die von der Komponente *TADODataSet* abgerufene Datenmenge *ADODataSet1* in eine Datei. Die Zieldatei ist eine ADTG-Datei mit dem Namen *SaveFile*, die auf einem lokalen Laufwerk gespeichert wird. Die zweite Prozedur lädt diese Datei in die *TADODataSet*-Komponente *ADODataSet2*.

```
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then begin
    DeleteFile('c:\SaveFile');
    StatusBar1.Panels[0].Text := 'Save file deleted!';
  end;
  ADODataSet1.SaveToFile('c:\SaveFile', pfADTG);
end;

procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataSet2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Datei nicht vorhanden!';
end;
```

Die zu speichernden oder zu ladenden Datenmengenkomponenten müssen sich nicht wie oben in demselben Formular, in derselben Anwendung oder auf demselben Computer befinden, was die Aktenkoffer-Übertragung von Daten erleichtert.

Beim Aufruf der Methode *LoadFromFile* wird die Datenmengenkomponente automatisch aktiviert.

Ist die im Parameter *FileNam*e der Methode *SaveToFile* angegebene Datei bereits vorhanden, wird die Exception *EOleException* ausgelöst. Diese Exception wird auch in dem ähnlichen Fall ausgelöst, daß die im Parameter *FileName* von *LoadFromFile* angegebene Datei nicht existiert.

Die Dateiformate ADTG und XML sind die einzigen Formate, die von ADO unterstützt werden. Sie sind jedoch nicht notwendigerweise für alle ADO-Versionen verfügbar. Schlagen Sie in der Dokumentation Ihrer ADO-Version nach, welche Dateiformate von dieser unterstützt werden.

Parameter in Befehlen verwenden

Die Verwendung von Parametern in Befehlen und SQL-Anweisungen, die als Befehl ausgeführt werden, setzt folgendes voraus:

1. Fügen Sie die Parameter in die SQL-Anweisung ein (diese ist gekennzeichnet durch den vorangestellten Doppelpunkt).
2. Definieren Sie die Eigenschaftswerte für jede *TParameter-Komponente.*

Für jedes Token in der SQL-Anweisung, das als Parameter interpretiert wird, wird automatisch eine *TParameter*-Komponente erstellt und der Eigenschaft *Parameters* der Datenmengenkomponente hinzugefügt (ein *TParameters*-Array von *TParameter*-Komponenten). Um zur Entwurfszeit die Werte der Parameterkomponenten zuzuweisen, verwenden Sie den Eigenschaftseditor für die Eigenschaft *Parameters*. Sie rufen diesen Editor auf, indem Sie im Objektinspektor auf die Ellipsenschaltfläche für die Eigenschaft *Parameters* klicken.

Zur Laufzeit greifen Sie auf Parameterkomponenten mit Hilfe der Eigenschaft *Parameters* der Datenmengenkomponente zu, um deren Werte festzulegen oder abzurufen. Geben Sie für Parameters eine Indexzahl an, die der Ordinalposition eines bestimmten Parameters in der SQL-Anweisung (relativ zu den anderen Parametern) entspricht. Die Zählung des Index beginnt bei Null, so daß der erste Parameter mit der Indexzahl Null referenziert wird, der zweite mit der Indexzahl Eins usw. Alternativ können Sie die *TParameters*-Referenz der Eigenschaft Parameters verwenden und die Methode *ParamByName* aufrufen, um den Parameter über seinen Namen zu referenzieren.

```
{ Den ersten Parameter über einen Index referenzieren }
ADOQuery1.Parameters[0].Value := 'telephone';

{ Einen Parameter über seinen Namen referenzieren }
ADOQuery1.Parameters.ParamByName('Amount').Value := 123;
```

Im Beispiel unten wird die folgende SQL-Anweisung in einer *TADOQuery*-Komponente eingesetzt.

```
SELECT CustNo, Company, State
FROM CUSTOMER
WHERE (State = :StateParam)
```

Diese Anweisung enthält einen Parameter: *StateParam*. Die nachfolgende Routine schließt die ADO-Abfragekomponente, legt den Wert des Parameters *StateParam* über die Eigenschaft *Parameters* fest und öffnet dann die ADO-Abfragekomponente erneut. Die Eigenschaft *Parameters* erfordert die Angabe eines Parameters als Zahl, welche die Ordinalposition des Parameters in der Anweisung, relativ zu anderen Parametern, angibt. Die Zählung beginnt bei Null, so daß der erste Parameter in der Eigenschaft *Parameters* mit der Indexzahl Null gekennzeichnet wird, der zweite mit Eins usw. Da *StateParam* der erste Parameter in der Anweisung ist, wird er durch die Indexzahl Null gekennzeichnet.

```
procedure TForm1.GetCaliforniaBtnClick(Sender: TObject);
begin
  with ADOQuery1 do begin
    Close;
    Parameters[0].Value := 'CA';
    Open;
  end;
end;
```

Die nachfolgende Prozedur dient im wesentlichen demselben Zweck, verwendet jedoch die Methode *TParameters.ParamByName* zur Festlegung des Parameterwertes. Für die Methode *ParamByName* wie in der SQL-Anweisung muß ein Parameter durch seinen Namen gekennzeichnet sein (jedoch ohne den Doppelpunkt).

```
procedure TForm1.GetFloridaBtnClick(Sender: TObject);
begin
  with ADOQuery1 do begin
    Close;
    Parameters.ParamByName('StateParam').Value := 'FL';
    Open;
  end;
end;
```

TADODataSet verwenden

Die Komponente *TADODataSet* ermöglicht es Delphi-Anwendungen, auf Daten in einer oder mehreren Tabellen zuzugreifen, die sich in einer Datenbank befinden, auf die über ADO zugegriffen wird. Die Tabellen werden mit Hilfe der Eigenschaft *CommandText* der ADO-Datenmengenkomponente durch Angabe des Namens oder durch eine SQL-Anweisung festgelegt.

Der Zugriff auf die Datenbank erfolgt über eine Datenspeicherverbindung, die über die Eigenschaft *ConnectionString* einer ADO-Datenmengenkomponente oder über eine gesonderte *TADOConnection*-Komponente, die in der Eigenschaft *Connection* definiert ist, eingerichtet wird. Weitere Informationen hierzu finden Sie im Abschnitt »Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen« auf Seite 23-15.

Der Zugriff auf Daten einer *TADODataSet*-Komponente über visuelle Steuerelemente, das Navigieren zwischen Datensätzen und das Ändern von Daten durch den Quelltext erfolgt auf die gleiche Weise wie bei den übrigen ADO-Datenmengenkomponenten. Weitere Informationen zu gemeinsamen Merkmalen aller Datenmengen-

komponenten finden Sie unter »Gemeinsame Merkmale aller ADO-Datenmengenkomponenten« auf Seite 23-14.

Mit Befehlen auf Datenmengen zugreifen

Die Komponente *TADODataSet* kann Daten aus einer einzelnen Tabelle über den Tabellennamen abrufen. Sie kann aber auch mit Hilfe einer gültigen SQL-Anweisung auf Daten in einer oder mehreren Tabellen zugreifen. In beiden Fällen wird der Tabellenname oder die SQL-Anweisung als Befehl ausgeführt.

Legen Sie den Namen der Tabelle oder eine SQL-Anweisung in der Eigenschaft *CommandText* fest, und aktivieren Sie die Komponente. Zur Entwurfszeit können Sie den Befehlstexteditor zum Schreiben des Befehls verwenden. Sie rufen diesen Editor auf, indem Sie im Objektinspektor auf die Ellipsenschaltfläche für die Eigenschaft *CommandText* klicken. Zur Laufzeit übergeben Sie *CommandText* einen Befehl als String.

```
ADODataSet1.CommandText := 'SELECT * FROM Customer';
```

Verwenden Sie die Eigenschaft *CommandType*, um den Typ des auszuführenden Befehls festzulegen: *cmdTable* (oder *cmdTableDirect*), wenn der Befehl aus dem Namen der Tabelle besteht, oder *cmdText* für SQL-Anweisungen. Sie können auch *cmdUnknown* angeben, wenn der Befehlstyp zur Ausführungszeit noch unbekannt ist oder wenn ADO den Befehlstyp auf Grundlage des Inhalts von *CommandText* ermitteln soll. Zur Entwurfszeit können Sie die gewünschten Werte für *CommandType* in der Dropdown-Liste im Objektinspektor auswählen. Zur Laufzeit weisen Sie einen Wert des Typs *TCommandType* zu.

```
ADODataSet1.CommandType := cmdText;
```

Sie aktivieren *TADODataSet*, indem Sie die zugehörige Methode *Open* aufrufen oder indem Sie der Eigenschaft *Active* den Wert *True* zuweisen.

```
with ADODataSet1 do begin
  Connection := ADOConnection1;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Customer';
  Open;
end;
```

TADOTable verwenden

Die Komponente *TADOTable* ermöglicht es Delphi-Anwendungen, auf Daten in einer einzelnen Tabelle zuzugreifen, die sich in einer Datenbank befindet, auf die über ADO zugegriffen wird. Die entsprechende Tabelle wird mit Hilfe der Eigenschaft *TableName* der ADO-Tabellenkomponente festgelegt.

Der Zugriff auf die Datenbank erfolgt über eine Datenspeicherverbindung, die über die Eigenschaft *ConnectionString* einer ADO-Tabellenkomponente oder über eine gesonderte *TADOConnection*-Komponente, die in der Eigenschaft Connection definiert ist, eingerichtet wird. Weitere Informationen hierzu finden Sie im Abschnitt »Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen« auf Seite 23-15.

Der Zugriff auf Daten einer *TADOTable*-Komponente über visuelle Steuerelemente, das Navigieren zwischen Datensätzen und das Ändern von Daten durch den Quelltext erfolgt auf die gleiche Weise wie bei den übrigen ADO-Datenmengenkomponenten. Weitere Informationen hierzu finden Sie im Abschnitt »Gemeinsame Merkmale aller ADO-Datenmengenkomponenten« auf Seite 23-14.

Zu verwendende Tabellen festlegen

Sobald eine *TADOTable*-Komponente eine gültige Verbindung zu einer Datenbank eingerichtet hat, kann sie auf die in dieser Datenbank enthaltenen Tabellen zugreifen. Eine einzelne Tabelle der Datenbank geben Sie in der Eigenschaft *TableName* an. Wenn die ADO-Tabellenkomponente aktiviert wird, kann auf die Tabelle und die enthaltenen Daten über *TADOTable* zugegriffen werden.

Wenn für die *TADOTable*-Komponente eine gültige Datenspeicherverbindung vorhanden ist, zeigt der Eigenschaftseditor für die Eigenschaft *TableName* zur Entwurfszeit eine Liste der verfügbaren Tabellen an. Wählen Sie in dieser Liste die gewünschte Tabelle. Zur Laufzeit weisen Sie der Eigenschaft *TableName* als Wert einen String mit dem Namen der Tabelle zu.

```
ADOTable1.TableName := 'Orders';
```

Erfolgt die Verbindung zu einem Datenspeicher über eine *TADOConnection*-Komponente, können Sie deren Methode *GetTableNames* zum Abrufen einer Liste der verfügbaren Tabellen verwenden. *GetTableNames* füllt ein bereits vorhandenes Stringlistenobjekt mit den Namen der Tabellen, die über die Verbindung zur Verfügung stehen.

Die erste der nachfolgenden Routinen füllt beispielsweise eine *TListBox*-Komponente namens *ListBox1* mit den Namen der Tabellen, die über die *TADOConnection*-Komponente *ADOConnection1* verfügbar sind. Die zweite Routine ist eine Ereignisbehandlungsroutine für das Ereignis *OnDblClick* von *ListBox1*. In dieser Routine wird der Eigenschaft *TableName* der *TADOTable*-Komponente *ADOTable1* der aktuell in *ListBox1* ausgewählte Tabellenname zugewiesen. Anschließend wird die ADO-Tabellenkomponente aktiviert.

```
procedure TForm1.ListTablesButtonClick(Sender: TObject);
begin
  ADOConnection1.GetTableNames(ListBox1.Items, False);
end;

procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  with ADOTable1 do begin
    Close;
    TableName := (Sender as TListBox).Items[(Sender as TListBox).ItemIndex];
    Open;
  end;
end;
```

TADOQuery verwenden

Die Komponente *TADOQuery* ermöglicht es Delphi-Anwendungen, mit Hilfe von SQL-Anweisungen auf Daten in einer oder mehreren Tabellen einer ADO-Datenbank zuzugreifen. Die SQL-Anweisung zur Verwendung für die ADO-Abfragekomponente wird in der Eigenschaft *SQL* festgelegt. *TADOQuery* kann Daten entweder mit der DML-Sprache (Data Manipulation Language) abfragen oder Metadatenobjekte mit der DDL-Sprache (Data Definition Language) erzeugen oder löschen. Die in einer *TADOQuery*-Komponente verwendete SQL-Sprache muß mit dem verwendeten ADO-Treiber kompatibel sein. Delphi überprüft eine SQL-Anweisung nicht auf Gültigkeit und führt die Anweisung auch nicht aus. Sie wird lediglich zur Ausführung an das Datenbank-Back-End weitergegeben. Generiert die SQL-Anweisung eine Ergebnismenge, wird diese über Delphi vom Datenbank-Back-End an die *TADOQuery*-Komponente übermittelt, die sie für die Anwendung verfügbar macht.

Der Zugriff auf die Datenbank erfolgt über eine Datenspeicherverbindung, die über die Eigenschaft *ConnectionString* der ADO-Abfragekomponente oder über eine gesonderte *TADOConnection*-Komponente, die in der Eigenschaft *Connection* definiert ist, eingerichtet wird. Weitere Informationen hierzu finden Sie im Abschnitt »Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen« auf Seite 23-15.

Der Zugriff auf Daten einer *TADOQuery*-Komponente über visuelle Steuerelemente, das Navigieren zwischen Datensätzen und das Ändern von Daten durch den Quelltext erfolgt auf die gleiche Weise wie bei den übrigen ADO-Datenmengenkomponenten. Weitere Informationen hierzu finden Sie im Abschnitt »Gemeinsame Merkmale aller ADO-Datenmengenkomponenten« auf Seite 23-14.

SQL-Anweisungen festlegen

Zur Entwurfszeit rufen Sie den Eigenschaftseditor für die Eigenschaft *SQL* auf, indem Sie im Objektinspektor auf die Ellipsenschaltfläche klicken. Geben Sie im Editor die SQL-Anweisung für *TADOQuery* an.

Zur Laufzeit weisen Sie der Eigenschaft *SQL* einen Wert zu. Wie die Standardabfragekomponente *TQuery* ist auch die Eigenschaft *TADOQuery.*SQL ein Stringlistenobjekt. Verwenden Sie die Eigenschaften und Methoden der Stringlistenklasse, um der Eigenschaft *SQL* Werte zuzuweisen.

Im nachfolgenden Beispiel wird der Eigenschaft *SQL* einer *TADOQuery*-Komponente mit dem Namen *ADOQuery1* eine SELECT-Anweisung zugewiesen.

```
with ADOQuery1 do begin
  Close;
  with SQL do begin
    Clear;
    Add('SELECT Company, State');
    Add('FROM CUSTOMER');
    Add('WHERE State = ' + QuotedStr('HI'));
```

```
      Add('ORDER BY Company');
    end;
    Open;
  end;
```

SQL-Anweisungen ausführen

Eine *TADOQuery*-Komponente mit einer gültigen SQL-Anweisung in der Eigenschaft *SQL* kann auf zwei Weisen ausgeführt werden. Welches dieser Verfahren zu verwenden ist, hängt davon ab, ob die SQL-Anweisung eine Ergebnismenge zurückgibt.

Handelt es sich um eine SQL-Anweisung, die eine Ergebnismenge zurückgibt, sollte die ADO-Abfragekomponente durch Aufruf ihrer Methode *Open* oder durch Zuweisung des Wertes *True* an ihre Eigenschaft *Active* aktiviert werden. Da nur SELECT-Anweisungen eine Ergebnismenge zurückgeben, wird eine *TADOQuery*-Komponente mit einer SELECT-Anweisung immer auf diese Weise aktiviert.

```
ADOQuery1.SQL.Text := 'SELECT * FROM TrafficViolations';
ADOQuery1.Open;
```

Beachten Sie, daß während des Entwurfs einer Anwendung in der integrierten Entwicklungsumgebung von Delphi keine Methoden aufgerufen werden können. Aus diesem Grund können diese Art von Abfragen zur Entwurfszeit nur über die Eigenschaft *Active* ausgeführt werden. Von der Funktionsweise her entspricht dies dem Aufruf der Methode *Open* zur Laufzeit.

Sie führen eine SQL-Anweisung aus, die keine Ergebnismenge zurückgibt, indem Sie die Methode *ExecSQL* der Komponente *TADOQuery* aufrufen. Dies gilt für alle SQL-Anweisungen wie INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE usw., außer für SELECT. *TADOQuery*-Komponenten mit einer dieser SQL-Anweisungen in ihrer Eigenschaft *SQL* werden immer auf diese Weise aktiviert.

```
ADOQuery1.SQL.Text := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
ADOQuery1.ExecSQL;
```

Wie im vorhergehenden Beispiel kann die Komponente *TADOCommand* zur Ausführung von SQL-Anweisungen verwendet werden, die keine Ergebnismengen zurückgeben.

TADOStoredProc verwenden

Die Komponente *TADOStoredProc* ermöglicht es Delphi-Anwendungen, Stored Procedures in einer Datenbank auszuführen, auf die über einen ADO-Datenspeicher zugegriffen wird. Die auszuführende Stored Procedure wird mit Hilfe der Eigenschaft *ProcedureName* der ADO-Komponente für Stored Procedures festgelegt.

Der Zugriff auf die Datenbank erfolgt über eine Datenspeicherverbindung, die über die Eigenschaft *ConnectionString* einer ADO-Datenmengenkomponente oder über eine gesonderte *TADOConnection*-Komponente, die in der Eigenschaft *Connection* definiert ist, eingerichtet wird. Weitere Informationen hierzu finden Sie im Abschnitt »Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen« auf Seite 23-15.

Anwendungen können auf die von einer *TADOStoredProc*-Komponente abgerufenen Ergebnismengen auf die gleiche Weise zugreifen wie auf die der üblichen, BDE-bezogenen Abfragekomponente *TStoredProc*. Verwenden Sie die ADO-Komponente für Stored Procedures für die Eigenschaft *DataSet* einer *TDataSource*-Standardkomponente. *TDataSource* fungiert dann als Verbindungselement für den Datenaustausch zwischen der ADO-Komponente für Stored Procedures und datensensitiven Steuerelementen. Weitere Informationen hierzu finden Sie im Abschnitt »Visuelle datensensitive Steuerelemente verwenden« auf Seite 23-15.

Der Zugriff auf Daten einer *TADOStoredProc*-Komponente über visuelle Steuerelemente, das Navigieren zwischen Datensätzen und das Ändern von Daten durch den Quelltext erfolgt auf die gleiche Weise wie bei den übrigen ADO-Datenmengenkomponenten. Weitere Informationen zu gemeinsamen Merkmalen aller Datenmengenkomponenten finden Sie im Abschnitt »Gemeinsame Merkmale aller ADO-Datenmengenkomponenten« auf Seite 23-14.

Stored Procedures festlegen

Sobald eine *TADOStoredProc*-Komponente eine gültige Verbindung zu einer Datenbank eingerichtet hat, kann sie gepeicherte Prozeduren ausführen, die in dieser Datenbank verfügbar sind. Geben Sie den Namen der Stored Procedure aus der Datenbank in der Eigenschaft *ProcedureName* an. Verwenden Sie zum Aktivieren der ADO-Komponente für Stored Procedures die Methode *Open,* wenn eine Ergebnismenge zurückgegeben wird. Andernfalls verwenden Sie die Methode *ExecProc*.

Wenn für die Komponente *TADOStoredProc* eine gültige Datenspeicherverbindung vorhanden ist, zeigt der Eigenschaftseditor für die Eigenschaft *ProcedureName* zur Entwurfszeit eine Liste der verfügbaren Stored Procedures an. Wählen Sie in dieser Liste die gewünschte Stored Procedure. Zur Laufzeit weisen Sie der Eigenschaft *ProcedureName* als Wert einen String mit dem Namen der Stored Procedure zu.

```
ADOStoredProc1.ProcedureName := 'DeleteEmployee';
```

Erfolgt die Verbindung zu einem Datenspeicher über eine *TADOConnection*-Komponente, können Sie deren Methode *GetProcedureNames* zum Abrufen einer Liste der verfügbaren Stored Procedures verwenden. *GetProcedureNames* füllt ein bereits vorhandenes Stringlistenobjekt mit den Namen der Stored Procedures, die über die Verbindung zur Verfügung stehen.

Die erste der nachfolgenden Routinen füllt beispielsweise eine *TListBox*-Komponente namens *ListBox1* mit den Namen der Stored Procedures, die über die *TADOConnection*-Komponente *ADOConnection1* verfügbar sind. Die zweite Routine ist eine Ereignisbehandlungsroutine für das Ereignis *OnDblClick* von *ListBox1*. In dieser Routine wird der Eigenschaft *ProcedureName* der *TADOStoredProc*-Komponente *ADOStoredProc1* der aktuell in *ListBox1* ausgewählte Name einer Stored Procedure zugewiesen. Die ADO-Komponente für Stored Procedures wird dann mit ihrer Methode *ExecProc* ausgeführt.

```
procedure TForm1.ListProceduresButtonClick(Sender: TObject);
begin
  ADOConnection1.GetProcedureNames(ListBox1.Items);
end;
```

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  with ADOStoredProc1 do begin
    ProcedureName := TListBox(Sender).Items[TListBox(Sender).ItemIndex];
   ExecProc;
  end;
end;
```

Stored Procedures ausführen

Eine *TADOStoredProc*-Komponente mit dem Namen einer vorhandenen Stored Procedure in der Eigenschaft *ProcedureName* kann auf zwei Weisen ausgeführt werden. Welches dieser Verfahren zu verwenden ist, hängt davon ab, ob die Stored Procedure eine Ergebnismenge zurückgibt.

Handelt es sich um eine Stored Procedure, die eine Ergebnismenge zurückgibt, sollte die ADO-Komponente für Stored Procedures durch Aufruf ihrer Methode *Open* oder durch Zuweisung des Wertes *True* an ihre Eigenschaft *Active* aktiviert werden.

```
ADOStoredProc1.ProcedureName := 'ShowPurebreds';
ADOStoredProc1.ExecProc;
```

Beachten Sie, daß während des Entwurfs einer Anwendung in der integrierten Entwicklungsumgebung von Delphi keine Methoden aufgerufen werden können. Aus diesem Grund können diese Art von Stored Procedures zur Entwurfszeit nur über die Eigenschaft *Active* ausgeführt werden.

Sie führen eine Stored Procedure aus, die keine Ergebnismenge zurückgibt, indem Sie die Methode *ExecProc* der Komponente *TADOStoredProc* aufrufen. Dies gilt für alle SQL-Anweisungen wie INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE usw., außer für SELECT. *TADOQuery*-Komponenten mit einer SQL-Anweisung in ihrer Eigenschaft *SQL* werden immer auf diese Weise aktiviert.

```
ADOStoredProc1.ProcedureName := 'DeletePoodles';
ADOStoredProc1.ExecProc;
```

Parameter mit Stored Procedures verwenden

Die Komponente *TADOStoredPro*c kann drei verschiedene Parametertypen aufnehmen, die jedoch möglicherweise nicht von allen Datenbanktypen unterstützt werden. Ein Parameter kann für die Eingabe, die Ausgabe oder die Rückgabe einer Ergebnismenge verwendet werden. Dieser Abschnitt beschreibt den Einsatz von Parametern für Stored Procedures für diese drei Funktionen. Ein Parameter kann auch für zwei Zwecke, beispielsweise als Eingabe- und Ausgabeparameter verwendet werden. Dies ist jedoch lediglich eine Variation der drei grundlegenden Funktionen. Der Einsatz eines Parameters in dieser doppelten Funktion besteht lediglich in der Kombination von zwei funktionellen Verfahren, die nachfolgend beschrieben werden.

Die Richtung bzw. der Zweck eines Parameters wird in der Stored Procedure definiert, während diese erstellt wird. Dieser Zweck kann zu einem späteren Zeitpunkt von einer Front-End-Anwendung nicht mehr geändert werden. So ist es beispielsweise nicht möglich, einen Eingabeparameter mit Hilfe von Delphi-Code in einen Ausgabeparameter umzuwandeln. Um dies zu erreichen, muß die Stored Procedure ver-

worfen und neu erstellt werden, wobei dem Parameter die neue Funktion zugewiesen wird. Der Zweck eines bestimmten Parameters wird in der Eigenschaft *TParameter.Direction* angezeigt, auf die zur Entwurfszeit über den Objektinspektor und zur Laufzeit programmatisch zugegriffen werden kann.

Tabelle 23.2 Parameterrichtung

Parameter	Zweck
pdInput	Dieser Parameter dient zur Übergabe eines Wertes an die Stored Procedure vor der Ausführung.
pdOutput	Dieser Parameter dient zur Rückgabe eines einzelnen Wertes aus einer Stored Procedure nach der Ausführung.
pdInputOutput	Dieser Parameter kann sowohl als Eingabe- und als Ausgabeparameter verwendet werden (siehe oben).
pdReturnValue	Dieser Parameter enthält nach der Ausführung eine Ergebnismenge.
pdUnknown	Für diesen Parameter konnte zum Zeitpunkt der Auswertung kein Zweck ermittelt werden.

Eingabeparameter von TADOStoredProc verwenden

Sie verwenden einen Eingabeparameter, um einer Stored Procedure vor deren Ausführung einen Wert zuzuweisen. Diese Werte sind normalerweise in der WHERE-Klausel der SQL-Anweisung einer Stored Procedure enthalten, um die Anzahl der betroffenen Tabellenzeilen einzuschränken. Weisen Sie dem Parameter einen Wert zu, bevor Sie *TADOStoredProc* aktivieren, indem Sie die Methode *Open* aufrufen oder der Eigenschaft *Active* den Wert *True* zuweisen (bei Stored Procedures mit Ergebnismenge) bzw. bevor *TADOStoredProc* mit der Methode *ExecProc* ausgeführt wird.

Sie können zur Entwurfszeit auf Parameter zugreifen, indem Sie im Objektinspektor auf die Ellipsenschaltfläche für die Eigenschaft *Parameters* klicken. Hierdurch wird der Parametereditor geöffnet. Geben Sie für die Eigenschaft *Value* einen Wert des gewünschten Typs ein.

Zur Laufzeit weisen Sie der Eigenschaft *Value* der Komponente *TParameter* einen Wert für den Zielparameter zu. Verwenden Sie die Referenz *TParameter*, die in der Eigenschaft *Parameters* von *TADOStoredProc* verfügbar gemacht wird.

```
with ADOStoredProc1 do begin
  Close;
  Parameters[1].Value := 1;
  Open;
end;
```

Ausgabeparameter von TADOStoredProc verwenden

Sie verwenden einen Ausgabeparameter, um einen einzelnen Wert aus einer Stored Procedure zurückzugeben. Während Ergebnismengen aus mehreren Zeilen verschiedener Spalten bestehen können, muß dieser Ausgabeparameter einer Zeile mit einer Spalte entsprechen. Verwendet die Stored Procedure zum Abrufen dieses Wertes eine SELECT-Anweisung, wird beim Zurückgeben mehrerer Werte eine Exception ausgelöst.

Ein Ausgabeparameter enthält erst dann einen Wert, wenn die Stored Procedure aktiviert oder ausgeführt wurde. Beachten Sie, daß nicht alle Datenbanksysteme sowohl die Rückgabe von Ergebnismengen als auch Ausgabeparameter unterstützen. Schlagen Sie in der Dokumentation Ihres Datenbanksystems nach, welche Optionen in dieser Hinsicht verfügbar sind. Unterstützt das Datenbanksystem sowohl die Rückgabe von Ergebnismengen als auch Ausgabeparameter, kann *TADOStoredProc* durch ihre Methode *Open* (oder die Eigenschaft *Active*) bzw. durch *ExecProc* aktiviert werden. Stehen nicht beide Optionen zur Verfügung, können Sie Ausgabeparameter zum Abrufen von Werten nur über einen Aufruf der Methode *ExecProc* verwenden.

Sie können zur Entwurfszeit auf Parameter zugreifen, indem Sie im Objektinspektor auf die Ellipsenschaltfläche für die Eigenschaft *Parameters* klicken. Hierdurch wird der Parametereditor geöffnet. Wird *TADOStoredProc* über ihre Eigenschaft *Active* aufgerufen (zur Entwurfszeit die einzige Möglichkeit), entnehmen Sie der Eigenschaft *Value* des Ausgabeparameters den zurückgegebenen Wert.

Zur Laufzeit weisen Sie der Eigenschaft *Value* der *TParameter*-Komponente einen Wert für den Zielparameter zu. Verwenden Sie die Referenz *TParameter*, die über die Eigenschaft *Parameters* von *TADOStoredProc* zur Verfügung steht. Die nachfolgende Beispielroutine gibt über den Ausgabeparameter *@OutParam1* als Wert einen String zurück.

```
with ADOStoredProc1 do begin
  Close;
  ShowMessage(VarToStr(Parameters.ParamByName('@OutParam1').Value));
  ExecProc;
end;
```

Rückgabewertparameter von TADOStoredProc verwenden

Es ist nicht erforderlich, direkt auf die Rückgabewertparameter zuzugreifen. Verfahren Sie statt dessen mit der über einen Rückgabewertparameter abgerufenen Ergebnismenge wie mit jeder anderen Datenmenge.

Um über visuelle datensensitive Steuerelemente auf die Ergebnismenge zuzugreifen, verwenden Sie als Wert für die Eigenschaft *DataSet* einer *TDataSource*-Komponente eine Referenz auf die Komponente *TADOStoredProc*. *TDataSource* fungiert dann als Verbindungselement für den Datenaustausch zwischen der Komponente *TADOStoredProc* und datensensitiven Steuerelementen. Zur Entwurfszeit rufen Sie hierfür den Objektinspektor auf. Wählen Sie für die Eigenschaft *DataSet* von *TDataSource* die Komponente *TADOStoredProc* in der Dropdown-Liste aus. Zur Laufzeit weisen Sie der Eigenschaft *DataSet* eine Referenz auf *TADOStoredProc* zu.

```
ADOStoredProc1.Close;
DataSource1.DataSet := ADOStoredProc1;
ADOStoredProc1.Open;
```

Alternativ kann der Zugriff auf die und die Bearbeitung der Ergebnismenge über die von *TDataSet* geerbten Navigations- und Bearbeitungseigenschaften und -methoden erfolgen. Weitere Informationen zum Bearbeiten von Daten mit Hilfe von Datenmengenkomponenten finden Sie im Abschnitt »Modifying data« auf Seite 18-31. Informationen zum Navigieren zwischen Tabellenzeilen in Datenmengenkomponenten finden Sie im Abschnitt »Navigating datasets« auf Seite 18-15.

Befehle ausführen

Die in Delphi zur Verfügung stehenden ADO-Komponenten ermöglichen einer Anwendung die Ausführung von Befehlen. In diesem Abschnitt wird beschrieben, wie Befehle ausgeführt werden und welche Komponenten hierfür eingesetzt werden.

In der ADO-Umgebung sind Befehle Textrepräsentationen der für den Provider spezifischen Aktionsanforderungen. Normalerweise handelt es sich um DDL- und DML-SQL-Anweisungen. Die in Befehlen zu verwendende Sprache ist für den Provider spezifisch, normalerweise aber konform mit dem Standard SQL-92 für die Abfragesprache SQL.

Befehle können von unterschiedlichen Delphi-Komponenten ausgeführt werden. Jede Komponente, die zur Ausführung von Befehlen geeignet ist, führt diese auf etwas unterschiedliche Weise aus, was verschiedene Vor- und Nachteile mit sich bringt. Welche Komponente für einen bestimmten Befehl am besten geeignet ist, hängt von dem Befehlstyp sowie davon ab, ob dieser Befehl eine Ergebnismenge zurückgibt oder nicht. Im allgemeinen sollte für Befehle, die keine Ergebnismenge zurückgeben, die Komponente *TADOCommand* verwendet werden (obwohl auch die Komponente *TADOQuery* zur Ausführung dieser Befehle verwendet werden kann). Befehle, die Ereignismengen zurückgeben, sollten über die Komponente *TADODataSet* aufgerufen werden. Alternativ kann die Anweisung auch in der Eigenschaft *SQL* einer *TADOQuery*-Komponente verwendet werden.

Die Komponente *TADOCommand* ermöglicht die Ausführung von Befehlen, wobei diese nacheinander ausgeführt werden. Sie ist speziell für die Ausführung von Befehlen geeignet, die keine Ergebnismenge zurückgeben, beispielsweise für DDL-SQL-Anweisungen. Durch eine überladene Version ihrer Methode *Execute* ist sie jedoch auch in der Lage, eine Ergebnismenge zurückzugeben, auf die ADO-Datenmengenkomponenten zugreifen können.

Sie legen Befehle in der Eigenschaft *CommandText* fest. Optional können Sie einen Befehl auch mit Hilfe der Eigenschaft *CommandType* beschreiben. Geben Sie keinen bestimmten Befehlstyp an, versucht der Server nach Möglichkeit, diesen auf Grundlage des Befehls in *CommandText* zu ermitteln. Befehle zur Verwendung für ADO-Befehlskomponenten können Parameter enthalten, die vor der Ausführung des Befehls durch Werte ersetzt werden. Die ADO-Befehlskomponente muß über eine gültige Verbindung zu einem ADO-Datenspeicher verfügen, bevor ein Befehl ausgeführt werden kann.

Befehle festlegen

Geben Sie die Befehle, die mit Hilfe der ADO-Befehlskomponente ausgeführt werden sollen, in deren Eigenschaft *CommandText* an. Zur Entwurfszeit geben Sie den Befehl (eine SQL-Anweisung, einen Tabellennamen oder den Namen einer Prozedur) über den Objektinspektor in der Eigenschaft *CommandText* an. Zur Laufzeit übergeben Sie der Eigenschaft *CommandText* als Wert einen String mit dem Befehl.

Gegebenenfalls können Sie den Typ des Befehls, der ausgeführt werden soll, in der Eigenschaft *CommandType* explizit definieren. Sie können u. a. folgende Konstanten

für *CommandType* auswählen: *cmdText* (wenn der Befehl eine SQL-Anweisung ist), *cmdTable* (wenn er ein Tabellenname ist) und *cmdStoredProc* (wenn er der Name einer Stored Procedure ist). Zur Entwurfszeit wählen Sie in einer Liste im Objektinspektor den erforderlichen Befehlstyp aus. Zur Laufzeit weisen Sie der Eigenschaft *CommandType* einen Wert des Typs *TCommandType* zu.

```
with ADOCommand1 do begin
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
...
end;
```

Die Methode Execute verwenden

Bevor ein Befehl mit Hilfe einer ADO-Befehlskomponente ausgeführt werden kann, muß für die *TADOCommand*-Komponente eine gültige Verbindung zu einem Datenspeicher bestehen. Weitere Informationen hierzu finden Sie im Abschnitt »Mit ADO-Datenmengenkomponenten Verbindungen zu Datenspeichern herstellen« auf Seite 23-15.

Zur Ausführung des Befehls rufen Sie die Methode *Execute* der ADO-Befehlskomponente auf. Rufen Sie für Befehle, die keine Parameter oder Optionen zur Ausführung benötigen, die einfache überladene Version von *Execute* ohne jegliche Methodenparameter auf.

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

Informationen zur Ausführung von Befehlen, die eine Ergebnismenge zurückgeben, finden Sie im Abschnitt »Mit Befehlen auf Ergebnismengen zugreifen« auf Seite 23-32.

Befehle abbrechen

Nachdem ein Versuch zur Ausführung eines Befehls eingeleitet wurde (mit der Methode *Execute* einer *TADOCommand*-Komponente), kann diese durch Aufruf der Methode *Cancel* abgebrochen werden.

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;

procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

Ein Aufruf der Methode *Cancel* wirkt sich nur aus, wenn ein Befehl aussteht und dieser asynchron ausgeführt wurde (*eoAsynchExecute* im Parameter *ExecuteOptions* der

Methode *Execute*). Ein Befehl wird als ausstehend bezeichnet, wenn die Methode *Execute* aufgerufen wurde und der Befehl noch nicht ausgeführt oder eine Zeitüberschreitung aufgetreten ist. Diese tritt auf, wenn der Befehl nicht vor dem Ende des Zeitraums, der in der Eigenschaft *CommandTimeout* der *TADOCommand*-Komponente in Sekunden festgelegt wurde, erfolgreich ausgeführt wurde. Wollen Sie für den Zeitraum, nach dessen Ablauf ein Befehl abgebrochen wird, einen anderen als den Standardwert von 30 Sekunden festlegen, geben Sie diesen vor dem Aufruf der Methode Execute in der Eigenschaft *CommandTimeout* an.

Mit Befehlen auf Ergebnismengen zugreifen

Die Ausführung von Befehlen, die Ergebnismengen zurückgeben, erfolgt auf die gleiche Weise wie bei Befehlen, die dies nicht tun. Der einzige Unterschied besteht darin, daß eine bereits vorhandene ADO-Datenmengenkomponente die Ergebnismenge repräsentieren muß. Die Methode *Execute* von *TADOCommand* gibt ein ADO-Datensatzmengenobjekt zurück. Weisen Sie diesen Rückgabewert der Eigenschaft *RecordSet* einer ADO-Datensatzmengenkomponente (wie *TADODataSet*) zu.

Im nachfolgenden Beispiel wird die ADO-Datensatzmenge, die von einem Aufruf der Methode *Execute* einer *TADOCommand*-Komponente (*ADOCommand1*) erzeugt wurde, der Eigenschaft *RecordSet* einer *TADODataSet*-Komponente (*ADODataSet1*) zugewiesen.

```
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataSet1.Recordset := Execute;
end;
```

Sobald diese Zuweisung an die Eigenschaft *RecordSet* der ADO-Datenmengenkomponente erfolgte, wird die Datenmengenkomponente automatisch aktiviert, und die Daten werden verfügbar. Verwenden Sie die Methoden und Eigenschaften der Datenmengenkomponente, um über den Quelltext auf die Daten zuzugreifen. Um über visuelle datensensitive Steuerelemente auf die Daten zuzugreifen, verwenden Sie eine *TDataSource*-Komponente als Verbindungselement für den Datenaustausch zwischen der ADO-Datenmenge und den datensensitiven Steuerelementen.

Weitere Informationen zum Ausführen von Befehlen, die keine Ergebnismengen zurückgeben, finden Sie im Abschnitt »Befehle ausführen« auf Seite 23-30.

Befehlsparameter verarbeiten

Die Ausführung eines Befehls mit Parametern erfolgt auf die gleiche Weise wie bei Befehlen ohne diese. Der einzige Unterschied besteht darin, daß den Parametern vor der Ausführung des Befehls Werte zugewiesen werden müssen.

Für jeden Parameter in einem Befehl wird der Eigenschaft *Parameters* der Komponente *TADOCommand* automatisch ein *TParameter*-Objekt hinzugefügt. Verwenden Sie

zur Entwurfszeit den Parametereditor, um auf Parameter zuzugreifen. Sie rufen diesen Editor auf, indem Sie im Objektinspektor auf die Ellipsenschaltfläche für die Eigenschaft *Parameters* klicken. Zur Laufzeit verwenden Sie die Eigenschaften und Methoden von *TParameter*, um die Werte der einzelnen Parameter festzulegen (oder abzurufen).

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

Sie greifen auf einzelne *TParameter*-Objekte in Parameters zu, indem Sie mit Hilfe der Eigenschaft *Parameters* der *TADOCommand*-Komponente eine Zahl angeben, die die relative Position im Befehl bezeichnet. Verweisen Sie auf *TParameter*-Objekte durch Verwendung ihres Namens in der Methode *TParameters.ParamByName*.

Kapitel

24

Client-Datenmengen

TClientDataSet ist eine Datenmengenkomponente, die ohne BDE (Borland Database Engine) oder ActiveX-Datenobjekte (ADO) verwendet werden kann. Ihre gesamten Datenzugriffsfunktionen befinden sich in der Datei MIDAS.DLL, die viel kleiner ist als die BDE und einfacher installiert und konfiguriert werden kann. Bei der Arbeit mit Client-Datenmengen werden keine Datenbankkomponenten oder ADO-Verbindungskomponenten benötigt, da keine Datenbankverbindung vorhanden ist.

Client-Datenmengen unterstützen alle von *TDataset* eingeführten Datenzugriffs-, Navigations-, Datenbeschränkungs- und Filterfunktionen. In der Anwendung muß jedoch ein Mechanismus implementiert werden, durch den die Client-Datenmenge Daten lesen und Aktualisierungen schreiben kann. Dazu stehen folgende Möglichkeiten zur Verfügung:

- Direkte Lese- und Schreibzugriffe durch die Client-Datenmenge auf eine einfache Datei. Dieser Mechanismus wird von Datenbankanwendungen mit unstrukturierten Dateien verwendet. Weitere Informationen zu diesem Thema finden Sie unter »Datenbankanwendungen mit unstrukturierten Daten« auf Seite 13-15.
- Lesen aus einer anderen Datenmenge. Client-Datenmengen verfügen über eine Vielzahl von Mechanismen, um Daten von anderen Datenmengen zu kopieren. Informationen hierzu finden Sie unter »Daten aus einer anderen Datenmenge kopieren« auf Seite 24-13.
- Austauschen von Daten mit einem Remote-Anwendungsserver über eine *IAppServer*-Schnittstelle. Diese Technik wird von den Clients in einer mehrschichtigen Datenbankanwendung verwendet. Weitere Informationen zu diesem Thema finden Sie in Kapitel 14, »Mehrschichtige Anwendungen erstellen«.

Diese Mechanismen können in einer Anwendung kombiniert werden, um das *Aktenkoffer*-Modell zu unterstützen. Die Benutzer speichern eine Kopie der Daten in einer unstrukturierten Datei und bearbeiten diese offline. Später werden die lokalen Änderungen von der Client-Datenmenge an den Anwendungsserver gesendet. Dieser trägt sie in die richtige Datenbank ein und gibt eventuell auftretende Fehler zur Be-

handlung an die Client-Datenmenge zurück. Informationen über diese Art von Anwendungen finden Sie unter »Das Aktenkoffer-Modell« auf Seite 13-19.

Client-Datenmengen

Wie jede andere Datenmenge können auch Client-Datenmengen dazu verwendet werden, um die datensensitiven Steuerelemente durch eine Datenquellenkomponente mit Daten zu versorgen. Erläuterungen zum Anzeigen der Informationen einer Datenbank mit Hilfe datensensitiver Steuerelemente finden Sie in Kapitel 26, »Datensensitive Steuerelemente«.

TClientDataSet ist von *TDataSet* abgeleitet und erbt daher die leistungsfähigen Eigenschaften, Methoden und Ereignisse, die für alle Datenmengenkomponenten definiert sind. Ausführliche Informationen zu diesen generischen Datenmengenfunktionen finden Sie in Kapitel 18, »Datenmengen«.

Client-Datenmengen halten im Gegensatz zu den anderen Datenmengen ihre gesamten Daten im Hauptspeicher. Aus diesem Grund können neben der Unterstützung allgemeiner Datenbankfunktionen zusätzliche Möglichkeiten in Betracht gezogen werden.

Durch die Daten einer Client-Datenmenge navigieren

Wenn eine Anwendung datensensitive Standardsteuerelemente verwendet, kann der Benutzer durch eine Client-Datenmenge wie durch jede andere Datenmenge navigieren. Die Navigation kann auch programmseitig mit Hilfe der Standardmethoden *First, GotoKey, Last, Next* und *Prior* implementiert werden. Informationen über diese Methoden finden Sie unter »Durch Datenmengen navigieren« auf Seite 18-11.

Client-Datenmengen unterstützen auch Lesezeichen (Bookmarks), um bestimmte Datensätze zu markieren und zu diesen zurückzukehren. Weitere Informationen hierzu finden Sie unter »Datensätze markieren und dorthin zurückkehren« auf Seite 18-15.

Bei einer Client-Datenmenge kann der Cursor mit Hilfe der Eigenschaft *RecNo* in einen bestimmten Datensatz gesetzt werden. Normalerweise wird in einer Anwendung mit RecNo die Nummer des aktuellen Datensatzes ermittelt. In einer Client-Datenmenge kann mit dieser Eigenschaft jedoch ein bestimmter Datensatz zum aktuellen gemacht werden.

Datensätze einschränken

Mit Hilfe eines Bereichs oder Filters können die in einer Anwendung verfügbaren Daten vorübergehend auf eine bestimmte Teilmenge eingeschränkt werden. Die Client-Datenmenge zeigt dann nicht alle in ihrem Puffer enthaltenen Daten an, sondern nur diejenigen, die mit den angegebenen Bereichs- oder Filterbedingungen übereinstimmen. Weitere Informationen zu Filtern finden Sie unter »Teilmengen von

Daten mit Hilfe von Filtern anzeigen und bearbeiten« auf Seite 18-19. Informationen über Bereiche finden Sie unter »Mit Teilmengen der Daten arbeiten« auf Seite 20-12

Bei den meisten Datenmengen werden Filter-Strings in SQL-Befehle umgewandelt, die anschließend auf dem Datenbankserver ausgeführt werden. Aus diesem Grund bestimmt der SQL-Dialekt des jeweiligen Servers, welche Operationen in den Filter-Strings verwendet werden. Client-Datenmengen implementieren eine eigene Filterunterstützung, die mehr Operationen als bei anderen Datenmengen ermöglicht. So können Filterausdrücke beispielsweise String-Operatoren enthalten, die Substrings zurückgeben, Operatoren, die Datums-/Zeitwerte auswerten und vieles mehr. Es können auch BLOB-Felder oder komplexe Feldtypen wie ADT- und Array-Felder gefiltert werden. Weitere Informationen finden Sie unter in der Online-Dokumentation zu *TClientSet.Filter*.

Die Client-Datenmenge hält auch dann noch alle Datensätze im Speicher, wenn ein Bereich oder Filter aktiv ist. Die Beschränkung definiert nur, welche Datensätze in den datensensitiven Steuerelementen angezeigt werden. In mehrschichtigen Anwendungen können auch die in der Client-Datenmenge gespeicherten Informationen eingeschränkt werden, indem Sie die entsprechenden Parameter an den Anwendungsserver übergeben. Weitere Informationen zu diesem Thema finden Sie unter »Datensätze durch Parameter einschränken« auf Seite 24-18».

Haupt/Detail-Beziehungen

Client-Datenmengen unterstützen ebenso wie Tabellen Haupt/Detail-Formulare. Beim Einrichten einer Haupt/Detail-Beziehung werden zwei Datenmengen so verknüpft, daß jeweils alle Datensätze der einen (die Detaildatensätze) mit einem Datensatz der anderen (dem Hauptdatensatz) übereinstimmen. Weitere Informationen zu diesem Thema finden Sie unter »Haupt/Detail-Formulare erstellen« auf Seite 20-28.

Sie können Haupt/Detail-Beziehungen in Client-Datenmengen auch mit Hilfe verschachtelter Tabellen einrichten. Dazu haben Sie folgende Möglichkeiten:

- Rufen Sie Datensätze mit verschachtelten Detaildaten von einer Provider-Komponente ab. Wenn ein Provider für die Haupttabelle einer Haupt/Detail-Beziehung verwendet wird, erstellt er automatisch ein verschachteltes Datenmengenfeld für die Detaildaten.
- Definieren Sie im Felder-Editor verschachtelte Detaildatenmengen. Klicken Sie dazu mit der rechten Maustaste in der Client-Datenmenge, und wählen Sie *Felder-Editor*. Fügen Sie neue persistente Felder hinzu, indem Sie mit der rechten Maustaste klicken und *Neues Feld* wählen. Geben Sie für das neue Feld den Typ *DataSet* an. Definieren Sie anschließend im Editor die Struktur der Detailtabelle.

Wenn die Client-Datenmenge verschachtelte Datenmengen enthält, ermöglicht *TDBGrid* das Anzeigen dieser Daten in einem Popup-Fenster. Alternativ können Sie auch datensensitive Steuerelemente verwenden, indem Sie für die Detaildaten eine separate Client-Datenmenge einsetzen. Erstellen Sie dann persistente Felder für die Datenmenge, einschließlich eines *DataSet*-Feldes für die verschachtelte Datenmenge.

Sie können nun eine Client-Datenmenge für die verschachtelte Detailmenge erstellen. Weisen Sie ihrer Eigenschaft *DataSetField* das persistente *DataSet*-Feld der Hauptdatenmenge zu.

In mehrschichtigen Anwendungen müssen verschachtelte Detailmengen verwendet werden, wenn Aktualisierungen der Haupt- und Detailtabellen an den Anwendungsserver gesendet werden sollen. In Datenbankanwendungen mit unstrukturierten Dateien können mit Hilfe verschachtelter Detailmengen die Detaildatensätze zusammen mit den Hauptdatensätzen in einer Datei gespeichert werden. Dadurch müssen nicht zwei Datenmengen getrennt geladen und anschließend die Indizes neu erstellt werden, um die Haupt/Detail-Beziehung wieder einzurichten.

Hinweis Verschachtelte Detailmengen können nur verwendet werden, wenn die Eigenschaft *ObjectView* der Client-Datenmenge den Wert *True* hat.

Datenwerte beschränken

Client-Datenmengen unterstützen auch Datenbeschränkungen. Sie können daher jederzeit eigene Beschränkungen definieren. Auf diese Weise können Sie in Ihrer Anwendung festlegen, welche Werte die Benutzer in eine Client-Datenmenge eingeben können. Weitere Informationen über das Festlegen von Beschränkungen finden Sie unter »Benutzerdefinierte Beschränkungen hinzufügen« auf Seite 24-22.

Wird für die Kommunikation mit einem entfernten Datenbankserver eine *Provider*-Komponente verwendet, können die Server-Beschränkungen auch an die Client-Datenmenge weitergegeben werden. Weitere Informationen zu diesen Datenbeschränkungen finden Sie unter »Server-Beschränkungen« auf Seite 15-11.

In mehrschichtigen Anwendungen gibt es oft Situationen, in denen die Datenbeschränkungen deaktiviert werden müssen, besonders wenn die Client-Datenmenge nicht alle Datensätze der entsprechenden Datenmenge auf dem Anwendungsserver enthält. Weitere Informationen hierzu finden Sie im Abschnitt »Server-Beschränkungen verarbeiten« auf Seite 24-21.

Daten das Attribut Nur-Lesen zuweisen

Mit der Eigenschaft *CanModify* von *TDataSet* kann ermittelt werden, ob die Informationen in einer Datenmenge bearbeitet werden können. Die Eigenschaft kann jedoch nicht in einer Anwendung geändert werden, da bei manchen von *TDataSet* abgeleiteten Klassen die zugrundeliegende Datenbank und nicht die Anwendung bestimmt, ob die Daten bearbeitet werden können.

Bei einer Client-Datenmenge kann der Zugriff auf die Informationen immer gesteuert werden, da sich die Daten im Hauptspeicher befinden. Setzen Sie einfach die Eigenschaft *ReadOnly* auf *True*, um Schreibzugriffe zu verhindern. Die Eigenschaft *CanModify* erhält dann automatisch den Wert *False*.

Im Gegensatz zu anderen Datenmengen brauchen Sie eine Client-Datenmenge nicht zu schließen, um ihren Nur-Lesen-Status zu ändern. Eine Anwendung kann einer Client-Datenmenge das Attribut Nur-Lesen auch vorübergehend zuweisen, indem sie die aktuelle Einstellung der Eigenschaft *ReadOnly* ändert.

Daten bearbeiten

Die Daten einer Client-Datenmenge werden mit Hilfe der Eigenschaft *Data* in einem Datenpaket im Hauptspeicher verwaltet. Änderungen werden jedoch standardmäßig nicht in dieses Paket geschrieben. Die Einfüge-, Lösch- und Bearbeitungsoperationen (durch den Benutzer oder im Quelltext) werden statt dessen in einem internen Änderungsprotokoll festgehalten, auf das mit der Eigenschaft *Delta* zugegriffen werden kann. Das Protokoll wird in zwei Situationen verwendet:

- Bei der Arbeit mit einer Provider-Komponente wird es für das Senden von Aktualisierungen an den Anwendungsserver benötigt.
- Es ermöglicht jeder Anwendung einen leistungsfähigen Mechanismus, um Änderungen an den Daten rückgängig zu machen.

Mit der Eigenschaft *LogChanges* kann die Protokollierung vorübergehend deaktiviert werden. Hat die Eigenschaft den Wert *True*, werden alle Änderungen festgehalten. Wird *LogChanges* auf *False* gesetzt, werden die Aktualisierungen direkt in der Eigenschaft *Data* durchgeführt. Die Protokollierung kann in einschichtigen Anwendungen deaktiviert werden, wenn Sie keine Rückgängig-Funktion benötigen.

Die Protokolleinträge bleiben erhalten, bis sie von der Anwendung gelöscht werden. Dies erfolgt bei den folgenden Operationen:

- Änderungen rückgängig machen
- Änderungen speichern

Hinweis Beim Speichern der Client-Datenmenge werden die Änderungen nicht aus dem Protokoll entfernt. Wenn Sie die Datenmenge wieder laden, werden die Eigenschaften *Data* und *Delta* mit den Werten wiederhergestellt, die sie beim Speichern hatten.

Änderungen rückgängig machen

Die Originalversion eines Datensatzes bleibt in der Eigenschaft *Data* unverändert erhalten. Dennoch sieht ein Benutzer, der einen Datensatz bearbeitet, ihn verläßt und dann erneut darauf zugreift, immer die zuletzt geänderte Version des Datensatzes. Wenn ein Benutzer oder eine Anwendung einen Datensatz mehrmals ändert, wird jede geänderte Version als separater Eintrag im Änderungsprotokoll gespeichert.

Das Speichern jeder einzelnen Datensatzänderung ermöglicht es, bei Bedarf mehrstufige Rückgängig-Operationen durchzuführen, um einen früheren Zustand des Datensatzes wiederherzustellen.

- Um die letzte Änderung zu entfernen, rufen Sie *UndoLastChange* auf. *UndoLastChange* erwartet den Booleschen Parameter *FollowChange*, der angibt, ob der Cursor auf den wiederhergestellten Datensatz positioniert wird (*True*) oder im aktuellen Datensatz bleibt (*False*). Wenn es für einen Datensatz mehrere Änderungen gibt, wird durch jeden Aufruf von *UndoLastChange* eine weitere Änderungsstufe gelöscht. UndoLastChange gibt einen Booleschen Wert zurück, der angibt, ob eine Änderung erfolgreich (*True*) entfernt werden konnte. Mit Hilfe der Eigenschaft *ChangeCount* können Sie ermitteln, ob noch Änderungen rückgängig gemacht werden müssen. *ChangeCount* enthält die Anzahl der Protokolleinträge.

- Anstatt alle Änderungsstufen eines Datensatzes einzeln zu entfernen, können Sie alle Änderungen auf einmal löschen. Wählen Sie zu diesem Zweck den gewünschten Datensatz aus, und rufen Sie *RevertRecord* auf. *RevertRecord* entfernt alle Änderungen des aktuellen Datensatzes aus dem Änderungsprotokoll.
- Sie können den aktuellen Protokollstatus während der Bearbeitung mit Hilfe der Eigenschaft*SavePoint* jederzeit speichern. Rufen Sie mit *SavePoint* die aktuelle Position im Änderungsprotokoll ab. Wenn Sie dann später die Änderungen seit dem Lesen der Eigenschaft rückgängig machen wollen, setzen Sie *SavePoint* auf den zuvor gelesenen Wert. Sie können auch mehrere Speicherpunkte abrufen. Nachdem Sie jedoch das Protokoll bis zu einem bestimmten Punkt gesichert haben, sind alle später gelesenen Werte ungültig.
- Mit *CancelUpdates* können Sie alle Änderungen aus dem Protokoll entfernen. *CancelUpdates* löscht das Änderungsprotokoll und verwirft dadurch alle Bearbeitungen der Datensätze. Gehen Sie aber vorsichtig mit dem Aufruf von *CancelUpdates* um. Nach dem Aufruf lassen sich die aus dem Protokoll gelöschten Änderungen nicht wiederherstellen.

Änderungen speichern

Client-Datenmengen verwenden unterschiedliche Mechanismen zum Eintragen der Änderungen aus dem Protokoll, je nachdem, ob sie in einer eigenständigen Anwendung oder einem Anwendungsserver verwendet werden. Bei beiden Möglichkeiten wird das Protokoll automatisch geleert, nachdem alle Aktualisierungen eingetragen wurden.

Bei eigenständigen Anwendungen können die Änderungen einfach mit den lokalen Daten in der Eigenschaft *Data* zusammengeführt werden. In diesem Fall müssen keine Bearbeitungen anderer Benutzer berücksichtigt werden. Das Eintragen wird mit der Methode *MergeChangeLog* durchgeführt. Informationen hierzu finden Sie unter »Änderungen in die Daten schreiben« auf Seite 24-27.

Die Methode *MergeChangeLog* kann in mehrschichtigen Anwendungen nicht verwendet werden. Die Informationen im Änderungsprotokoll werden vom Anwendungsserver benötigt, um die aktualisierten Datensätze in die Datenbank zu schreiben. Rufen Sie in diesem Fall *ApplyUpdates* auf. Diese Methode sendet die Änderungen an den Anwendungsserver und aktualisiert die Eigenschaft *Data* nur, wenn die geänderten Datensätze erfolgreich in die Datenbank eingetragen wurden. Weitere Informationen finden Sie im Abschnitt »Aktualisierungen eintragen« auf Seite 24-23.

Sortieren und Indizieren

Die Verwendung von Indizes hat mehrere Vorteile:

- Die Client-Datenmengen können die Daten schnell lokalisieren.
- In der Anwendung können Beziehungen zwischen Client-Datenmengen (z. B. Lookup-Tabellen oder Haupt/Detail-Formulare) eingerichtet werden.
- Mit ihnen kann die Reihenfolge der Datensätze festgelegt werden.

Wenn eine mehrschichtige Anwendung mit einer Client-Datenmenge arbeitet, erbt diese einen Standardindex und eine Standardsortierreihenfolge, die auf den vom Anwendungsserver empfangenen Daten beruhen. Der Standardindex hat den Namen DEFAULT_ORDER. Sie können diese Sortierreihenfolge verwenden, aber den Index weder ändern noch löschen.

Zusätzlich zu diesem Standardindex verwaltet die Client-Datenmenge einen zweiten Index namens CHANGEINDEX, der sich auf die geänderten und im Änderungsprotokoll gespeicherten Datensätze bezieht (Eigenschaft *Delta*). CHANGEINDEX ordnet alle vorhandenen Datensätze in der Client-Datenmenge an, wenn die in Delta festgelegten Änderungen übernommen wurden. CHANGEINDEX beruht auf der Sortierreihenfolge, die von DEFAULT_ORDER geerbt wird. Der Index CHANGEINDEX läßt sich ebenso wie DEFAULT_ORDER weder ändern noch löschen.

Sie können andere vorhandene Indizes für eine Datenmenge verwenden oder eigene Indizes anlegen. In den folgenden Abschnitten erfahren Sie, wie neue Indizes erzeugt und auf Client-Datenmengen angewendet werden.

Einen neuen Index hinzufügen

Um einen neuen Index für eine Client-Datenmengen zu erstellen, rufen Sie ihre Methode *AddIndex* auf. Mit ihr können Sie folgende Merkmale des Index festlegen:

- Den Namen des Index. Der Name kann zur Laufzeit zum Wechseln des Index verwendet werden.
- Die Felder für den Index. Die Werte dieser Felder werden zum Sortieren und Lokalisieren der Datensätze verwendet.
- Die Sortierfolge des Index. Standardmäßig werden Indizes aufsteigend (entsprechend der Ländereinstellung im System) sortiert, und es wird zwischen Groß- und Kleinschreibung unterschieden. Diese Einstellungen können natürlich geändert werden. Sie können auch angeben, daß eine Liste von Feldern ohne Unterscheidung der Groß-/Kleinschreibung und eine andere Liste absteigend sortiert wird.
- Die Standardstufe für die Gruppenunterstützung des Index.

Tip Sie können intern berechnete Felder verwenden, um Client-Datenmengen zu indizieren und zu sortieren.

Die von Ihnen erzeugten Indizes werden entsprechend den Ländereinstellungen Ihres Rechners in aufsteigender alphabetischer Reihenfolge sortiert. Sie können dem Optionsparameter das Flag *ixDescending* hinzufügen, um die Standardeinstellung außer Kraft zu setzen.

Hinweis Wenn Sie die Indizes zur selben Zeit wie die Client-Datenmenge erstellen, können einige Felder aufsteigend und andere absteigend indiziert werden. Informationen hierzu finden Sie unter »Datenmengen mit Feld- und Indexdefinitionen erstellen« auf Seite 13-16.

Beim Sortieren von String-Feldern in Indizes wird per Voreinstellung die Groß-/Kleinschreibung berücksichtigt. Wenn Sie dem Optionsparameter das Flag *ixCaseInsensitive* hinzufügen, wird die Schreibweise bei der Sortierung ignoriert.

Hinweis Werden Indizes zu demselben Zeitpunkt wie die Client-Datenmenge erstellt, können die Indizes in einigen Feldern ohne Berücksichtigung und in anderen mit Berücksichtigung der Groß-/Kleinschreibung erstellt werden. Weitere Informationen finden Sie unter »Datenmengen mit Feld- und Indexdefinitionen erstellen« auf Seite 13-16.

Achtung Mit der Methode *AddIndex* hinzugefügte Indizes werden nicht zusammen mit der Client-Datenmenge in einer Datei gespeichert.

Indizes entfernen und wechseln

Wenn Sie einen Index löschen wollen, den Sie für eine Client-Datenmenge erzeugt haben, rufen Sie *DeleteIndex* auf und geben den Namen des betreffenden Index an. Die Indizes DEFAULT_ORDER und CHANGEINDEX können nicht gelöscht werden.

Wenn mehrere Indizes zur Verfügung stehen und Sie für die Client-Datenmenge einen anderen Index verwenden wollen, weisen Sie den gewünschten Index der Eigenschaft *IndexName* zu. Zur Entwurfszeit können Sie den gewünschten Index im Objektinspektor im Dropdown- Feld der Eigenschaft *IndexName* auswählen.

Daten mit Indizes gruppieren

Wenn Sie in einer Client-Datenmenge einen Index verwenden, werden die Datensätze automatisch in der entsprechenden Reihenfolge angeordnet. Aus diesem Grund enthalten benachbarte Felder in indizierten Feldern normalerweise identische Werte. Sehen Sie sich dazu folgendes Beispiel aus der Tabelle ORDERS an, die nach den Feldern SALESREP und CUSTOMER indiziert ist:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Augrund der Sortierreihenfolge stimmen benachbarte Werte in der Spalte *SalesRep* überein. Bei allen Datensätzen mit dem Wert 1 stimmen benachbarte Werte in der Spalte *Customer* überein. Die Daten sind also nach *SalesRep* und innerhalb der *SalesRep*-Gruppe nach *Customer* gruppiert. Jeder Gruppierung ist eine Ebene zugeordnet. In diesem Beispiel ist die Gruppe *SalesRep* Ebene 1 und die Gruppe *Customer* Ebene 2 (da sie in Ebene 1 verschachtelt ist). Die Gruppierungsebene entspricht der Feldreihenfolge im Index. Bei der Angabe des Index können Sie auch die Gruppierungsebenen angeben (bis zur Anzahl der Felder im Index).

Bei einer Client-Datenmenge können Sie ermitteln, wo sich der aktuelle Datensatz innerhalb einer bestimmten Gruppierungsebene befindet. Auf diese Weise können die Datensätze unterschiedlich angezeigt werden, je nachdem, ob sie der erste Datensatz in der Gruppe, in der Mitte einer Gruppe oder der letzte Datensatz einer Gruppe sind. So können beispielsweise doppelte Datensätze von der Anzeige ausgeschlossen

werden, indem nur der jeweils erste Datensatz einer Gruppe angezeigt wird. Diese Möglichkeit führt bei der vorherigen Tabelle zu folgender Anzeige:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

Mit der Methode *GetGroupState* können Sie ermitteln, wo sich der aktuelle Datensatz innerhalb einer Gruppe befindet (erster Datensatz, letzter Datensatz oder keines von beiden). Übergeben Sie dabei die Gruppenebene in einem Integer-Parameter.

Bei Bedarf indizieren

Anstatt einen Index zu erzeugen, der einen festen Bestandteil der Datenmenge bildet, können Sie bei Bedarf einen temporären Index anlegen. Dazu geben Sie in der Eigenschaft *IndexFieldNames* die Felder an, die indiziert werden sollen. Trennen Sie bei der Eingabe die einzelnen Feldnamen mit einem Semikolon voneinander ab. Achten Sie dabei auf die richtige Reihenfolge der Feldnamen in der Liste.

Hinweis Wenn die Indizierung zur Laufzeit erfolgt, kann kein absteigender Index und auch kein Index mit Berücksichtigung von Groß- und Kleinschreibung festgelegt werden.

Achtung Zur Laufzeit erstellte Indizes unterstützen keine Gruppierung.

Berechnete Felder hinzufügen

Sie können einer Client-Datenmenge wie jeder anderen Datenmenge berechnete Felder hinzufügen. Die Werte dieser Felder werden dynamisch berechnet, normalerweise anhand der Werte anderer Felder im selben Datensatz. Weitere Informationen hierzu finden Sie im Abschnitt »Berechnete Felder definieren« auf Seite 19-9.

Bei Client-Datenmengen kann der Zeitpunkt der Berechnung mit Hilfe von intern berechneten Feldern optimiert werden. Weitere Informationen zu diesem Thema finden Sie unten im Abschnitt »Intern berechnete Felder in Client-Datenmengen verwenden«.

Sie können auch berechnete Felder erstellen, um die Daten mehrerer Datensätze zusammenzufassen. Dies erfolgt mit Hilfe gewarteter Aggregate. Weitere Informationen zu diesem Thema finden Sie unter »Gewartete Aggregate verwenden« auf Seite 24-10.

Intern berechnete Felder in Client-Datenmengen verwenden

Bei gewöhnlichen Datenmengen muß der Wert der berechneten Felder bei jeder Änderung des aktuellen Datensatzes oder jeder Bearbeitung durch den Benutzer in der *OnCalcFields*-Ereignisbehandlungsroutine berechnet werden.

Dies ist auch in einer Client-Datenmenge möglich. Hier kann jedoch die Anzahl der Berechnungen minimiert werden, indem die berechneten Werte zusammen mit den Daten gespeichert werden. Die Werte müssen dann zwar immer noch berechnet werden, wenn der Benutzer den aktuellen Datensatz bearbeitet, bei Änderungen im Programm ist dies aber nicht mehr nötig. Damit die berechneten Werte in den Daten gespeichert werden, müssen Sie intern berechnete Felder anstelle von berechneten Feldern verwenden.

Auch die intern berechneten Felder werden in einer *OnCalcFields*-Ereignisbehandlungsroutine bearbeitet. Sie können jedoch die Routine optimieren, indem Sie die Eigenschaft State der Client-Datenmenge prüfen. Hat sie den Wert *dsInternalCalc*, müssen nur die internen Felder erneut berechnet werden. Bei *dsCalcFields* müssen die normalen berechneten Felder erneut verarbeitet werden.

Die Felder müssen vor dem Erstellen der Client-Datenmenge explizit als intern berechnet definiert werden. Wenn Sie die Client-Datenmenge mit persistenten Feldern erstellen, wählen Sie für die betreffenden Felder *InternalCalc* im Felder-Editor. Wenn Sie die Client- Datenmenge mit Felddefinitionen erstellen, setzen Sie die Eigenschaft *InternalCalcField* der gewünschten Felder auf *True*.

Hinweis Intern berechnete Felder werden auch von anderen Datenmengen verwendet. Jedoch werden sie dort nicht in der Ereignisbehandlungsroutine für *OnCalcFields* berechnet. Die Berechnung wird automatisch von der BDE oder vom Remote-Datenbankserver durchgeführt.

Gewartete Aggregate verwenden

In Client-Datenmengen können Gruppen von Datensätzen zusammengefaßt werden. Diese Zusammenfassungen werden beim Bearbeiten der Daten automatisch aktualisiert. Man bezeichnet sie daher als *Gewartete Aggregate*.

In der einfachsten Form können Sie mit einem gewarteten Aggregat Informationen wie z. B. die Summe aller Werte in einer Spalte abrufen. Aggregate unterstützen jedoch eine Vielzahl von Berechnungsarten und ermöglichen Zwischensummen für Datensatzgruppen, die durch ein Feld in einem Index definiert sind, der die Gruppierung unterstützt.

Aggregate angeben

Mit Hilfe der Eigenschaft *Aggregates* können Sie festlegen, daß in einer Client-Datenmenge Zusammenfassungen durchgeführt werden sollen. *Aggregates* ist eine Kollektion von Aggregatdefinitionen (*TAggregate*). Sie können Aggregatdefinitionen während des Entwurfs mit dem entsprechenden Editor oder zur Laufzeit mit der Methode *Add* von *Aggregates* hinzufügen. Um Feldkomponenten für die *Aggregate* zu

erstellen, definieren Sie im Felder- Editor persistente Felder für die zusammengefaßten Werte.

Hinweis Beim Erstellen von Zusammenfassungsfeldern werden die entsprechenden Aggregatobjekte automatisch der Eigenschaft *Aggregates* der Client-Datenmenge hinzugefügt. Sie dürfen nicht explizit hinzugefügt werden, wenn Sie persistente Aggregatfelder erstellen . Genaue Informationen hierzu finden Sie unter »Aggregatfelder definieren« auf Seite 19-13.

Bei jeder Aggregatdefinition kann mit der Eigenschaft *Expression* die Zusammenfassungsart angegeben werden. Sie können hier einfache Berechnungen wie die folgende verwenden:

```
Sum(Field1)
```

Es sind aber auch komplexe Ausdrücke mit mehreren Feldern wie im folgenden Beispiel möglich:

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregatausdrücke enthalten einen oder mehrere der in der folgenden Tabelle aufgeführten Operatoren:

Tabelle 24.1 Zusammenfassungsoperatoren für gewartete Aggregate

Operator	Beschreibung
Sum	Summieren der Werte eines numerischen Feldes oder Ausdrucks.
Avg	Berechnen des Mittelwertes eines numerischen oder Datums- / Zeitfeldes bzw. -ausdrucks.
Count	Zählen der Werte eines Feldes oder Ausdrucks, die nicht leer sind.
Min	Ermitteln des kleinsten Wertes in einem String-, numerischen oder Datums- / Zeitfeld bzw. -ausdruck.
Max	Ermitteln des größten Wertes in einem String-, numerischen oder Datums- / Zeitfeld bzw. -ausdruck.

Die Zusammenfassungsoperatoren arbeiten mit Feldwerten oder mit Ausdrücken zusammen, die aus Feldwerten mit denselben Operatoren wie bei einem Filter erstellt wurden (Zusammenfassungsoperatoren können jedoch nicht verschachtelt werden). In einem Ausdruck können Operatoren entweder nur für zusammengefaßte Werte oder für zusammengefaßte Werte und Konstanten verwendet werden. Sie können aber keine zusammengefaßten Werte mit Feldwerten kombinieren, da diese Ausdrücke nicht eindeutig sind (es ist nicht bekannt, welcher Datensatz den Feldwert liefern soll). Diese Regeln werden durch folgende Ausdrücke veranschaulicht:

Sum(Qty * Price)	{**Zulässig** – Zusammenfassung eines Ausdrucks mit Feldern}
Max(Field1) - Max(Field2)	{**Zulässig** – Ausdruck mit Zusammenfassungen}
Avg(DiscountRate) * 100	{**Zulässig** – Ausdruck mit Zusammenfassung und Konstante}

Min(Sum(Field1))	{**Unzulässig** – verschachtelte Zusammenfassungen}
Count(Field1) - Field2	{**Unzulässig** – Ausdruck mit Zusammenfassung und Feld}

Datensatzgruppen zusammenfassen

Standardmäßig werden bei gewarteten Aggregaten alle Datensätze in der Client-Datenmenge zusammengefaßt. Die Zusammenfassung kann jedoch auch mit den Datensätzen einer Gruppe durchgeführt werden. Auf diese Weise können beispielsweise Zwischensummen für Gruppen von Datensätzen mit einem gemeinsamen Feldwert berechnet werden.

Diese Art der Zusammenfassung ist aber nur möglich, wenn Sie einen Index verwenden, der die entsprechende Gruppierung unterstützt. Weitere Informationen hierzu finden Sie unter »Daten mit Indizes gruppieren« auf Seite 24-8.

Wenn Sie den für die Gruppierung erforderlichen Index definiert haben, geben Sie mit den Eigenschaften *IndexName* und *GroupingLevel* der Aggregatdefinition den Index und die Gruppe bzw. Untergruppe an.

Sehen Sie sich dazu die folgenden Daten aus der Tabelle ORDERS an, die zuerst nach dem Feld SALESREP und dann CUSTOMER gruppiert sind:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Mit den folgenden Anweisungen wird ein gewartetes Aggregat eingerichtet, um den Gesamtumsatz der einzelnen Vertreter zu berechnen:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'VertreterGesamt‘;
```

Wenn für jeden Vertreter auch die Umsätze bei den verschiedenen Kunden zusammengefaßt werden sollen, erstellen Sie ein gewartetes Aggregat für Ebene 2.

Gewartete Aggregate, die Gruppen von Datensätzen zusammenfassen, sind einem bestimmten Index zugeordnet. Die Eigenschaft *Aggregates* kann *Aggregate* enthalten, die verschiedene Indizes verwenden. Es sind aber jeweils nur die Aggregate gültig, die die gesamte Datenmenge zusammenfassen und die den aktuellen Index verwenden. Durch das Wechseln des aktuellen Index ändern sich auch die zulässigen Aggregatdefinitionen. Mit Hilfe der Eigenschaft *ActiveAggs* können Sie jederzeit feststellen, welche Aggregate gerade gültig sind.

Aggregatwerte abrufen

Mit der Eigenschaft *Value* eines TAggregate-Objekts können Sie den Wert eines Aggregats abrufen. Die Eigenschaft gibt das Aggregat für die Gruppe zurück, die den aktuellen Datensatz der Client-Datenmenge enthält.

Wenn Sie die gesamte Client-Datenmenge zusammenfassen, können Sie mit *Value* jederzeit auf das gewartete Aggregat zugreifen. Beim Zusammenfassen gruppierter Informationen müssen Sie jedoch sicherstellen, daß sich der aktuelle Datensatz in der gewünschten Gruppe befindet. Aus diesem Grund ist es ratsam, Aggregatwerte zu klar definierten Zeitpunkten (z. B. beim Wechseln zum ersten oder letzten Datensatz einer Gruppe) abzurufen. Mit der Methode *GetGroupState* können Sie prüfen, ob sich der aktuelle Datensatz in einer Gruppe befindet.

Um gewartete Aggregate in datensensitiven Steuerelementen anzuzeigen, erstellen Sie im Felder-Editor eine persistente Aggregatfeldkomponente. Dadurch wird in die Eigenschaft *Aggregates* der Client-Datenmenge automatisch die entsprechende Aggregatdefinition aufgenommen. Die neue Feldkomponente ist dann in der Eigenschaft *AggFields* enthalten und wird von der Methode *FindField* zurückgegeben.

Anwendungsspezifische Informationen zu den Daten hinzufügen

Sie können den Daten (Eigenschaft *Data*) einer Client-Datenmenge auch anwendungsspezifische Informationen hinzufügen. Diese Informationen werden zusammen mit den Daten in einer Datei oder einem Stream gespeichert und mit den Daten in eine andere Datenmenge kopiert. Die Informationen können optional in die Eigenschaft *Delta* aufgenommen werden. Sie werden dann mit den Aktualisierungen der Client-Datenmengen an den Anwendungsserver gesendet.

Anwendungsspezifische Informationen werden mit der Methode *SetOptionalParam* hinzugefügt. Die Daten werden dadurch in einem OleVariant-Wert unter einem bestimmten Namen gespeichert.

Um die anwendungsspezifischen Informationen zu lesen, verwenden Sie die Methode *GetOptionalParam*. Übergeben Sie der Methode den Namen, der beim Speichern der Informationen verwendet wurde.

Daten aus einer anderen Datenmenge kopieren

Sie können während des Entwurfs die Daten einer anderen Datenmenge kopieren, indem Sie mit der rechten Maustaste auf die Client-Datenmenge klicken und *Lokale Daten* zuweisen wählen. Dadurch wird ein Dialogfeld mit den im Projekt verfügbaren Datenmengen angezeigt. Wählen Sie die gewünschte Datenmenge, und bestätigen Sie mit *OK*. Beim Kopieren wird die Client-Datenmenge automatisch aktiviert.

Zur Laufzeit können Informationen aus anderen Datenmengen durch direktes Zuweisen der Daten oder, wenn es sich bei der Quelldatenmenge um eine andere Client-Datenmenge handelt, durch Replizieren kopiert werden.

Daten direkt zuweisen

Sie können der Eigenschaft *Data* einer Client-Datenmenge die Daten einer anderen Datenmenge zuweisen. Data enthält ein *OleVariant*-Datenpaket. Die Daten können von einer anderen Client-Datenmenge oder bei Verwendung eines Provider-Objekts von einer anderen Datenmenge kopiert werden. Sobald der Eigenschaft *Data* ein Datenpaket zugewiesen wurde, wird sein Inhalt automatisch in den datensensitiven Steuerelementen angezeigt, die mit der Client-Datenmenge über eine Datenquellenkomponente verknüpft sind.

Wenn Sie eine Client-Datenmenge öffnen, die über eine Provider-Komponente verfügt, werden der Eigenschaft *Data* automatisch Datenpakete zugewiesen. Weitere Informationen zu Client-Datenmengen und Provider-Objekten finden Sie im Abschnitt »Eine Client-Datenmenge mit einem Daten-Provider verwenden« auf Seite 23-15.

Wird keine Provider-Komponente verwendet, können die Daten einer anderen Client-Datenmenge durch eine einfache Zuweisung kopiert werden:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

Hinweis Beim Kopieren der Eigenschaft *Data* einer anderen Datenmenge wird auch das Änderungsprotokoll berücksichtigt. In der Kopie sind jedoch keine definierten Filter oder Bereiche enthalten. Wenn Sie dies wünschen, müssen Sie statt dessen eine Replikation durchführen.

Handelt es sich bei der Datenmenge nicht um eine Client-Datenmenge, können Sie eine Provider-Komponente erstellen, diese mit der Quelldatenmenge verknüpfen und dann die Daten kopieren:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

Hinweis Bei direkten Zuweisungen an die Eigenschaft *Data* werden die neuen Daten nicht in die vorhandenen eingefügt, sondern die vorhandenen Daten werden vollständig ersetzt.

Wenn Sie die Daten einer anderen Datenmenge mit den vorhandenen zusammenführen wollen, müssen Sie eine Provider-Komponente verwenden. Erstellen Sie dazu einen Datenmengen-Provider wie im vorhergehenden Beispiel, ordnen Sie ihn aber der Zieldatenmenge zu, und verwenden Sie die Methode *ApplyUpdates*, anstatt die Eigenschaft *Data* zu kopieren:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

Datenmengen replizieren

TClientDataSet stellt die Prozedur *CloneCursor* zur Verfügung, mit deren Hilfe Sie zur Laufzeit mit einer zweiten Ansicht einer bestimmten Client-Datenmenge arbeiten können. *CloneCursor* ermöglicht einer zweiten Client-Datenmenge, die Originaldaten mit der Client-Datenmenge gemeinsam zu verwenden. Dies ist natürlich bedeutend schneller und belegt weniger Ressourcen als das Kopieren. Da aber die Daten gemeinsam verwendet werden, können sie von der zweiten Datenmenge nicht ohne Auswirkungen auf die Originalmenge geändert werden.

CloneCursor erwartet drei Parameter. *Source* gibt die zu kopierende Client-Datenmenge an. Die letzten beiden Parameter (*Reset* und *KeepSettings*) geben an, ob außer den Daten noch weitere Informationen kopiert werden sollen. Dazu gehören sämtliche Filter, der aktuelle Index, Verknüpfungen mit einer Haupttabelle (wenn die Quelldatenmenge eine Detailmenge ist), der Wert der Eigenschaft *ReadOnly* und alle Verknüpfungen mit einer Verbindungskomponente oder Provider-Schnittstelle.

Wenn *Reset* und *KeepSettings* den Wert *False* haben, erhält die replizierte Client-Datenmenge alle Einstellungen der Quelldatenmenge. Wird *Reset* auf *True* gesetzt, werden den Eigenschaften der Zieldatenmenge Standardwerte zugewiesen (kein Index oder Filter, keine Haupttabelle, keine Verbindungs- oder Provider-Komponente, *ReadOnly* hat den Wert *False*). Hat *KeepSettings* den Wert *True*, werden die Eigenschaften der Zieldatenmenge nicht geändert.

Eine Client-Datenmenge mit einem Daten-Provider verwenden

Eine Client-Datenmenge in einer mehrschichtigen Anwendung erhält ihre Daten von einer Provider-Komponente auf dem Anwendungsserver, bearbeitet die Informationen lokal und trägt die Aktualisierungen anschließend in die Remote-Datenbank ein. Es kann auch eine Datenmenge mit einem Provider verwendet werden, der sich in derselben Anwendung befindet.

Die folgenden Abschnitte beschreiben, wie eine Client-Datenmenge mit einem Provider eingesetzt wird:

1. Einen Daten-Provider festlegen
2. Optional: Parameter vom Anwendungsserver abrufen oder
3. Optional: Die Datenmenge auf dem Anwendungsserver überschreiben
4. Die Datenmenge auf dem Anwendungsserver überschreiben
5. Beschränkungen verarbeiten , die vom Anwendungsserver empfangen wurden
6. Datensätze aktualisieren
7. Daten aktualisieren

Sie können außerdem mit Hilfe von Client-Datenmengen mit dem Provider über ein benutzerdefiniertes Ereignis kommunizieren.

Einen Daten-Provider festlegen

Einer Client-Datenmenge muß erst ein Datenmengen-Provider zugeordnet werden, damit sie Daten von einem Anwendungsserver erhalten und Aktualisierungen an diesen senden kann. Verwenden Sie dazu in einer mehrschichtigen Anwendung die Eigenschaften *RemoteServer* und *ProviderName* . In einschichtigen Anwendungen und Client-Anwendungen, die als temporäre einschichtige Anwendungen im Aktenkoffermodus eingesetzt werden, werden diese Eigenschaften nicht verwendet. Wenn Sie eine Client-Datenmenge mit einem Provider einsetzen, der in derselben Anwendung instantiiert wird, müssen Sie *RemoteServer* nicht verwenden. Sie können aber immer noch die Eigenschaft *ProviderName* angeben, wenn die Provider-Komponente denselben Eigentümer wie die Client-Datenmenge hat.

RemoteServer gibt den Namen einer Verbindungskomponente an, von der eine Liste der verfügbaren Provider abgerufen werden kann. Die Komponente befindet sich in demselben Datenmodul wie die Client-Datenmenge. Sie wird zum Einrichten und Verwalten einer Verbindung mit einem Anwendungsserver (einem Daten-Broker) verwendet. Weitere Informationen finden Sie unter »Die Struktur der Client-Anwendung« auf Seite 14-4.

Nachdem Sie einen Remote-Server angegeben haben, können Sie zur Entwurfszeit im Objektinspektor einen Provider in der Dropdown-Liste der Eigenschaft *ProviderName* auswählen. Hier sind alle lokalen Provider-Objekte des Datenmoduls aufgeführt. Zur Laufzeit können Sie zwischen den Provider-Komponenten wechseln, indem Sie *ProviderName* einen anderen Wert zuweisen.

Um einen lokalen Provider mit einem anderen Eigentümer (*Owner*) zu verwenden, müssen Sie die Zuordnung zur Laufzeit mit Hilfe der Methode *SetProvider* der Client-Datenmenge erstellen.

Parameter vom Anwendungsserver abrufen

Es gibt zwei Situationen, in denen die Client-Anwendung Parameterwerte vom Anwendungsserver abrufen muß:

- Der Client benötigt den Wert der Ausgabeparameter einer Stored Procedure.
- Der Client will die Eingabeparameter einer Abfrage oder Stored Procedure mit den aktuellen Werten einer Abfrage oder Stored Procedure auf dem Anwendungsserver initialisieren.

Die Parameterwerte werden bei einer Client-Datenmenge in der Eigenschaft *Params* gespeichert. Sie werden jedesmal entsprechend der Ausgabeparameter aktualisiert, wenn die Client-Datenmenge Daten vom Anwendungsserver abruft. In manchen Fällen benötigt die Datenmenge jedoch die Ausgabeparameter, wenn keine Daten abgerufen werden.

Dies kann mit Hilfe der Methode *FetchParams* durchgeführt werden. Die Parameter werden als Datenpaket vom Anwendungsserver zurückgegeben und der Eigenschaft *Params* der Client-Datenmenge zugewiesen.

Sie können die Eigenschaft *Params* auch in der Entwurfsumgebung initialisieren. Klicken Sie dazu mit der rechten Maustaste auf die Client-Datenmenge, und wählen Sie *Parameter holen*.

Hinweis Die Methode *FetchParams* (bzw. der Befehl Parameter holen) kann nur verwendet werden, wenn die Client-Datenmenge mit einem Provider-Objekt verbunden ist, dessen zugeordnete Datenmenge Parameter bereitstellen kann. *TDataSetProvider* muß dazu einer Abfrage oder Stored Procedure entsprechen.

Bei der Arbeit mit einem statuslosen Anwendungsserver können die Ausgabeparameter mit *FetchParams* nicht abgerufen werden. Die anderen Clients können bei dieser Art von Server die Abfrage oder Stored Procedure ändern und erneut ausführen. Dadurch werden die Ausgabeparameter vor dem Aufruf von FetchParams geändert. Rufen Sie daher die Parameter mit Hilfe der Methode *Execute* ab. Ist das Provider-Objekt einer Abfrage oder Stored Procedure zugeordnet, wird es von *Execute* angewiesen, diese auszuführen und die Ausgabeparameter anschließend zurückzugeben. Diese Werte werden dann automatisch der Eigenschaft *Params* zugewiesen.

Mit der Eigenschaft *Params* können auch Parameterwerte an den Anwendungsserver übergeben werden.

Parameter an den Anwendungsserver übergeben

Eine Client-Datenmenge kann optional Parameter an den Anwendungsserver übergeben. Auf diese Weise können Sie steuern, welche Daten in den Paketen gesendet werden. Die Parameter können folgende Informationen enthalten:

- Parameterwerte für eine Abfrage oder Stored Procedure auf dem Anwendungsserver
- Feldwerte zum Einschränken der gesendeten Datensätze

Die Parameterwerte können zur Entwurfs- oder zur Laufzeit angegeben werden. Wählen Sie in der Entwicklungsumgebung die Client-Datenmenge aus, und doppelklicken Sie im Objektinspektor auf die Eigenschaft *Params*. Dadurch wird ein Editorfenster geöffnet, in dem Sie Parameter hinzufügen, entfernen oder umstellen können. Wenn Sie im Editor einen Parameter auswählen, stehen seine Eigenschaften im Objektinspektor zur Verfügung.

Zur Laufzeit können Parameter mit der Methode *CreateParam* der Eigenschaft *Params* hinzugefügt werden. Diese Methode gibt ein Parameterobjekt mit dem angegebenen Namen, Parametertyp und Datentyp zurück. Weisen Sie dem Parameter mit der entsprechenden Eigenschaft des Objekts einen Wert zu.

Im folgenden Beispiel wird dem Parameter *CustNo der Wert 605 zugewiesen:*

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
     AsInteger := 605;
```

Wenn die Client-Datenmenge nicht aktiv ist, brauchen Sie nur ihre Eigenschaft *Active* auf *True* zu setzen, um die Parameter an den Anwendungsserver zu senden und ein entsprechendes Datenpaket zurückzuerhalten.

Hinweis Sie können die Werte von Parametern auch mit den aktuellen Einstellungen des Anwendungsservers initialisieren. Klicken Sie dazu mit der rechten Maustaste auf die Client-Datenmengenkomponente, und wählen Sie *Parameter holen*, oder rufen Sie zur Laufzeit die Methode *FetchParams* auf.

Parameter für Abfragen oder Stored Procedures senden

Wird die Provider-Komponente des Anwendungsservers für die Ergebnismenge einer Abfrage oder Stored Procedure verwendet, können Sie mit der Eigenschaft *Params* Parameterwerte angeben. Wenn die Client-Datenmenge Daten vom Anwendungsserver abruft oder mit ihrer Methode *Execute* eine Abfrage oder Stored Procedure ausführt, die keine Datenmenge zurückgibt, werden diese Parameter zusammen mit der Datenanforderung oder dem Ausführen-Befehl übergeben. Sobald die Provider-Komponente die Parameter erhält, werden sie der zugeordneten Abfrage oder Stored Procedure zugewiesen. Auf dem Anwendungsserver wird dann die Abfrage oder Stored Procedure mit diesen Parametern ausgeführt. Wurden von der Client-Datenmenge Daten angefordert, werden diese beginnend mit dem ersten Datensatz der Ergebnismenge gesendet.

Hinweis Die Parameternamen sollten immer mit den entsprechenden Parametern der Abfrage oder Stored Procedure auf dem Anwendungsserver übereinstimmen.

Datensätze durch Parameter einschränken

Wenn die Provider-Komponente des Anwendungsservers für eine Tabellenkomponente verwendet wird, können Sie mit Hilfe der Eigenschaft *Params* die an die Client-Datenmenge gesendeten Datensätze einschränken.

Jeder Parametername muß dem Namen eines Feldes in der betreffenden *TTable*-Komponente entsprechen. Die Provider-Komponente des Anwendungsservers sendet dann nur die Datensätze, deren Werte mit den übergebenen Parameterwerten übereinstimmen.

Nehmen wir als Beispiel eine Client-Anwendung, in der die Bestellungen eines bestimmten Kunden angezeigt werden. Sobald der Benutzer einen Kunden angibt, wird der Eigenschaft *Params* der Client-Datenmenge der Parameter *CustID* (das Feld kann natürlich auch anders heißen) zugewiesen. Dieser Wert legt den Kunden fest, dessen Bestellungen angezeigt werden sollen. Wenn die Client-Datenmenge Daten vom Anwendungsserver anfordert , übergibt sie diesen Parameterwert. Der Anwendungsserver sendet dann nur die Datensätze für diesen Kunden. Diese Vorgehensweise ist bedeutend effektiver, als wenn der Anwendungsserver alle Datensätze sendet und die Client-Anwendung die entsprechende Filterung durchführen muß.

Die Datenmenge auf dem Anwendungsserver überschreiben

Die Provider-Komponente auf dem Anwendungsserver ist normalerweise einer Datenmenge zugeordnet, die bestimmt, welche Daten an die Clients gesendet werden. Die Datenmenge kann eine Eigenschaft haben, die eine SQL-Anweisung zum Generieren der Daten enthält, oder sie kann eine bestimmte Tabelle oder Stored Procedure angeben.

Wenn der Provider dies zuläßt, kann die Client-Datenmenge die Eigenschaft überschreiben, welche die Daten in der Datenmenge repräsentiert. Dies ist mit Hilfe der Datenmengeneigenschaft *CommandText* möglich. Sie enthält eine SQL-Anweisung, die statt des SQL-Befehls in der Datenmenge des Providers verwendet wird, oder sie enthält den Namen einer Tabelle oder Stored Procedure, die anstelle der Tabelle oder Stored Procedure verwendet wird, welche die Datenmenge aktuell repräsentiert. Durch diesen Mechanismus kann die Client-Datenmenge dynamisch angeben, welche Daten sie benötigt.

Standardmäßig gestatten Provider-Komponenten den Client-Datenmengen nicht, auf diese Weise einen Wert für *CommandText* anzugeben. Um dies zu ermöglichen, muß der Wert *poAllowCommandText* zur Eigenschaft *Options* des Providers auf dem Anwendungsserver hinzugefügt werden. Andernfalls wird der Wert von *CommandText* ignoriert.

Die Client-Datenmenge sendet in den nachfolgend angegebenen Situationen den Wert von *CommandText* an den Provider:

- Wenn sie erstmals geöffnet wird. Nachdem die Datenmenge das erste Datenpaket vom Anwendungsserver erhalten hat, sendet sie beim Abrufen weiterer Pakete den Wert von *CommandText* nicht.
- Wenn Sie den Befehl *Execute* an den Anwendungsserver sendet.

Sie können auch zu jedem anderen Zeitpunkt einen SQL-Befehl senden oder den Namen einer Tabelle oder Stored Procedure ändern. Verwenden Sie dazu die *IAppServer*-Schnittstelle, die als Eigenschaft *AppServer* zur Verfügung steht.

Daten von einem Anwendungsserver anfordern

Die folgende Tabelle enthält die Eigenschaften und Methoden von *TClientDataSet*, die bestimmen, wie in einer mehrschichtigen Anwendung Daten von einem Anwendungsserver abgerufen werden:

Tabelle 24.2 Eigenschaften und Methoden von Client-Datenmengen zur Behandlung von Datenanforderungen

Eigenschaft oder Methode	Zweck
FetchOnDemand (Eigenschaft)	Legt fest, ob eine Client-Datenmenge Daten bei Bedarf automatisch abruft oder ob sie es der Anwendung überläßt, die Funktionen *GetNextPacket*, FetchBlobs und *FetchDetails* der Client-Datenmenge aufzurufen.
PacketRecords (Eigenschaft)	Gibt den Typ oder die Anzahl der Datensätze an, die in den einzelnen Datenpaketen zurückgeliefert werden sollen.
GetNextPacket (Methode)	Ruft das nächste Datenpaket vom Anwendungsserver ab.
FetchBlobs (Methode)	Ruft alle BLOB-Felder des aktuellen Datensatzes ab, wenn der Anwendungsserver diese nicht automatisch sendet.
FetchDetails (Methode)	Ruft verschachtelte Detailmengen des aktuellen Datensatzes ab, wenn der Anwendungsserver diese nicht automatisch sendet.

In der Voreinstellung holt eine Client-Datenmenge alle Datensätze vom Anwendungsserver. Mit *PacketRecords* und *FetchOnDemand* können Sie steuern, auf welche Weise die Daten übernommen werden.

PacketRecords legt entweder fest, wie viele Datensätze gleichzeitig übertragen werden, oder bestimmt den Typ der Datensätze. Die Voreinstellung -1 hat zur Folge, daß alle Datensätze gleichzeitig übertragen werden, und zwar entweder, wenn die Anwendung zum ersten Mal geöffnet wird, oder wenn sie explizit *GetNextPacket* aufruft. Wenn *PacketRecords* den Wert -1 hat, braucht die Client-Datenmenge nur ein einziges Mal Daten abzurufen, weil dadurch alle vorhandenen Daten übertragen werden.

Wenn die Daten in kleinen Blöcken übertragen werden sollen, weisen Sie *PacketRecords* die gewünschte Anzahl der Datensätze zu. Die folgende Anweisung setzt beispielsweise die Größe jedes Datenpakets auf 10 Datensätze:

```
ClientDataSet1.PacketRecords := 10;
```

Diese Vorgehensweise beim Abrufen von Daten nennt man *inkrementelles Abrufen*. Ist *PacketRecords* größer als 0, ruft die Client-Datenmenge bei Bedarf *GetNextPacket* auf. Dies wird von der Client-Datenmenge standardmäßig durchgeführt. Die neuen Pakete werden am Ende der bereits vorhandenen Daten hinzugefügt.

GetNextPacket gibt die Anzahl der übertragenen Datensätze zurück. Wenn der Rückgabewert dem Wert von *PacketRecords* entspricht, ist das Ende der verfügbaren Datensätze noch nicht erreicht. Ist der Rückgabewert größer als 0, aber kleiner als *PacketRecords*, wurden alle Datensätze übertragen. Wenn *GetNextPacket* 0 zurückgibt, sind keine weiteren Datensätze vorhanden.

Achtung Das inkrementelle Abrufen ist nur möglich, wenn die Statusinformationen im Remote-Datenmodul gespeichert werden. Sie dürfen daher MTS nicht verwenden, und das Remote-Modul muß so konfiguriert werden, daß jede Client-Anwendung über eine eigene Modulinstanz verfügt. Informationen über das inkrementelle Abrufen bei statuslosen Remote-Datenmodulen finden Sie unter »Statusinformationen in Remote-Datenmodulen unterstützen« auf Seite 14-30.

Sie können mit Hilfe von *PacketRecords* auch Metadaten über eine Datenbank vom Anwendungsserver abrufen. Setzen Sie zu diesem Zweck die Eigenschaft *PacketRecords* auf 0.

Das automatische Abrufen von Daten wird durch die Eigenschaft *FetchOnDemand* gesteuert. Wenn *FetchOnDemand True* ist (Voreinstellung), ist die automatische Übertragung aktiviert. Andernfalls muß die Anwendung Datensätze explizit mit *GetNextPacket* abrufen.

Deaktivieren Sie *FetchOnDemand* in Anwendungen, in denen sehr große Nur-Lesen-Datenmengen angezeigt werden müssen. Auf diese Weise ist sichergestellt, daß die Client-Datenmenge nicht mehr Daten lädt, als in den Speicher passen. Die Client-Datenmenge gibt ihren Puffer zwischen den Ladeoperationen mit der Methode *EmptyDataSet* frei. Diese Technik ist jedoch nicht geeignet, wenn die Client-Anwendung Aktualisierungen an den Anwendungsserver senden muß.

Beschränkungen verarbeiten

Client-Datenmengen unterstützen die folgenden Arten von Beschränkungen:

- Vom Anwendungsserver gesendete Beschränkungen.
- Vom Client bereitgestellte benutzerdefinierte Beschränkungen.

Server-Beschränkungen verarbeiten

Standardmäßig gibt der Anwendungsserver die Beschränkungen und Standardausdrücke des Datenbankservers an die Client-Datenmengen weiter, in denen sie für Datenbearbeitungen durch Benutzer verwendet werden können. Sind die Beschränkungen aktiv, werden Bearbeitungen in einer Client-Anwendung, die Beschränkungen verletzen und vom Datenbankserver zurückgewiesen werden, auf der Client-Seite überprüft und nicht an den Anwendungsserver weitergegeben. Dadurch treten bedeutend weniger Aktualisierungsfehler beim Eintragen in die Datenbank auf.

Das Importieren der Server-Beschränkungen und -Ausdrücke ist sehr hilfreich, um die Datenintegrität über Plattformen und Anwendungen hinweg aufrechtzuerhalten. Es gibt aber Situationen, in denen Sie diese Funktion vorübergehend deaktivieren müssen. Basiert eine Beschränkung beispielsweise auf dem aktuell größten Wert in einem Feld und ruft die Client-Datenmenge mehrere Pakete mit Datensätzen ab, kann sich der größte Feldwert auf dem Client von dem auf dem Datenbankserver unterscheiden, und die Beschränkungen werden nicht unter identischen Bedingungen angewendet. Es ist auch möglich, daß ein Filter nicht mit den Beschränkungsbedingungen übereinstimmt, wenn eine Client-Anwendung bei aktivierten Beschränkungen einen Filter für die Datensätze verwendet. In diesen Fällen sollten die Server-Beschränkungen vor dem Anfordern der Daten deaktiviert werden.

Mit der Methode *DisableConstraints* einer Client-Datenmenge können die Beschränkungen vorübergehend deaktiviert werden. Durch jeden Aufruf wird dann der Referenzzähler inkrementiert. Solange dieser Wert größer als Null ist, werden die Beschränkungen nicht angewendet.

Um die Beschränkungen wieder zu aktivieren, verwenden Sie die Datenmengenmethode *EnableConstraints*. Durch sie wird der Referenzzähler dekrementiert. Erreicht dieser den Wert Null, werden die Beschränkungen wieder angewendet.

Tip Verwenden Sie *DisableConstraints* und *EnableConstraints* immer paarweise. Auf diese Weise können Sie sicherstellen, daß die Beschränkungen wie beabsichtigt angewendet werden.

Hinweis Mit den Methoden *DisableConstraints* und *EnableConstraints* können Sie steuern, ob auf die Daten der Client-Datenmenge Beschränkungen angewendet werden. Sie haben aber keinen Einfluß auf die Beschränkungsinformationen, die vom Anwendungsserver in die Datenpakete aufgenommen werden. Mit Hilfe der Eigenschaft *Constraints* der Provider-Komponente kann verhindert werden, daß der Server Beschränkungen sendet. Weitere Informationen zur Verarbeitung der Server-Beschränkungen finden Sie unter »Server-Beschränkungen« auf Seite 15-11. Informationen zum Arbeiten mit den Beschränkungen nach dem Importieren finden Sie unter »Server-Beschränkungen« auf Seite 19-25.

Benutzerdefinierte Beschränkungen hinzufügen

Sie können als Ergänzung der Server-Beschränkungen anwendungsspezifische Beschränkungen definieren. Verwenden Sie dazu die folgenden beiden Eigenschaften der Feldkomponenten in der Client-Datenmenge:

- Mit der Eigenschaft *DefaultExpression* kann ein Standardwert angegeben werden, der dem Feld automatisch zugewiesen wird, wenn der Benutzer keine Eingabe vornimmt. Dieser Wert setzt den vom Server angegebenen Standardwert außer Kraft, da er zugewiesen wird, bevor die aktualisierten Daten wieder an den Anwendungsserver gesendet werden.
- Mit der Eigenschaft *CustomConstraint* können Sie eine Beschränkungsbedingung festlegen, die erfüllt sein muß, damit das Feld in die Datenbank eingetragen werden kann. Diese Beschränkungen werden zusätzlich zu den vom Server importierten verwendet. Weitere Informationen finden Sie unter »Selbstdefinierte Beschränkungen« auf Seite 19-24.

Sie können außerdem Beschränkungen auf Datensatzebene mit Hilfe der Eigenschaft *Constraints* der Client-Datenmenge erstellen. *Constraints* ist eine Kollektion von Objekten des Typs *TCheckConstraint*, von denen jedes einer Bedingung entspricht. Definieren Sie mit *CustomConstraint* eigene Beschränkungen, die beim Eintragen der Datensätze überprüft werden.

Datensätze aktualisieren

Wenn eine Client-Anwendung mit einem Anwendungsserver verbunden ist, arbeiten die Client-Datenmengen mit einer vom Server erhaltenen lokalen Kopie der Daten. Diese Kopien können dann mit den datensensitiven Steuerelementen der Client-Anwendung angezeigt und bearbeitet werden. Sind die Server-Beschränkungen aktiviert, werden sie für die Benutzereingaben verwendet. Die vorgenommenen Aktualisierungen werden von der Client-Datenmenge vorübergehend in einem internen Änderungsprotokoll verwaltet. Dessen Inhalt wird dann als Datenpaket in der Eigenschaft *Delta* gespeichert. Um die Änderungen dauerhaft zu machen, müssen sie von der Client-Datenmenge in die Datenbank eingetragen werden.

Das Senden der Aktualisierungen an den Server erfolgt in vier Schritten:

1. Die Client-Anwendung ruft die Methode *ApplyUpdates* einer Client-Datenmenge auf. Die Methode übergibt den Inhalt der Datenmengeneigenschaft *Delta*, ein Datenpaket mit den aktualisierten, eingefügten und gelöschten Datensätzen, an den Anwendungsserver.
2. Die Provider-Komponente des Anwendungsservers trägt die Aktualisierungen in die Datenbank ein. Dabei werden alle problematischen Datensätze zwischengespeichert, die nicht auf Server-Ebene bereinigt werden können. Wie der Server die Aktualisierungen in die Datenbank schreibt, wird unter »Auf Datenanforderungen des Client reagieren« auf Seite 15-5 beschrieben.
3. Die Provider-Komponente sendet die nicht bereinigten Datensätze in einem *Result*-Datenpaket an den *Client* zurück. Das Paket enthält alle Datensätze, die nicht

eingetragen werden konnten sowie Fehlerinformationen wie Fehlermeldungen und Fehlercodes.

4 Die Client-Anwendung versucht, die im Result-Paket empfangenen Aktualisierungsfehler satzweise zu beheben.

Aktualisierungen eintragen

Die in der Client-Anwendung an den lokalen Daten vorgenommenen Änderungen müssen explizit mit der Datenmengenmethode *ApplyUpdates* an den Anwendungsserver übertragen werden. Dabei wird der Inhalt des Änderungsprotokolls als Datenpaket (*Delta*) gesendet.

ApplyUpdates erhält als einzigen Parameter *MaxErrors*. Dieser Wert bestimmt, nach wie vielen Fehlern der Anwendungsserver die Aktualisierung abbricht. Wenn *MaxErrors* den Wert 0 hat, wird der gesamte Aktualisierungsvorgang beendet, sobald auf dem Anwendungsserver ein Aktualisierungsfehler auftritt. Hat *MaxErrors* den Wert -1, wird jede Fehleranzahl toleriert, und das Änderungsprotokoll enthält alle Datensätze, die nicht erfolgreich verarbeitet werden konnten. Enthält MaxErrors einen positiven Wert und treten mehr als *MaxErrors* Fehler auf, wird die Aktualisierung abgebrochen. Wenn weniger als *MaxErrors* Fehler auftreten, werden alle erfolgreich bearbeiteten Datensätze automatisch aus dem Änderungsprotokoll der Client-Datenmenge gelöscht.

ApplyUpdates gibt die Anzahl der aufgetretenen Fehler zurück. Dieser Wert ist immer kleiner oder gleich *MaxErrors* plus eins. Ihm können Sie entnehmen, wie viele Datensatz nicht in die Datenbank geschrieben werden konnten. Diese Datensätze werden vom Anwendungsserver an die Client-Datenmenge zurückgesendet.

Die Client-Datenmenge ist für das Bereinigen der problematischen Datensätze verantwortlich. *ApplyUpdates* ruft die Methode *Reconcile* auf, um die Aktualisierungen in die Datenbank zu schreiben. Reconcile ist eine Fehlerbehandlungsroutine, die die Funktion *ApplyUpdates* einer Provider-Komponente auf dem Anwendungsserver indirekt aufruft. Die Funktion trägt die Aktualisierungen in die Datenbank ein und versucht, etwaige Fehler zu korrigieren. Datensätze, die aufgrund von Fehlern nicht eingetragen werden konnten, werden an die Methode *Reconcile* der Client-Datenmenge zurückgesendet. *Reconcile* versucht dann, die verbleibenden Fehler durch den Aufruf der Behandlungsroutine für das Ereignis *OnReconcileError* zu korrigieren. Dazu müssen Sie diese Ereignisbehandlungsroutine mit Quelltext ausstatten. Informationen über *OnReconcileError* finden Sie unter »Aktualisierungsfehler bereinigen« auf Seite 24-24.

Abschließend entfernt *Reconcile* erfolgreich durchgeführte Änderungen aus dem Änderungsprotokoll und aktualisiert die Eigenschaft *Data*, so daß diese die aktualisierten Datensätze enthält. Nach Beendigung von *Reconcile* meldet *ApplyUpdates* die Anzahl der aufgetretenen Fehler.

Hinweis Wenn Sie MTS-Transaktionen einsetzen oder die Instanz eines Anwendungsserver mit anderen Clients gemeinsam verwenden, können Sie vor oder nach dem Eintragen der Aktualisierungen mit der Provider-Komponente des Servers persistente Statusinformationen austauschen. Die Client-Datenmenge empfängt dann ein *Before-*

ApplyUpdates -Ereignis, bevor die Aktualisierungen übertragen werden. Dies ermöglicht das Senden der persistenten Statusinformationen an den Server. Nach dem Eintragen der Aktualisierungen (aber vor der Berichtigung) empfängt die Client-Datenmenge ein *AfterApplyUpdates*-Ereignis. Dadurch kann auf die vom Anwendungsserver gesendeten Statusinformationen reagiert werden.

Aktualisierungsfehler bereinigen

Die Provider-Komponente des Anwendungsservers gibt die zu bereinigenden Datensätze und die Fehlerinformationen in einem Datenpaket an die Client-Datenmenge zurück. Ist der gelieferte Fehlerzähler größer als Null, wird für jeden Datensatz im Paket das Ereignis *OnReconcileError* der Client-Datenmenge ausgelöst.

Sie sollten in jedem Fall die Ereignisbehandlungsroutine für *OnReconcileError* implementieren, auch wenn Sie hier nur die vom Server zurückgegebenen Datensätze verwerfen. Übergeben Sie der Routine dabei vier Parameter:

- *DataSet*: Die betreffende Client-Datenmenge. Mit ihren Methoden können Informationen zu den problematischen Datensätzen abgerufen und die entsprechenden Berichtigungen durchgeführt werden. Insbesondere mit den Eigenschaften *CurValue*, *OldValue* und *NewValue* der Felder im aktuellen Datensatz kann die Ursache des Aktualisierungsproblems ermittelt werden. Sie dürfen aber in einer Ereignisbehandlungsroutine für *OnReconcileError* keine Datenmengenmethoden aufrufen, die den aktuellen Datensatz ändern.
- *E*: Ein Objekt des Typs *EReconcileError*, das den aufgetretenen Fehler angibt. Mit dieser Exception kann eine Fehlermeldung oder die Ursache des Aktualisierungsfehlers ermittelt werden.
- *UpdateKind*: Die Art der Aktualisierung, die zu dem Fehler geführt hat. Mögliche Werte sind *ukModify* (das Problem ist beim Aktualisieren eines geänderten Datensatzes aufgetreten), *ukInsert* (das Problem ist beim Einfügen eines neuen Datensatzes aufgetreten) oder *ukDelete* (das Problem ist beim Löschen eines Datensatzes aufgetreten).
- *Action*: Ein **var**-Parameter, mit dem die Aktion angegeben werden kann, die nach dem Beenden der Ereignisbehandlungsroutine ausgeführt wird. Beim Eintritt in die Routine wird *Action* auf die Aktion gesetzt, die vom Anwendungsserver zur Fehlerbereinigung vorgenommen wurde. Weisen Sie dem Parameter in Ihrer Routine einen der folgenden Werte zu:
  - *raSkip*: Der Datensatz wird nicht bearbeitet, bleibt aber im Änderungsprotokoll.
  - *raAbort*: Die gesamte Verarbeitung wird beendet.
  - *raMerge*: Die fehlgeschlagene Änderung wird mit dem entsprechenden Datensatz des Servers zusammengeführt. Dies ist aber nur möglich, wenn keines der vom Benutzer geänderten Felder auf dem Server aktualisiert wurde.
  - *raCorrect*: Die Aktualisierung im Änderungsprotokoll wird durch die Feldwerte des Datensatzes in der Ereignisbehandlungsroutine ersetzt.
  - *raCancel*: Die Änderungen des aktuellen Datensatzes in der Client-Datenmenge werden verworfen und die ursprünglichen Werte wiederhergestellt.

- *raRefresh*: Die Feldwerte des aktuellen Datensatzes werden mit den Werten auf dem Server aktualisiert.

In der folgenden Behandlungsroutine für *OnReconcileError* wird das Dialogfeld zur Fehlerbereinigung aus der Unit *RecError* in der Objektablage verwendet (dazu muß die Unit in die **uses**-Klausel aufgenommen werden).

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TClientDataSet; E: EReconcileError;
UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Daten aktualisieren

Client-Anwendung arbeiten mit einer Kopie der Daten vom Anwendungsserver. Diese können natürlich von anderen Benutzern geändert werden, so daß die Informationen des Clients mit der Zeit immer weniger mit den zugrundeliegenden Daten übereinstimmen.

Wie die anderen Datenmengen verfügen Client-Datenmengen über die Methode *Refresh* . Mit ihr können die Datensätze mit den aktuellen Werten auf dem Server aktualisiert werden. Rufen Sie die Methode aber nur auf, wenn das Änderungsprotokoll keine Bearbeitungen enthält. Sie erhalten sonst eine Exception.

Client-Anwendungen können ihre Daten auch aktualisieren, ohne das Änderungsprotokoll zu beeinträchtigen. Dazu muß die Methode *RefreshRecord* aufgerufen werden. Sie aktualisiert im Gegensatz zu *Refresh* nur den aktuellen Datensatz. Mit *RefreshRecord* können die ursprünglich vom Anwendungsserver abgerufenen Werte aktualisiert werden, ohne die Änderungen im Protokoll zu beeinträchtigen.

Achtung Die Methode *RefreshRecord* ist nicht für alle Situationen geeignet. Sind beispielsweise durch Eingaben mehrerer Benutzer Konflikte entstanden, werden diese durch *RefreshRecord* verdeckt. Wenn die Client-Anwendung ihre Aktualisierungen einträgt, tritt dann kein Aktualisierungsfehler auf, und der Konflikt kann nicht beseitigt werden.

Letzteres kann verhindert werden, indem in der Client-Anwendung vor dem Aufrufen von *RefreshRecord* geprüft wird, ob anstehende Aktualisierungen vorhanden sind. Im folgenden Beispiel wird eine Exception ausgelöst, wenn ein geänderter Datensatz aktualisiert werden soll:

```
if ClientDataSet1.UpdateStatus <> usUnModified then
  raise Exception.Create('Daten eintragen, bevor aktueller Datensatz aktualisiert wird.');
ClientDataSet1.RefreshRecord;
```

Mit Provider-Komponenten kommunizieren

Client-Datenmengen bieten viele Möglichkeiten, die Kommunikation zwischen Client-Anwendung und Anwendungsserver zu konfigurieren. Vor und nach jedem Aufruf der Methode *IAppServer*, der an den Provider der Datenmenge gerichtet ist, empfängt die Client-Datenmenge spezielle Ereignisse, die eine Kommunikation mit

dem Provider ermöglichen. Diese Ereignisse werden mit den entsprechenden Ereignissen der Provider-Komponente abgeglichen. Es treten beispielsweise folgende Ereignisse auf, wenn die Methode *ApplyUpdates* der Client-Datenmenge aufgerufen wird:

1. Die Client-Datenmenge empfängt ein *BeforeApplyUpdates*-Ereignis. In der zugehörigen Behandlungsroutine werden anwendungsspezifische Informationen in einem *OleVariant*-Objekt mit dem Namen *OwnerData* gespeichert.
2. Die Provider-Komponente empfängt ein *BeforeApplyUpdates*-Ereignis und reagiert durch Aktualisieren des Wertes von *OwnerData* mit neuen Informationen.
3. Die Provider-Komponente erstellt ein Datenpaket und löst die zugehörigen Ereignisse aus.
4. Die Provider-Komponente empfängt ein *AfterApplyUpdates*-Ereignis und reagiert durch Aktualisieren des Wertes von *OwnerData* mit Daten für den Client.
5. Die Client-Datenmenge empfängt ein *AfterApplyUpdates*-Ereignis und reagiert auf den zurückgegebenen Wert von OwnerData.

Bei jedem anderen Aufruf von *IAppServer* werden entsprechende *BeforeXXX*- und *AfterXXX*-Ereignisse ausgelöst, mit deren Hilfe Sie die Kommunikation zwischen Client und Server anpassen können.

Die Client-Datenmenge verfügt außerdem über die spezielle Methode *DataRequest*. Sie ermöglicht die anwendungsspezifische Kommunikation mit dem Provider. Beim Aufruf von *DataRequest* wird ein Parameter des Typs *OleVariant* mit den gewünschten Informationen übergeben. Dadurch wird ein *OnDataRequest* -Ereignis in der Provider-Komponente ausgelöst. In der Behandlungsroutine können Sie dann bestimmte Aktionen durchführen und einen Wert an die Client-Datenmenge zurückgeben.

Eine Client-Datenmenge mit unstrukturierten Daten verwenden

Client-Datenmengen können auch unabhängig von einer Provider-Komponente verwendet werden. Dies ist beispielsweise in einer Datenbankanwendung mit unstrukturierten Dateien und Anwendungen im Aktenkoffer-Modell der Fall. Ohne einen Daten-Provider erhält die Client-Anwendung jedoch keine Tabellendefinitionen und Daten vom Server, und es ist auch kein Server zum Eintragen der Aktualisierungen vorhanden. Die Datenmenge muß daher folgende Operationen selbst durchführen:

- Tabellen definieren und erstellen
- Gespeicherte Daten laden
- Änderungen in die Daten schreiben
- Daten speichern

Eine neue Datenmenge erstellen

Sie können auf drei Arten Client-Datenmengen erstellen, die ihre Daten nicht von einer Provider-Komponente erhalten:

- Sie können eine vorhandene Datenmenge kopieren (zur Entwurfs- oder Laufzeit). Informationen hierzu finden Sie unter »Daten aus einer anderen Datenmenge kopieren« auf Seite 24-13.
- Sie können eine neue Client-Datenmenge definieren, indem Sie persistente Feldkomponenten erstellen und dann im lokalen Menü *Datenmenge erstellen* wählen. Informationen hierzu finden Sie unter »Neue Datenmengen mit persistenten Feldern erstellen« auf Seite 13-16.
- Sie können eine neue Client-Datenmenge anhand von Feld- und Indexdefinitionen definieren und erstellen. Informationen hierzu finden Sie unter »Datenmengen mit Feld- und Indexdefinitionen erstellen« auf Seite 13-16.

Daten aus einer Datei oder einem Stream laden

Zum Laden von Daten aus einer Datei rufen Sie die Methode *LoadFromFile* der Client-Datenmenge auf. LoadFromFile erwartet einen String-Parameter, der angibt, aus welcher Datei die Daten abgerufen werden sollen. Bei dem Dateinamen kann es sich auch um einen vollständigen Pfadnamen handeln. Wenn Sie die Daten immer aus derselben Datei laden, verwenden Sie besser die Eigenschaft *FileName*. Hat *FileName* einen Wert, werden die Daten beim Öffnen der Client-Datenmenge automatisch aus dieser Datei geladen.

Um Daten aus einem Stream zu laden, rufen Sie die Methode *LoadFromStream* der Client-Datenmenge auf. Der Methode muß als Parameter ein Stream-Objekt übergeben werden.

Die mit *LoadFromFile* (*LoadFromStream*) zu ladenden Daten müssen zuvor mit der Methode *SaveToFile* (*SaveToStream*) in einem für die Client-Datenmenge geeigneten Format gespeichert werden. Informationen über das Speichern von Daten in einer Datei oder einem Stream finden Sie unter »Daten in einer Datei oder einem Stream speichern« auf Seite 24-28.

Beim Aufruf von *LoadFromFile* oder *LoadFromStream* werden alle Daten in der Datei in die Eigenschaft *Data* eingelesen. Die Bearbeitungen im Änderungsprotokoll werden der Eigenschaft *Delta* zugewiesen.

Änderungen in die Daten schreiben

Wenn Sie die Daten in einer Client-Datenmenge bearbeiten, werden sämtliche Änderungen in das Änderungsprotokoll eingetragen. Die Originaldaten werden dadurch aber nicht aktualisiert.

Um die Änderungen dauerhaft einzutragen, rufen Sie die Methode *MergeChangeLog* auf. Sie überschreibt die Datensätze in *Data* mit den geänderten Feldwerten aus dem Protokoll.

Nach der Ausführung von *MergeChangeLog* enthält *Data* eine Mischung aus den vorhandenen Daten und allen Änderungen, die im Protokoll aufgezeichnet wurden. Diese Mischung wird zur neuen Grundlage in Data, auf die sich zukünftige Änderungen beziehen. *MergeChangeLog* löscht das Änderungsprotokoll aller Datensätze und belegt die Eigenschaft *ChangeCount* mit 0.

Achtung Rufen Sie *MergeChangeLog* nicht in Client-Anwendungen auf, die mit einem Anwendungsserver verbunden sind. Tragen Sie in diesem Fall die Aktualisierungen mit *ApplyUpdates* in die Datenbank ein. Weitere Informationen hierzu finden Sie im Abschnitt »Aktualisierungen eintragen« auf Seite 24-23.

Hinweis Die Änderungen können auch in die Daten einer anderen Client-Datenmenge geschrieben werden, wenn diese ursprünglich die Informationen für die Eigenschaft *Data* bereitgestellt hat. Dazu müssen aber eine Provider- und eine Resolver-Komponente verwendet werden. Ein Beispiel hierzu finden Sie unter »Daten direkt zuweisen« auf Seite 24-14.

Daten in einer Datei oder einem Stream speichern

Bei einer Client-Datenmenge in einer einschichtigen Anwendung werden die Änderungen nur in den Hauptspeicher geschrieben. Um sie dauerhaft einzutragen, müssen Sie die Daten mit der Methode *SaveToFile*> auf der Festplatte speichern.

SaveToFile wird ein String mit dem Namen der gewünschten Datei übergeben. Es kann auch ein vollständiger Pfadname angegeben werden. Ist die Datei bereits vorhanden, wird ihr gesamter Inhalt überschrieben.

Wenn Sie die Daten immer in dieselbe Datei schreiben, verwenden Sie besser die Eigenschaft *FileName*. Hat *FileName* einen Wert, werden die Daten beim Schließen der Client-Datenmenge automatisch in diese Datei geschrieben.

Mit der Methode *SaveToStream* können die Daten auch in einen Stream geschrieben werden. *SaveToStream* muß als Parameter ein Stream-Objekt übergeben werden.

Hinweis Wenn sich beim Speichern einer Client-Datenmenge noch Bearbeitungen im Änderungsprotokoll befinden, werden diese nicht in die Daten geschrieben. Beim erneuten Laden der Daten mit *LoadFromFile* oder *LoadFromStream* enthält das Protokoll immer noch diese Bearbeitungen. Dies ist für Anwendungen wichtig, die das Aktenkoffer-Modell unterstützen, da die Aktualisierungen hier möglicherweise zu einer Provider-Komponente auf dem Anwendungsserver gesendet werden.

Hinweis Beim Speichern mit *SaveToFile* gehen alle in der Client-Datenmenge definierten Indizes verloren.

Kapitel

25

Zwischengespeicherte Aktualisierungen

Mit zwischengespeicherten Aktualisierungen können Daten aus einer Datenbank abgerufen, lokal zwischengespeichert und bearbeitet und dann insgesamt in die Datenbank zurückgeschrieben werden. Wenn die Zwischenspeicherung aktiviert ist, werden Aktualisierungen einer Datenmenge (z.B. nach Änderungen oder nach dem Löschen von Datensätzen) nicht direkt in der zugrundeliegenden Tabelle, sondern in einem internen Zwischenspeicher abgelegt. Am Ende wird durch die Anwendung eine Methode aufgerufen, welche die zwischengespeicherten Änderungen in die Datenbank schreibt und den Zwischenspeicher leert.

Dieses Kapitel beschreibt, wann und wie zwischengespeicherte Aktualisierungen eingesetzt werden. Außerdem lernen Sie die Komponente *TUpdateSQL* kennen, mit der auch Datenmengen aktualisiert werden können, bei denen normalerweise keine Aktualisierung möglich ist.

Wann werden zwischengespeicherte Aktualisierungen eingesetzt?

Zwischengespeicherte Aktualisierungen verringern den konkurrierenden Zugriff auf Remote-Datenbankserver durch folgende Faktoren:

- Der Zeitbedarf einer Transaktion wird minimiert.
- Der Datenverkehr im Netzwerk wird minimiert.

Trotz dieser Vorteile sind zwischengespeicherte Aktualisierungen nicht für alle Client-Datenbankanwendungen die ideale Lösung. Dabei spielen folgende Überlegungen eine Rolle:

- **Zwischengespeicherten Daten sind für die Anwendung lokal und unterliegen keiner Transaktionskontrolle.** In einer stark belasteten Client/Server-Umgebung

ergeben sich verschiedene Konstellationen, auf die sich Ihre Anwendung einstellen muß:

- Andere Anwendungen können auf die Server-Daten zugreifen und diese ändern, während Ihre Anwendung mit der lokalen Kopie der Daten arbeitet.
- Für andere Anwendungen sind die von Ihrer Anwendung vorgenommenen Änderungen erst nach dem Commit sichtbar.

- **In Haupt/Detail-Beziehungen kann sich die Reihenfolge, in der zwischengespeicherte Aktualisierungen in die Datenbank geschrieben werden, als kritisch erweisen**. Dies gilt besonders bei verschachtelten Haupt/Detail-Beziehungen, in denen eine Detailtabelle die Haupttabelle einer anderen Detailtabelle ist usw.
- **Bei Datenmengen, die aus einer schreibgeschützten Abfrage hervorgehen, erfordern zwischengespeicherte Aktualisierungen die Verwendung von Aktualisierungsobjekten.**

Die Datenzugriffskomponenten stellen Methoden für die Zwischenspeicherung von Aktualisierungen und die Steuerung von Transaktionen bereit. Damit lassen sich die beschriebenen Situationen bewältigen. Trotzdem müssen Sie überprüfen, ob Ihre Anwendung in jedem möglichen Szenario die erwartete Leistung bringt. Schließlich stellt jede Arbeitsumgebung ihre eigenen Anforderungen.

Die Realisierung im Überblick

Dieser Abschnitt ist wesentlich für das Verständnis des restlichen Kapitels. Wenn Sie noch nie mit zwischengespeicherten Aktualisierungen gearbeitet haben, können Sie sich hier einen Überblick verschaffen, wie dieses Konzept in eine Anwendung eingebunden wird.

Die folgenden Schritte sind dazu erforderlich:

1. **Aktivieren Sie die Zwischenspeicherung von Aktualisierungen**. Dadurch starten Sie eine Nur-Lesen-Transaktion, die alle benötigten Daten vom Server abruft und dann beendet wird. Eine lokale Kopie der Daten wird zu Anzeige- und Bearbeitungszwecken im Speicher gehalten. Weitere Informationen über diesen Schritt finden Sie unter »Die Zwischenspeicherung aktivieren und deaktivieren« auf Seite 25-3
2. **Zeigen Sie die lokale Kopie der Daten an**. Der Benutzer kann Datensätze einfügen und löschen. Sowohl das Original als auch die bearbeitete Version der Daten bleiben im Speicher. Weitere Informationen über Anzeige und Bearbeitung finden Sie unter »Zwischengespeicherte Aktualisierungen zurückschreiben« auf Seite 25-4
3. **Rufen Sie bei Bedarf weitere Datensätze ab** (z.B. wenn der Benutzer einen Bildlauf durchführt). Jede Abrufoperation erfolgt im Kontext einer eigenen, zeitlich begrenzten Nur-Lesen-Transaktion (eine Anwendung kann optional auch alle Datensätze auf einmal abrufen). Weitere Informationen über das Abrufen aller Datensätze finden Sie unter »Datensätze abrufen« auf Seite 25-4.

4 **Setzen Sie die Anzeige und die Bearbeitung fort**, bis die Änderungen abgeschlossen sind.

5 **Schreiben Sie die lokal zwischengespeicherten Daten in die Datenbank** oder verwerfen Sie die Änderungen. Für jeden zurückgeschriebenen Datensatz wird ein *OnUpdateRecord*-Ereignis ausgelöst. Wenn ein Fehler auftritt, wird das Ereignis *OnUpdateError* ausgelöst, wodurch die Anwendung die Möglichkeit hat, den Fehler zu korrigieren und mit der Aktualisierung der Datenbank fortzufahren. Am Ende werden alle Datensätze, die in der Datenbank gespeichert werden konnten, aus dem lokalen Zwischenspeicher gelöscht. Weitere Informationen über das Zurückschreiben der Aktualisierungen in die Datenbank finden Sie unter »Zwischengespeicherte Aktualisierungen zurückschreiben« auf Seite 25-4.

   Wenn eine Anwendung die Aktualisierungen nicht zurückschreibt, sondern verwirft, wird die lokale Kopie der Daten ohne weitere Aktionen aus dem Speicher entfernt. Ausführliche Informationen über das Verwerfen von Aktualisierungen finden Sie unter »Anstehende Aktualisierungen verwerfen« auf Seite 25-8.

Die Zwischenspeicherung aktivieren und deaktivieren

Die Zwischenspeicherung von Aktualisierungen wird durch die Eigenschaft *CachedUpdates* einer Datenmenge (*TTable*, *TQuery* oder *TStoredProc*) aktiviert und deaktiviert. Die Voreinstellung für *CachedUpdates* ist *False*, also deaktiviert.

Hinweis Client-Datenmengen legen Aktualisierungen immer in einem Zwischenspeicher ab. Sie haben keine Eigenschaft *CachedUpdates*, da bei ihnen die Zwischenspeicherung nicht deaktiviert werden kann.

Wenn Aktualisierungen zwischengespeichert werden sollen, muß *CachedUpdates* zur Entwurfszeit (im Objektinspektor) oder zur Laufzeit auf *True* gesetzt werden. Dadurch wird das Ereignis *OnUpdateRecord* der Datenmenge ausgelöst (wenn Sie ihm Quelltext zugewiesen haben). Weitere Informationen über das Ereignis *OnUpdateRecord* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25.

Die folgende Anweisung aktiviert die Zwischenspeicherung von Aktualisierungen zur Laufzeit:

```
CustomersTable.CachedUpdates := True;
```

Wenn die Zwischenspeicherung aktiviert ist, wird eine Kopie aller benötigten Datensätze in einem lokalen Zwischenspeicher gehalten. Die Anzeige und die Bearbeitung der Daten erfolgt mit dieser lokalen Kopie. Alle Änderungs-, Einfüge- und Löschoperationen werden ebenfalls zwischengespeichert. Sie bleiben im Speicher, bis der lokale Zwischenspeicher in die Datenbank zurückgeschrieben wird. Alle erfolgreich zurückgeschriebenen Datensätze werden aus dem Zwischenspeicher gelöscht.

Hinweis Die Zwischenspeicherung ist nach dem Zurückschreiben weiterhin aktiv.

Um die Zwischenspeicherung für eine Datenmenge zu deaktivieren, setzen Sie *CachedUpdates* auf *False*. Dabei werden nicht zurückgeschriebene Änderungen ohne Rückfrage verworfen. Ihre Anwendung sollte daher die Eigenschaft *UpdatesPending* überprüfen, bevor sie die Zwischenspeicherung deaktiviert. Das folgende Pro-

grammfragment bittet um eine Bestätigung, bevor die Zwischenspeicherung deaktiviert wird:

```
if (CustomersTable.UpdatesPending)
   if (Application.MessageBox('Anstehende Aktualisierungen verwerfen?',
                              'Nicht eingetragene Änderungen',
                              MB_YES + MB_NO) = IDYES) then
    CustomersTable.CachedUpdates = False;
```

Datensätze abrufen

Wenn die Zwischenspeicherung von Aktualisierungen aktiviert ist, sorgen BDE-Datenmengen per Voreinstellung automatisch dafür, daß die benötigten Datensätze von der Datenbank abgerufen werden. Daraus kann sich eine große Anzahl von Abrufaktionen ergeben. Ihre Anwendung hat jedoch die Möglichkeit, durch einen Aufruf der Methode *FetchAll* der Datenmenge alle Datensätze auf einmal abzurufen. *FetchAll* erzeugt im Speicher eine lokale Kopie aller Datensätze. Bei sehr großen Datenmengen oder Datenmengen mit großen BLOB-Feldern sollten Sie auf den Einsatz von *FetchAll* verzichten.

Über die Eigenschaft *PacketRecords* geben Client-Datenmengen die Anzahl der Datensätze an, die zu einem beliebigen Zeitpunkt abgerufen werden sollen. Wenn Sie die Eigenschaft *FetchOnDemand* auf *True* setzen, kümmert sich die Client-Datenmenge automatisch um das Abrufen der Datensätze. Andernfalls können Sie Datensätze mit der Methode *GetNextPacket* vom Server abrufen. Weitere Informationen, wie Sie mit Client-Datenmengen Datensätze abrufen, finden Sie unter »Daten von einem Anwendungsserver anfordern« auf Seite 24-19.

Zwischengespeicherte Aktualisierungen zurückschreiben

Aktualisierungen zwischenzuspeichern bedeutet, daß Änderungen erst dann wirklich in die Datenbank zurückgeschrieben (eingetragen) werden, wenn die Anwendung zu diesem Zweck explizit Methoden aufruft. Üblicherweise geschieht dies als Reaktion auf eine Benutzereingabe wie einen Mausklick oder die Auswahl eines Menübefehls.

Wichtig Um Änderungen an nicht aktualisierbaren Ergebnismengen zurückzuschreiben, wie sie aus manchen SQL-Abfragen resultieren, müssen Sie mit Hilfe eines *TUpdateSQL*-Objekts festlegen, wie die Aktualisierung vor sich gehen soll. Bei Joins (Abfragen über zwei oder mehr Tabellen) müssen Sie für jede beteiligte Tabelle ein *TUpdateSQL*-Objekt bereitstellen, das in seiner Ereignisbehandlungsroutine für *OnUpdateRecord* die Änderungen durchführt. Weitere Informationen über *TUpdateSQL*-Objekte finden Sie unter »Schreibgeschützte Ergebnismengen aktualisieren« auf Seite 25-23. Weitere Informationen über Ereignisbehandlungsroutinen für *OnUpdateRecord* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25.

Das Eintragen von Aktualisierungen läuft in zwei Phasen ab. Dabei sollte die Datenbank-Komponente die Transaktion so steuern, daß die Anwendung nicht in einen undefinierten Fehlerzustand gerät. Weitere Informationen über Transaktionen und

Datenbank-Komponenten finden Sie unter »Interaktionen zwischen Datenbank- und Sitzungskomponenten« auf Seite 17-10.

Beim Eintragen von Änderungen im Rahmen einer Transaktion geschieht folgendes:

1 Eine Datenbanktransaktion wird gestartet.

2 Zwischengespeicherte Aktualisierungen werden in die Datenbank geschrieben (Phase 1). Wenn Sie dem Ereignis *OnUpdateRecord* Quelltext zugewiesen haben, wird es für jeden zurückgeschriebenen Datensatz ausgelöst. Wenn dabei ein Fehler auftritt, wird zusätzlich das Ereignis *OnUpdateError* ausgelöst (vorausgesetzt, Sie haben ihm Quelltext zugewiesen).

   Wenn dieser Vorgang fehlschlägt, finden folgende Aktionen statt:

   - Die Änderungen werden zurückgenommen, und die Datenbanktransaktion wird beendet.
   - Die Aktualisierungen werden nicht eingetragen, bleiben aber im Zwischenspeicher erhalten.

   Wird der Schreibvorgang erfolgreich abgeschlossen, finden folgende Aktionen statt:

   - Datenbankänderungen werden eingetragen, und die Datenbanktransaktion wird beendet.
   - Zwischengespeicherte Aktualisierungen werden eingetragen, und der interne Zwischenspeicher wird gelöscht (Phase 2).

Diese zweiphasige Vorgehensweise ermöglicht das Abfangen von Fehlern, besonders wenn mehrere Datenmengen aktualisiert werden (beispielsweise die Datenmengen, die einem Haupt/Detailformular zugeordnet sind). Weitere Informationen hierzu finden Sie unter »Fehlerbehandlung« auf Seite 25-26.

Eigentlich gibt es zwei Möglichkeiten, Aktualisierungen zurückzuschreiben: Zurückschreiben der Aktualisierungen für eine Gruppe von Datenmengen und Zurückschreiben der Aktualisierungen für eine einzelne Datenmenge. Wenn Sie es mit mehreren Datenmengen zu tun haben, rufen Sie die Methode *ApplyUpdates* der Datenbank-Komponente auf, mit der sie verknüpft sind. Bei einzelnen Datenmengen verwenden Sie deren Methoden *ApplyUpdates* und *Commit*. Die Unterschiede zwischen den beiden Verfahren werden in den folgenden Abschnitten erläutert.

Aktualisierungen mit Datenbank-Komponenten eintragen

Die Zwischenspeicherung von Aktualisierungen erfolgt normalerweise auf Datenmengenebene. Es gibt aber Konstellationen, in denen die Aktualisierungen in einer einzigen Transaktion in mehrere, miteinander verknüpfte Datenmengen zurückgeschrieben werden müssen. Bei Verwendung von Haupt/Detailformularen ist es beispielsweise sinnvoll, die Änderungen gleichzeitig in die Haupt- und in die Detailtabellen einzutragen.

Zum Eintragen von zwischengespeicherten Aktualisierungen im Kontext einer Datenbankverbindung wird die Methode *ApplyUpdates* der Datenbank-Komponente

aufgerufen. Im folgenden Beispiel werden als Reaktion auf einen Mausklick alle Aktualisierungen der Datenmenge *CustomersQuery* wirksam:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
   // Für lokale Datenbanken wie Paradox, dBASE und FoxPro
   // wird TransIsolation auf DirtyRead gesetzt.
   if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
   Database1.ApplyUpdates([CustomersQuery]);
end;
```

Dieses Programmfragment startet eine Transaktion und schreibt zwischengespeicherte Aktualisierungen in die Datenbank. Im Erfolgsfall findet erst ein *Commit* für die Transaktion und dann für die zwischengespeicherten Aktualisierungen statt. Im Fehlerfall wird die Transaktion rückgängig gemacht (*Rollback*), wobei aber der Status der zwischengespeicherten Aktualisierungen unverändert bleibt. Die Anwendung sollte auf solche Fehler mit Hilfe einer Behandlungsroutine für das Ereignis *OnUpdateError* reagieren. Näheres hierzu finden Sie unter »Fehlerbehandlung« auf Seite 25-26.

Der große Vorteil der Methode *ApplyUpdates* einer Datenbank-Komponente liegt darin, daß sie eine beliebige Anzahl von Datenmengenkomponenten aktualisieren kann, die mit der Datenbank verknüpft sind. Für eine Datenbank wird der Methode *ApplyUpdates* ein Array mit *TDBDataSet*-Objekten als Parameter übergeben. Mit der folgenden Anweisung werden zum Beispiel zwei in einem Haupt/Detailformular verwendete Tabellen aktualisiert:

```
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
   Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

Weitere Informationen über das Aktualisieren von Haupt/Detailtabellen finden Sie unter »Haupt/Detailtabellen aktualisieren« auf Seite 25-7.

Zurückschreiben bei Datenmengenkomponenten

Zum Eintragen zwischengespeicherter Aktualisierungen in eine Datenmenge muß die Anwendung zwei Methoden der Datenmenge aufrufen:

1. *ApplyUpdates* zum Schreiben von zwischengespeicherten Aktualisierungen in eine Datenbank (Phase 1).
2. *CommitUpdates* zum Löschen des internen Zwischenspeichers nach erfolgreichem Schreibvorgang (Phase 2).

Beim Zurückschreiben auf Datenmengenebene können Sie die Reihenfolge der Aktualisierungen steuern. Dies ist besonders bei Haupt/Detailtabellen wichtig. Weitere Informationen dazu finden Sie unter »Haupt/Detailtabellen aktualisieren« auf Seite 25-7.

Das folgende Beispiel demonstriert, wie innerhalb einer Transaktion Aktualisierungen für die Datenmenge *CustomerQuery* mit einer Datenbankmethode zurückgeschrieben werden:

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
```

```
begin
   Database1.StartTransaction;
   try
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates; { Versuch, die Aktualisierungen in Datenbank zu schreiben }
    Database1.Commit; { Im Erfolgsfall Änderungen eintragen }
   except
    Database1.Rollback; { Andernfalls alle Änderungen rückgängig machen }
    raise; { Exception erneut auslösen, um einen Aufruf von CommitUpdates zu verhindern }
   end;
   CustomerQuery.CommitUpdates; { Im Erfolgsfall internen Zwischenspeicher leeren }
end;
```

Wenn der Aufruf von *ApplyUpdates* eine Exception auslöst, wird für die Transaktion ein *Rollback* durchgeführt. Dadurch ist sichergestellt, daß die zugrundeliegende Datenbanktabelle nicht geändert wird. Die **raise**-Anweisung im **try**...**except**-Block löst die Exception erneut aus und verhindert den Aufruf von *CommitUpdates*, wodurch der interne Zwischenspeicher erhalten bleibt. So können Sie den Fehler vielleicht korrigieren und danach einen neuen Versuch unternehmen.

Haupt/Detailtabellen aktualisieren

Beim Zurückschreiben von Aktualisierungen in Haupt/Detailtabellen spielt die Reihenfolge eine große Rolle. Im allgemeinen sollten Haupttabellen immer vor Detailtabellen aktualisiert werden, außer wenn gelöschte Datensätze im Spiel sind. Diese Regel gilt auch für komplexe Haupt/Detail-Beziehungen, in denen eine Detailtabelle ihrerseits die Haupttabelle einer weiteren Detailtabelle ist.

Haupt/Detailtabellen können auf Datenbank- oder Datenmengenebene aktualisiert werden. Die besseren Kontrollmöglichkeiten haben Sie auf Datenmengenebene. Das folgende Beispiel zeigt die Aktualisierung für die zwei Tabellen *Master* und *Detail* einer Haupt/Detail-Beziehung:

```
Database1.StartTransaction;
try
   Master.ApplyUpdates;
   Detail.ApplyUpdates;
   Database1.Commit;
except
   Database1.Rollback;
   raise;
end;
Master.CommitUpdates;
Detail.CommitUpdates;
```

Wenn ein Fehler auftritt, bleiben in diesem Beispiel sowohl der Zwischenspeicher als auch die zugrundeliegenden Daten in der Datenbank unverändert.

Wenn der Aufruf von *Master.ApplyUpdates* eine Exception auslöst, wird sie behandelt, wie im vorangegangenen Beispiel mit der einzelnen Datenmenge gezeigt. Nehmen wir jedoch an, daß nicht *Master.ApplyUpdates*, sondern der folgende Aufruf von *Detail.ApplyUpdates* fehlschlägt. In diesem Fall sind die Änderungen bereits in der Haupttabelle eingetragen. Da aber die Aktualisierung aller Daten in einer Daten-

banktransaktion vor sich geht, nimmt der Aufruf von *Database1.Rollback* im **except**-Block sogar die Änderungen an der Haupttabelle zurück. Auch *UpdatesMaster.CommitUpdates* wird nicht aufgerufen, da der entsprechende Quelltext nach dem erneuten Auslösen der Exception übersprungen wird. Der Zwischenspeicher bleibt also in dem Zustand, den er vor der versuchten Aktualisierung hatte.

Der wahre Wert des Zwei-Phasen-Modells zeigt sich, wenn Sie sich einen Moment lang vorstellen, *ApplyUpdates* würde die Daten *und den Zwischenspeicher* in einem Schritt aktualisieren. Wenn in diesem Fall beim Eintragen der Daten in die Detailtabelle ein Fehler auftreten würde, gäbe es keinerlei Möglichkeit, die Daten und den Zwischenspeicher in ihren Ausgangszustand zurückzuversetzen. *Database1.Rollback* könnte zwar die Datenbank wiederherstellen, nicht aber den Zwischenspeicher.

Anstehende Aktualisierungen verwerfen

Anstehende Aktualisierungen sind geänderte Datensätze im Zwischenspeicher, die noch nicht in die Datenbank zurückgeschrieben wurden. Es gibt drei Möglichkeiten, solche Aktualisierungen zu verwerfen:

- Um alle anstehenden Aktualisierungen zu verwerfen und die Zwischenspeicherung von nun an zu deaktivieren, setzen Sie die Eigenschaft *CachedUpdates* auf *False*.
- Um alle anstehenden Aktualisierungen zu verwerfen, ohne die Zwischenspeicherung zu deaktivieren, rufen Sie die Methode *CancelUpdates* auf.
- Um Änderungen am aktuellen Datensatz zu verwerfen, rufen Sie *RevertRecord* auf.

In den folgenden Abschnitten werden diese Optionen detailliert beschrieben.

Anstehende Aktualisierungen verwerfen und die Zwischenspeicherung deaktivieren

Um die weitere Zwischenspeicherung zu deaktivieren und alle anstehenden Aktualisierungen zu verwerfen, setzen Sie die Eigenschaft *CachedUpdates* auf *False*. Dadurch wird automatisch die Methode *CancelUpdates* aufgerufen.

Alle noch im Zwischenspeicher vorhandenen gelöschten Datensätze werden wiederhergestellt, für geänderte Datensätze gelten wieder die ursprünglichen Werte, und neu eingefügte Datensätze werden verworfen.

Hinweis Diese Option ist bei Client-Datenmengen nicht verfügbar.

Nur anstehende Aktualisierungen verwerfen

CancelUpdates löscht alle anstehenden Aktualisierungen aus dem Zwischenspeicher und stellt den Zustand der Datenmenge her, der beim Öffnen der Tabelle, beim Aktivieren der Zwischenspeicherung oder beim letzten erfolgreichen Eintragen von Aktualisierungen vorlag. Mit der folgenden Anweisung werden die Aktualisierungen für *CustomersTable* verworfen:

```
CustomersTable.CancelUpdates;
```

Alle im Zwischenspeicher enthalten gelöschten Datensätze werden wiederhergestellt, für geänderte Datensätze gelten wieder die ursprünglichen Werte und neu eingefügte Datensätze werden verworfen.

Hinweis Der Aufruf von *CancelUpdates* deaktiviert nicht die Zwischenspeicherung, sondern verwirft lediglich die anstehenden Aktualisierungen. Um die Zwischenspeicherung zu deaktivieren, setzen Sie *CachedUpdates* auf *False*.

Aktualisierungen am aktuellen Datensatz verwerfen

RevertRecord versetzt den aktuellen Datensatz wieder in den Zustand, den er beim Öffnen der Tabelle, beim Aktivieren der Zwischenspeicherung oder beim letzten erfolgreichen Eintragen von Aktualisierungen hatte. Die Methode kommt meist in einer Ereignisbehandlungsroutine für *OnUpdateError* zum Einsatz, um Fehlersituationen zu korrigieren. Ein Beispiel:

```
CustomersTable.RevertRecord;
```

Das Verwerfen zwischengespeicherter Änderungen an einem Datensatz hat keine Auswirkungen auf die anderen Datensätze. Wenn *RevertRecord* aufgerufen wird, während sich nur ein Datensatz im Zwischenspeicher befindet, ändert sich die Eigenschaft *UpdatesPending* für die Datenmengenkomponenten automatisch von *True* in *False*.

Wenn der Datensatz nicht geändert wurde, hat der Aufruf keine Wirkung. Weitere Informationen über Ereignisbehandlungsroutinen für *OnUpdateError* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25.

Gelöschte Datensätze wiederherstellen

Das Wiederherstellen von Datensätzen erfordert einigen Programmieraufwand, da ein gelöschter Datensatz nicht mehr der aktuelle Datensatz ist und auch nicht mehr in der Datenmenge angezeigt wird. Der gelöschte Datensatz wird bei diesem Vorgang mit Hilfe der Eigenschaft *UpdateRecordTypes* »sichtbar« gemacht. Anschließend wird die Methode *RevertRecord* aufgerufen. Im folgenden Programmbeispiel werden alle gelöschten Datensätze einer Tabelle wiederhergestellt:

```
procedure TForm1.UndeleteAll(DataSet: TBDEDataSet)
begin
   DataSet.UpdateRecordTypes := [rtDeleted];   { Nur die gelöschten Datensätze anzeigen }
   try
    DataSet.First; { Auf den ersten gelöschten Datensatz positionieren }
    while not (DataSet.Eof)
      DataSet.RevertRecord; { Wiederherstellen, bis letzter Datensatz erreicht ist ]
   except { Bei der Wiederherstellung nur geänderte, eingefügte und unveränderte Datensätze
            berücksichtigen }
    DataSet.UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
    raise;
   end;
   DataSet.UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
end;
```

Die Datensatztypen für die Datenmenge festlegen

Wenn die Zwischenspeicherung von Aktualisierungen aktiviert ist, lassen sich mit der Eigenschaft *UpdateRecordTypes* die Datensatztypen festlegen, die in eine Datenmenge aufgenommen werden. Hier bestehen Ähnlichkeiten zu Bereichen und Filtern. Bei *UpdateRecordTypes* handelt es sich um eine Menge, so daß die Eigenschaft jede Kombination der folgenden Werte enthalten kann:

Tabelle 25.1 Werte von TUpdateRecordType

Wert	Bedeutung
rtModified	Geänderte Datensätze
rtInserted	Eingefügte Datensätze
rtDeleted	Gelöschte Datensätze
rtUnmodified	Unveränderte Datensätze

Die Voreinstellung für *UpdateRecordTypes* ist *rtModified*, *rtInserted* und *rtUnmodified*. Gelöschte Datensätze (*rtDeleted*) werden bei dieser Voreinstellung nicht angezeigt.

Die Eigenschaft *UpdateRecordTypes* wird in erster Linie in Ereignisbehandlungsroutinen für *OnUpdateError* zum Zugriff auf gelöschte Datensätze verwendet, um diese durch einen Aufruf von *RevertRecord* wiederherzustellen. Außerdem haben Sie damit die Möglichkeit, Benutzern nur eine Teilmenge der zwischengespeicherten Datensätze anzuzeigen, z.B. alle neu eingefügten (*rtInserted*) Datensätze.

Angenommen, eine Anwendung enthält eine Gruppe mit vier Optionsfeldern (*RadioButton1* bis *RadioButton4*) mit den Beschriftungen *Alle*, *Geändert*, *Eingefügt* und *Gelöscht*. Nachfolgend sehen Sie eine *OnClick*-Ereignisbehandlungsroutine für diese Optionsfelder, die durch entsprechendes Setzen der Eigenschaft *UpdateRecordTypes* dafür sorgt, daß abhängig von einer Bedingung entweder alle Datensätze (außer den gelöschten, da dies die Voreinstellung ist), nur die geänderten Datensätze, nur die neu eingefügten Datensätze oder nur die gelöschten Datensätze angezeigt werden.

```
procedure TForm1.UpdateFilterRadioButtonsClick(Sender: TObject);
begin
   if RadioButton1.Checked then
    CustomerQuery.UpdateRecordTypes := [rtUnmodified, rtModified, rtInserted]
   else if RadioButton2.Checked then
    CustomerQuery.UpdateRecordTypes := [rtModified]
   else if RadioButton3.Checked then
    CustomerQuery.UpdateRecordTypes := [rtInserted]
   else
    CustomerQuery.UpdateRecordTypes := [rtDeleted];
end;
```

Weitere Einzelheiten zum Schreiben einer Behandlungsroutine für das Ereignis *OnUpdateError* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25.

Aktualisierungsstatus prüfen

Ist die Zwischenspeicherung von Aktualisierungen aktiviert, kann der Status eines jeden Datensatzes, für den eine Änderung ansteht, mit Hilfe der Eigenschaft *UpdateStatus* beobachtet werden (meist in Ereignisbehandlungsroutinen für *OnUpdateRecord* und *OnUpdateError*). Weitere Informationen über *OnUpdateRecord* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25. Weitere Informationen über *OnUpdateError* finden Sie unter »Fehlerbehandlung« auf Seite 25-26.

Wenn Sie durch die Datensätze iterieren, zeigt *UpdateStatus* immer den Status des aktuellen Datensatzes an. Die Eigenschaft kann einen der folgenden Werte annehmen:

Tabelle 25.2 Rückgabewerte für UpdateStatus

Wert	Bedeutung
usUnmodified	Der Datensatz wurde nicht geändert.
usModified	Der Datensatz wurde geändert.
usInserted	Der Datensatz ist neu.
usDeleted	Der Datensatz wurde gelöscht.

Beim ersten Öffnen einer Datenmenge hat *UpdateStatus* für alle Datensätze den Wert *usUnmodified*. Sobald Datensätze eingefügt, gelöscht usw. werden, geben die Statuswerte Auskunft über die Änderung. Das folgende Beispiel zeigt die Verwendung der Eigenschaft *UpdateStatus* in einer Behandlungsroutine für das Ereignis *OnScroll* einer Datenmenge. Die Routine zeigt den Aktualisierungsstatus der einzelnen Datensätze in der Statusleiste an.

```
procedure TForm1.CustomerQueryAfterScroll(DataSet: TDataSet);
begin
   with CustomerQuery do begin
    case UpdateStatus of
      usUnmodified: StatusBar1.Panels[0].Text := 'Nicht geändert';
      usModified: StatusBar1.Panels[0].Text := 'Geändert';
      usInserted: StatusBar1.Panels[0].Text := 'Eingefügt';
      usDeleted: StatusBar1.Panels[0].Text := 'Gelöscht';
      else StatusBar1.Panels[0].Text := 'Status nicht definiert';
    end;
   end;
end;
```

Hinweis Hat *UpdateStatus* bei einem Datensatz den Wert *usModified*, kann durch Auswertung der Eigenschaft *OldValue* der einzelnen Felder der ursprüngliche Wert des jeweiligen Feldes ermittelt werden. Die Eigenschaft *OldValue* hat bei Datensätzen, bei denen *UpdateStatus* einen anderen Wert als *usModified* hat, keine Bedeutung. Weitere Informationen über *OldValue* finden Sie unter »Die Feldeigenschaften OldValue, NewValue und CurValue« auf Seite 25-29.

Update-Objekte

TUpdateSQL ist eine Update-Komponente, die eine Datenmenge mit Hilfe von SQL-Anweisungen aktualisiert. Sie benötigen für jede an der Abfrage beteiligte Tabelle eine eigene *TUpdateSQL*-Komponente.

Hinweis Wenn Sie für eine Aktualisierungsoperation mehr als eine Update-Komponente verwenden, müssen Sie ein *OnUpdateRecord*-Ereignis erzeugen, um die einzelnen Update-Komponenten auszuführen.

Eine Update-Komponente kapselt eigentlich drei *TQuery*-Komponenten, von denen jede eine andere Rolle bei der Aktualisierung übernimmt. Eine Abfragekomponente stellt die SQL-Anweisung UPDATE zum Ändern von Datensätzen bereit, eine zweite Komponente liefert eine INSERT-Anweisung zum Hinzufügen von Datensätzen, und eine dritte stellt eine DELETE-Anweisung zur Verfügung, mit der Datensätze aus einer Tabelle entfernt werden können.

Wenn eine Update-Komponente in einem Datenmodul plaziert wird, werden die enthaltenen Abfragekomponenten nicht angezeigt. Die Update-Komponente erstellt sie erst zur Laufzeit auf der Basis dreier Eigenschaften, für die Sie SQL-Anweisungen bereitstellen:

- *ModifiySQL* legt die UPDATE-Anweisung fest.
- *InsertSQL* legt die INSERT-Anweisung fest.
- *DeleteSQL* legt die DELETE-Anweisung fest.

Wenn die Update-Komponente zur Laufzeit aufgerufen wird, führt sie folgende Aktionen durch:

1. Sie wählt auf der Grundlage des Parameters *UpdateKind* die ausführende SQL-Anweisung aus. Der Parameter wird nach jedem Datensatzaktualisierungsereignis automatisch generiert und gibt an, ob der aktuelle Datensatz geändert, eingefügt oder gelöscht wurde.
2. Sie stellt Parameterwerte für die SQL-Anweisung bereit.
3. Sie führt die SQL-Anweisung aus, die die festgelegte Aktualisierung durchführt.

Die Eigenschaft UpdateObject einer Datenmenge

Mit einer Datenmenge, die aktualisiert werden soll, können ein oder mehrere Update-Objekte verknüpft werden. Die entsprechende Zuordnung erfolgt über die Eigenschaft *UpdateObject* der Datenmengenkomponente oder über die Eigenschaft *DataSet* des Update-Objekts. Bei der ersten Möglichkeit weisen Sie der Eigenschaft *UpdateObject* der Datenmengenkomponente das Update-Objekt zu, bei der zweiten legen Sie in der Eigenschaft *DataSet* des Update-Objekts die Datenmenge fest, die aktualisiert werden soll. Welche Vorgehensweise verwendet werden muß, hängt davon ab, ob an der zu aktualisierenden Datenmenge eine oder mehrere Basistabellen beteiligt sind.

Für die Zuordnung einer zu aktualisierenden Datenmenge zu einem Update-Objekt muß eine dieser Methoden verwendet werden. Ohne diese Zuordnung ist es nicht möglich, die Parameter für die SQL-Anweisungen des Update-Objekts mit den korrekten Werten zu füllen. Entscheiden sie sich für eine Zuordnungsmethode. Eine Kombination beider Vorgehensweisen ist nicht zulässig.

Die Methode, die für die Zuordnung eines Update-Objekts an eine Datenmenge verwendet wird, wirkt sich auf die Ausführung des Update-Objekts aus. Ein Update-Objekt kann entweder automatisch (ohne Intervention der Anwendung) oder erst nach einer expliziten Anweisung durch die Anwendung ausgeführt werden. Die Ausführung erfolgt automatisch, wenn die Zuordnung mit der Eigenschaft *UpdateObject* der Datenmengenkomponente vorgenommen wurde. Wenn die Zuordnung dagegen mit Hilfe der Eigenschaft *DataSet* des Update-Objekts erfolgt ist, muß zur Ausführung des Objekts entsprechender Quelltext geschrieben werden.

In den folgenden Abschnitten wird detailliert auf die Zuordnung von Update-Objekten eingegangen. Sie finden dort auch Hinweise zur Zuordnungsmethode und deren Auswirkung auf die Aktualisierung.

Ein einzelnes Update-Objekt verwenden

Wenn nur eine der in der Datenmenge referenzierten Basistabellen aktualisiert werden muß, ordnen Sie der Datenmenge ein Objekt zu, indem Sie in der Eigenschaft *UpdateObject* der Datenmengenkomponente den Namen des Update-Objekts angeben. Ein Beispiel:

```
Query1.UpdateObject := UpdateSQL1;
```

Die im Update-Objekt enthaltenen SQL-Anweisungen werden automatisch ausgeführt, sobald die Methode *ApplyUpdates* der zu aktualisierenden Datenmenge aufgerufen wird. Das Update-Objekt wird für jeden Datensatz aufgerufen, der aktualisiert werden muß. Erstellen Sie keine Behandlungsroutine für das Ereignis *OnUpdateRecord*, in der die Methode *ExecSQL* des Update-Objekts explizit aufgerufen wird. Dies hätte zur Folge, daß die Aktualisierungen der einzelnen Datensätze ein zweites Mal eingetragen werden.

Wenn Sie eine Behandlungsroutine für das Ereignis *OnUpdateRecord* der Datenmenge bereitstellen, muß darin zumindest der *UpdateAction*-Parameter der Routine auf *uaApplied* gesetzt werden. Optional können Sie in der Behandlungsroutine Datenvalidierungen und Änderungen durchführen oder weitere Operationen definieren (z.B. Parameterwerte festlegen).

Mehrere Update-Objekte verwenden

Wenn mehrere der in der Datenmenge referenzierten Basistabellen aktualisiert werden müssen, ist die Verwendung einer entsprechenden Anzahl von Update-Objekten erforderlich. Da über die Eigenschaft *UpdateObject* der Datenmengenkomponente jeweils nur ein Update-Objekt mit der Datenmenge verknüpft werden kann, ist eine Zuordnung über die Eigenschaft *DataSet* erforderlich. Dabei wird in der Eigenschaft *DataSet* jedes Update-Objekts der Name der Datenmenge angegeben. Während des Entwurfs ist die *DataSet*-Eigenschaft für Update-Objekte nicht im Objektinspektor verfügbar. Sie kann deshalb nur zur Laufzeit festgelegt werden:

```
UpdateSQL1.DataSet := Query1;
```

Die im Update-Objekt enthaltenen SQL-Anweisungen werden *nicht* automatisch ausgeführt, wenn die Methode *ApplyUpdates* der zu aktualisierenden Datenmenge aufgerufen wird. Sie müssen deshalb eine Behandlungsroutine für das Ereignis *OnUpdateRecord* der Datenmengenkomponente schreiben, in der die Methode *ExecSQL* oder *Apply* des Update-Objekts explizit aufgerufen wird. Das Update-Objekt wird dadurch für jeden Datensatz aufgerufen, der aktualisiert werden muß.

In einer Behandlungsroutine für das Ereignis *OnUpdateRecord* der Datenmenge müssen mindestens die folgenden Aktionen durchgeführt werden:

- Ein Aufruf der Methode *SetParams* des Update-Objekts (wenn später ein Aufruf von *ExecSQL* folgt).
- Die Ausführung des Update-Objekts für den aktuellen Datensatz mit der Methode *ExecSQL* oder *Apply*.
- Das Setzen des *UpdateAction*-Parameters der Ereignisbehandlungsroutine auf *uaApplied*.

Optional können Sie in der Behandlungsroutine Datenvalidierungen und Änderungen durchführen oder weitere Operationen für die einzelnen Datensätze definieren.

Hinweis Es ist möglich, ein Update-Objekt über die *UpdateObject*-Eigenschaft der Datenmenge mit der Datenmenge zu verknüpfen, die Zuordnung des zweiten und aller weiteren Update-Objekte aber über die *DataSet*-Eigenschaft vorzunehmen. Das erste Update-Objekt wird beim Aufruf der Methode *ApplyUpdates* der Datenmengenkomponenten automatisch ausgeführt, während die restlichen Update-Objekte manuell ausgeführt werden müssen.

SQL-Anweisungen für Update-Komponenten erstellen

Ein Update-Objekt führt eine von drei möglichen SQL-Anweisungen aus, um einen Datensatz in einer Datenmenge zu aktualisieren. Mit diesen Anweisungen werden die zu aktualisierenden Datensätze in der zugrundeliegenden Tabelle gelöscht, eingefügt oder geändert. Die Anweisungen sind in den Stringlisten-Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* des Update-Objekts enthalten. Da ein Update-Objekt jeweils nur eine bestimmte Tabelle aktualisiert, beziehen sich alle Aktualisierungsanweisungen des Objekts auf dieselbe Basistabelle.

Beim Zurückschreiben der Aktualisierungen für die einzelnen Datensätze wird eine SQL-Anweisung für die entsprechende Basistabelle ausgeführt. Welche von den drei möglichen SQL-Anweisungen verwendet wird, hängt vom Parameter *UpdateKind* ab, der nach jedem Aktualisierungsereignis automatisch generiert wird.

Die SQL-Anweisungen für Update-Objekte können während des Entwurfs oder zur Laufzeit erstellt werden. In den folgenden Abschnitten wird die Erstellung dieser SQL-Anweisungen detailliert beschrieben:

SQL-Anweisungen zur Entwurfszeit erstellen

Zur Erstellung einer SQL-Anweisung für eine Update-Komponente müssen folgende Schritte ausgeführt werden:

1. Wählen Sie die *TUpdateSQL*-Komponente aus.
2. Wählen Sie im Objektinspektor den Namen der Update-Komponente in der Dropdown-Liste der Eigenschaft *UpdateObject* der Datenmengenkomponente. Dieser Schritt stellt sicher, daß der UpdateSQL-Editor, der im nächsten Schritt aktiviert wird, passende Standardwerte festlegen kann, die bei der SQL-Generierung verwendet werden.
3. Klicken Sie mit der rechten Maustaste auf die Update-Komponente, und wählen Sie *UpdateSQL-Editor* im lokalen Menü, um den UpdateSQL-Editor zu aktivieren. Der Editor generiert, basierend auf der zugrundeliegenden Datenmenge und auf den von Ihnen bereitgestellten Werten, die SQL-Anweisungen für die Eigenschaften *ModifySQL*, *InsertSQL* und *DeleteSQL* der Update-Komponente.

Der UpdateSQL-Editor besteht aus zwei Seiten. Die Seite *Optionen* wird angezeigt, wenn der Editor erstmals aufgerufen wird. Wählen Sie mit Hilfe des Kombinationsfeldes *Tabellenname* die Tabelle aus, die aktualisiert werden soll. Sobald Sie einen Tabellennamen angegeben haben, enthalten die Listenfelder *Schlüsselfelder* und *Felder aktualisieren* die verfügbaren Spalten.

Im Listenfeld *Felder aktualisieren* werden die Spalten festgelegt, die aktualisiert werden sollen. Wenn eine Tabelle zum ersten Mal angegeben wird, sind alle Spalten in der Auswahl des Listenfeldes enthalten. Es können mehrere Felder ausgewählt werden.

Mit Hilfe des Listenfeldes *Schlüsselfelder* werden die Spalten angegeben, die bei der Aktualisierung als Schlüssel verwendet werden sollen. Die hier angegebenen Spalten müssen einem existierenden Index entsprechen, insbesondere bei Paradox-, dBASE- oder FoxPro-Tabellen. Bei Remote-SQL-Datenbanken wird kein Index benötigt. Anstatt die Schlüsselfelder über das gleichnamige Listenfeld festzulegen, können Sie auch auf die Schaltfläche *Primärschlüsselfelder* klicken und die Schlüsselfelder für die Aktualisierung basierend auf dem Primärindex der Tabelle auswählen. Durch Klikken auf *Datenmengen-Standard* werden die Voreinstellungen der Datenmenge in die Listenfelder *Schlüsselfelder* und *Felder aktualisieren* übernommen. Nach dem Klicken auf *Primärschlüsselfelder* werden die Schlüsselfelder basierend auf dem Primärindex einer Tabelle ausgewählt.

Wenn Sie das Kontrollfeld *Feldnamen in Anführungszeichen* aktivieren, werden Feldnamen in Anführungszeichen gesetzt. Diese Option muß bei einigen Servern aus Gründen der Kompatibilität aktiviert werden.

Nachdem eine Tabelle und die Spalten für Schlüssel und Aktualisierung festgelegt wurden, werden durch Klicken auf *SQL generieren* die vorläufigen SQL-Anweisungen für die Eigenschaften *ModifySQL*, *InsertSQL* und *DeleteSQL* generiert. Meist werden Sie die automatisch erzeugten SQL-Anweisungen noch geringfügig anpassen.

Die Seite *SQL* dient zum Anzeigen und Ändern der generierten SQL-Anweisungen. Sind vor dem Auswählen dieser Seite SQL-Anweisungen erstellt worden, wird die

Anweisung für die Eigenschaft *ModifySQL* bereits im Memofeld *SQL Text* angezeigt und kann dort bearbeitet werden.

Wichtig Bitte beachten Sie, daß die generierten SQL-Anweisungen lediglich als Basis gedacht sind. Damit diese Anweisungen richtig ausgeführt werden können, müssen Sie als Programmierer eventuell noch Ergänzungen vornehmen. Wenn Sie beispielsweise mit Daten arbeiten, die Null-Werte enthalten, sollte die WHERE-Klausel nicht die erzeugte Feldvariable verwenden, sondern

```
WHERE field IS NULL
```

lauten. Testen Sie jede einzelne Anweisung, bevor Sie sie akzeptieren.

Mit den Optionsfeldern der Gruppe *Anweisungstyp* können SQL-Anweisungen aus den generierten Anweisungen ausgewählt und anschließend bearbeitet werden.

Durch Klicken auf die Schaltfläche *OK* wird die Anweisung übernommen und der entsprechenden SQL-Eigenschaft der Update-Komponente zugeordnet.

Parameterersetzung in SQL-Anweisungen

SQL-Anweisungen für die Aktualisierung verwenden eine spezielle Form der Parameterersetzung. Diese ermöglicht es, alte oder neue Feldwerte in Datensatzaktualisierungen zu ersetzen. Während der Generierung der Anweisungen mit dem UpdateSQL-Editor legt dieser fest, welche Feldwerte verwendet werden. Wenn Sie eigene SQL-Anweisungen schreiben, können Sie die zu verwendenden Feldwerte selbst bestimmen.

Wenn der Parametername mit einem Spaltennamen in der Tabelle übereinstimmt, wird der neue Wert, der für das Feld in der zwischengespeicherten Aktualisierung des Datensatzes enthalten ist, automatisch als Wert für diesen Parameter verwendet. Stimmt der Parametername mit einem Spaltennamen überein, dem der String »OLD_« vorangestellt ist, wird der alte Wert des Feldes benutzt. In der folgenden SQL-Anweisung wird der Parameter *:LastName* automatisch mit dem neuen Feldwert gefüllt, der im Zwischenspeicher für den eingefügten Datensatz enthalten ist:

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

Bei den Anweisungen *InsertSQL* und *ModifySQL* werden üblicherweise die neuen Feldwerte verwendet. Bei der Aktualisierung eines geänderten Datensatzes wird in der UPDATE-Anweisung der neue Feldwert im Zwischenspeicher verwendet, um den alten Feldwert in der Basistabelle zu ersetzen.

Beim Löschen von Datensätzen gibt es keine neuen Werte, so daß bei der Eigenschaft *DeleteSQL* die Syntax »:OLD_Feldname« Verwendung findet. Üblicherweise werden die alten Feldwerte auch in der WHERE-Klausel der SQL-Anweisung zum Festlegen des zu aktualisierenden Datensatzes verwendet.

In der WHERE-Klausel einer UPDATE- oder DELETE-SQL-Anweisung müssen zumindest so viele Parameter angegeben werden, daß der zu aktualisierende Datensatz in der Basistabelle eindeutig identifiziert werden kann. Beispielsweise würde es in einer Kundenliste nicht ausreichen, nur den Nachnamen des Kunden anzugeben, da es mehrere Kunden mit diesem Namen geben könnte. Um einen Kunden eindeutig zu

identifizieren, könnten der Nachname, der Vorname und die Telefonnummer als Parameter übergeben werden. Besser wäre allerdings die Verwendung eines eindeutigen Feldes, z.B. einer Kundennummer.

Weitere Informationen zu diesem Thema finden Sie unter »Die Feldeigenschaften OldValue, NewValue und CurValue« auf Seite 25-29

SQL-Anweisungen schreiben

Die Komponente *TUpdateSQL* verfügt über drei Eigenschaften, die SQL-Anweisungen für die Aktualisierung aufnehmen können: *DeleteSQL*, *InsertSQL* und *ModifySQL*. Damit ist es möglich, Datensätze in der Basistabelle zu löschen, einzufügen oder zu ändern.

Die Eigenschaft *DeleteSQL* sollte nur SQL-Anweisungen mit dem Befehl DELETE enthalten. Die zu aktualisierende Basistabelle wird in der FROM-Klausel angegeben. Damit die SQL-Anweisung nur den Datensatz in der Basistabelle löscht, der dem gelöschten Datensatz im Aktualisierungszwischenspeicher entspricht, verwenden Sie eine WHERE-Klausel. In dieser Klausel identifizieren Sie den betreffenden Datensatz in der Basistabelle mit Hilfe eines oder mehrerer Parameter. Wenn ein Parameter den Namen eines Feldes mit einem vorangestellten »OLD_« enthält, wird der Wert des entsprechenden Feldes im Aktualisierungszwischenspeicher als Wert für den Parameter benutzt. Wenn Sie dem Parameter einen anderen Namen geben, müssen Sie den Wert selbst zuweisen. Das folgende Beispiel zeigt die Verwendung der Syntax »:OLD_Feldname«:

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Bei einigen Tabellentypen wird der gewünschte Datensatz in der Basistabelle nicht gefunden, wenn die Felder zu Identifizierung des Datensatzes den Wert Null enthalten. Das hat zur Folge, daß die betreffenden Datensätze nicht gelöscht werden. Sie können dieses Problem umgehen, indem Sie für Felder, die einen Null-Wert enthalten könnten, eine IS NULL-Klausel hinzufügen (zusammen mit einer Bedingung für Werte, die nicht Null sind). Das folgende Beispiel zeigt eine solche Klausel für ein Feld namens *FirstName*:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
   ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

Die Eigenschaft *InsertSQL* sollte nur SQL-Anweisungen mit dem Befehl INSERT enthalten. Die zu aktualisierende Basistabelle wird in der INTO-Klausel angegeben. Die VALUES-Klausel muß eine Liste mit Parametern enthalten, die durch Kommas voneinander getrennt sind. Wenn ein Parametername mit einem Feldnamen identisch ist, wird der Wert des entsprechenden Feldes im Aktualisierungszwischenspeicher als Wert für den Parameter benutzt. Wenn Sie dem Parameter einen anderen Namen geben, müssen Sie den Wert selbst zuweisen. Über die Liste der Parameter werden die Werte für die Felder des eingefügten Datensatzes bereitgestellt. Für jedes in der Anweisung enthaltene Feld muß ein Wertparameter angegeben werden:

```
INSERT INTO Inventory
(ItemNo, Amount)
```

```
VALUES (:ItemNo, 0)
```

Die Eigenschaft *ModifySQL* sollte nur SQL-Anweisungen mit dem Befehl UPDATE enthalten. Die zu aktualisierende Basistabelle wird in der FROM-Klausel angegeben. Wertzuweisungen werden mit Hilfe einer SET-Klausel durchgeführt. Wenn in den Zuweisungen der SET-Klausel Parameter angegeben werden, die mit dem Namen von Feldern identisch sind, erhalten die Parameter automatisch die Werte der betreffenden Felder des aktualisierten Datensatzes im Zwischenspeicher. Sie können weitere Feldwerte zuweisen, indem Sie zusätzliche Parameter angeben und Werte dafür bereitstellen. Die Namen dieser zusätzlichen Parameter dürfen dann aber nicht mit Feldnamen identisch sein. Wie bei einer *DeleteSQL*-Anweisung können Sie auch hier eine WHERE-Klausel verwenden, um den Datensatz in der Basistabelle eindeutig zu identifizieren. Dazu verwenden Sie Parameter, die denselben Namen wie die betreffenden Felder haben, fügen aber das Präfix »OLD_« hinzu. In der folgenden Update-Anweisung wird dem Parameter *:ItemNo* automatisch ein Wert zugewiesen, dem Parameter *:Price* jedoch nicht:

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Angenommen, ein Benutzer der Anwendung, in der diese SQL-Anweisung definiert ist, ändert einen vorhandenen Datensatz. Vor der Änderung enthält das Feld *ItemNo* den Wert 999. Der Benutzer ändert nun in einem Gitter, das mit der zwischengespeicherten Datenmenge verknüpft ist, den Wert von *ItemNo* in 123 und den Wert von *Amount* in 20. Wenn die Methode *ApplyUpdates* aufgerufen wird, wirkt sich diese SQL-Anweisung auf alle Datensätze in der Basistabelle aus, deren Wert im Feld *ItemNo* 999 lautet, da dieser über den Parameter *:OLD_ItemNo* festgelegt ist. Nach der Änderung enthalten diese Datensätze dann im Feld *ItemNo* den Wert 123 (verwendet wird der Parameter *:ItemNo*, der Wert stammt aus dem Gitter) und im Feld *Amount* den Wert 20.

Die Eigenschaft Query einer Update-Komponente

Über die Eigenschaft *Query* einer Update-Komponente können Sie auf die SQL-Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* zugreifen und eine SQL-Anweisung definieren oder ändern. *UpdateKind*-Konstanten können als Index für das Array mit Abfragekomponenten eingesetzt werden. Der Zugriff auf die Eigenschaft *Query* ist nur zur Laufzeit möglich.

In der folgenden Anweisung wird die *UpdateKind*-Konstante *ukDelete* mit der Eigenschaft *Query* verwendet, um auf die Eigenschaft *DeleteSQL* zuzugreifen:

```
with UpdateSQL1.Query[ukDelete] do begin
   Clear;
   Add('DELETE FROM Inventory I');
   Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

Üblicherweise werden die Eigenschaften, auf welche die Eigenschaft *SQL* verweist, zur Entwurfszeit mit Hilfe des UpdateSQL-Editors eingestellt. Eventuell muß jedoch auf diese Werte zur Laufzeit zugegriffen werden, wenn Sie eine eindeutige *UpdateSQL*-Anweisung für jeden Datensatz generieren und auf eine Parameterbindung ver-

zichten. Im folgenden Beispiel werden für jeden aktualisierten Datensatz eindeutige Werte für die Eigenschaft *SQL* generiert:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   with UpdateSQL1 do begin
    case UpdateKind of
      ukModified:
        begin
          Query[UpdateKind].Text := Format('update emptab set Salary = %d where EmpNo = %d',
            [EmpAuditSalary.NewValue, EmpAuditEmpNo.OldValue]);
          ExecSQL(UpdateKind);
        end;
      ukInserted:
        {...}
      ukDeleted:
        {...}
    end;
   end;
   UpdateAction := uaApplied;
end;
```

Hinweis *Query* gibt einen Wert des Typs *TDataSetUpdateObject* zurück. Sie können für die Eigenschaft *UpdateObject* eine Typumwandlung durchführen, damit dieser Rückgabewert als *TUpdateSQL*-Komponente verwendet und auf *TUpdateSQL*-spezifische Eigenschaften und Methoden zugegriffen werden kann:

```
with (DataSet.UpdateObject as TUpdateSQL).Query[UpdateKind] do begin
   { Operationen für die Anweisung in DeleteSQL durchführen. }
end;
```

Ein Beispiel für diese Eigenschaft finden Sie unter »Die Methode SetParams« auf Seite 25-21.

Die Eigenschaften DeleteSQL, InsertSQL und ModifySQL

Die Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* werden verwendet, um die entsprechenden SQL-Anweisungen zu definieren. Es handelt sich bei diesen Eigenschaften um Stringlisten-Container. Mit den Methoden einer Stringliste können Sie diesen Eigenschaften SQL-Anweisungszeilen zuweisen. Um eine bestimmte Zeile in der Eigenschaft zu referenzieren, verwenden Sie eine ganze Zahl als Index. Der Zugriff auf die Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* ist sowohl zur Entwurfs- als auch zur Laufzeit möglich.

```
with UpdateSQL1.DeleteSQL do begin
   Clear;
   Add('DELETE FROM Inventory I');
   Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

In der folgenden Anweisung wird der Index 2 für die Eigenschaft *ModifySQL* verwendet, um die dritte Zeile einer SQL-Anweisung zu ändern:

```
UpdateSQL1.ModifySQL[2] := 'WHERE ItemNo = :ItemNo';
```

Update-Anweisungen ausführen

An der Ausführung einer SQL-Anweisung für eine Datensatzaktualisierung können mehrere Methoden beteiligt sein. Der Aufruf dieser Methoden erfolgt normalerweise in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* des Update-Objekts, wenn die Werte des zwischengespeicherten Datensatzes zurückgeschrieben werden. Welche Methode verwendet wird, hängt von der jeweiligen Situation ab. In den folgenden Abschnitten werden diese Methoden detailliert beschrieben.

Die Methode Apply

Mit der Methode *Apply* einer Update-Komponente können Sie Aktualisierungen manuell in den aktuellen Datensatz eintragen. Bei diesem Vorgang werden folgende Schritte ausgeführt:

1. Die Werte für den Datensatz werden an die Parameter der entsprechenden SQL-Anweisung gebunden.
2. Die SQL-Anweisung wird ausgeführt.

Durch einen Aufruf der Methode *Apply* wird die im Zwischenspeicher befindliche Aktualisierung für den aktuellen Datensatz in die zugrundeliegende Tabelle geschrieben. Verwenden Sie *Apply* nur, wenn das Update-Objekt nicht über die *UpdateObject*-Eigenschaft der Datenmengenkomponente mit der Datenmenge verknüpft ist (in diesem Fall erfolgt keine automatische Ausführung des Update-Objekts). *Apply* ruft automatisch die Methode *SetParams* auf, um den Parametern der SQL-Anweisung die alten und neuen Feldwerte zuzuweisen. Nach einem Aufruf von *Apply* darf deshalb die Methode *SetParams* nicht direkt aufgerufen werden. *Apply* wird normalerweise in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* der Datenmenge aufgerufen.

Wenn die Datenmenge und das Update-Objekt über die *UpdateObject*-Eigenschaft der Datenmengenkomponente verknüpft sind, wird *Apply* automatisch aufgerufen. Verwenden Sie in diesem Fall keine Behandlungsroutine für *OnUpdateRecord*, in der die Methode *Apply* explizit aufgerufen wird. Dies hätte zur Folge, daß die Aktualisierung des aktuellen Datensatzes ein zweites Mal eingetragen wird.

In einer Behandlungsroutine für das Ereignis *OnUpdateRecord* wird mit Hilfe des Parameters *UpdateKind* festgestellt, welche SQL-Anweisung verwendet werden muß. Bei einem Aufruf durch die zugehörige Datenmenge wird *UpdateKind* automatisch gesetzt. Wenn Sie die Methode *Apply* in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* aufrufen, übergeben Sie ihr eine *UpdateKind*-Konstante als Parameter. Ein Beispiel:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   UpdateSQL1.Apply(UpdateKind);
   UpdateAction := uaApplied;
end;
```

Tritt während der Aktualisierung eine Exception auf, wird mit der Behandlungsroutine für das Ereignis *OnUpdateError* fortgefahren, wenn eine solche definiert wurde.

Hinweis Die von *Apply* durchgeführten Operationen entsprechen den in den folgenden Abschnitten beschriebenen Methoden *SetParams* und *ExecSQL*.

Die Methode SetParams

Die Methode *SetParams* einer Update-Komponente ersetzt nach speziellen Regeln die Parameter in der SQL-Anweisung. *SetParams* wird normalerweise von der Methode *Apply* der Update-Komponente automatisch aufgerufen. Wenn *Apply* direkt in einer Ereignisbehandlungsroutine für *OnUpdateRecord* aufgerufen wird, sollte das Programm *SetParams* nicht aufrufen. Wenn Sie ein Update-Objekt mit der zugehörigen *ExecSQL*-Methode ausführen, verwenden Sie die Methode *SetParams*, um den Parametern der SQL-Anweisung Werte zuzuweisen.

Der Parameter *UpdateKind* legt fest, in welcher SQL-Anweisung *SetParams* Parameter ersetzen soll. Eine automatische Wertzuweisung erfolgt nur für Parameter, die bestimmte Namenskonventionen verwenden. Dies trifft auf Parameter zu, deren Name mit dem Namen eines Feldes identisch ist, oder die zusätzlich zu einem Feldnamen das Präfix »OLD_« tragen. Parametern mit anderen Namen müssen die Werte manuell zugewiesen werden. Weitere Informationen finden Sie unter »Parameterersetzung in SQL-Anweisungen« auf Seite 25-16.

Das folgende Programmbeispiel zeigt die Verwendung von *SetParams*:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   with DataSet.UpdateObject as TUpdateSQL do begin
    SetParams(UpdateKind);
    if UpdateKind = ukModified then
      Query[UpdateKind].ParamByName('DateChanged').Value := Now;
    ExecSQL(UpdateKind);
   end;
   UpdateAction := uaApplied;
end;
```

Hinweis Dieses Beispiel geht davon aus, daß die Eigenschaft *ModifySQL* der Update-Komponente folgende Struktur hat:

```
UPDATE EmpAudit
SET EmpNo = :EmpNo, Salary = :Salary, Changed = :DateChanged
WHERE EmpNo = :OLD_EmpNo
```

In diesem Beispiel liefert der Aufruf von *SetParams* die Werte der Parameter *EmpNo* und *Salary*. Dem Parameter *DateChanged* wird kein Wert zugewiesen, da in der Datenmenge kein gleichnamiges Feld existiert. Der Wert wird daher in der nächsten Programmzeile explizit gesetzt.

Die Methode ExecSQL

Mit der Methode *ExecSQL* einer Update-Komponente lassen sich Aktualisierungen manuell in den aktuellen Datensatz eintragen. Dieser Vorgang umfaßt zwei Schritte:

1 Die Werte für den Datensatz werden an die Parameter der entsprechenden SQL-Anweisung gebunden.

2 Die SQL-Anweisung wird ausgeführt.

Die Methode *ExecSQL* schreibt die im Zwischenspeicher befindlichen Aktualisierungen für den aktuellen Datensatz in die zugrundeliegende Tabelle. Verwenden Sie *ExecSQL* nur, wenn das Update-Objekt nicht über die *UpdateObject*-Eigenschaft der Datenmengenkomponente mit der Datenmenge verknüpft ist (in diesem Fall erfolgt keine automatische Ausführung des Update-Objekts). *ExecSQL* ruft *nicht* automatisch die Methode *SetParams* auf, um den Parametern der SQL-Anweisung Werte zuzuweisen. Aus diesem Grund muß vor dem Aufruf von *ExecSQL* explizit die Methode *SetParams* aufgerufen werden. *ExecSQL* wird normalerweise in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* der Datenmenge aufgerufen.

Wenn die Datenmenge und das Update-Objekt über die *UpdateObject*-Eigenschaft der Datenmengenkomponente miteinander verknüpft sind, wird *ExecSQL* automatisch aufgerufen. Verwenden Sie in diesem Fall keine Behandlungsroutine für das Ereignis *OnUpdateRecord*, in der die Methode *ExecSQL* explizit aufgerufen wird. Dies hätte zur Folge, daß die Aktualisierung des aktuellen Datensatzes ein zweites Mal eingetragen wird.

In einer Behandlungsroutine für das Ereignis *OnUpdateRecord* wird mit Hilfe des Parameters *UpdateKind* festgestellt, welche SQL-Anweisung verwendet werden muß. Bei einem Aufruf durch die zugehörige Datenmenge wird *UpdateKind* automatisch gesetzt. Wenn Sie die Methode *ExecSQL* in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* aufrufen, übergeben Sie ihr eine *UpdateKind*-Konstante als Parameter. Ein Beispiel:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   with (DataSet.UpdateObject as TUpdateSQL) do begin
    SetParams(UpdateKind);
    ExecSQL(UpdateKind);
   end;
   UpdateAction := uaApplied;
end;
```

Tritt während der Aktualisierung eine Exception auf, wird mit der Behandlungsroutine für das Ereignis *OnUpdateError* fortgefahren, wenn eine solche definiert wurde.

Hinweis Die von *SetParams* und *ExecSQL* durchgeführten Operationen entsprechen der zuvor beschriebenen Methode *Apply*.

Eine Datenmenge mit Datenmengenkomponenten aktualisieren

Das Zurückschreiben zwischengespeicherter Aktualisierungen bedingt normalerweise die Verwendung eines oder mehrerer Update-Objekte. Die SQL-Anweisungen für diese Objekte sorgen dafür, daß die Basistabelle mit den Datenänderungen aktualisiert wird. Zwar läßt sich die Aktualisierung einer Datenmenge auf diese Weise am einfachsten durchführen, alternativ können zu diesem Zweck aber auch Datenmengenkomponenten wie *TTable* und *TQuery* eingesetzt werden.

Verwenden Sie in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* einer Datenmengenkomponente die Eigenschaften und Methoden einer anderen Daten-

mengenkomponente, um die zwischengespeicherten Aktualisierungen für einen Datensatz zurückzuschreiben.

Im folgenden Programmbeispiel werden Aktualisierungen mit Hilfe einer Tabellenkomponente zurückgeschrieben:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
   else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;
        ukInsert:
          begin
            Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;
        ukModify: DeleteRecord;
      end;
        UpdateAction := uaApplied;
end;
```

Schreibgeschützte Ergebnismengen aktualisieren

Obwohl die BDE versucht, aktualisierbare Ergebnismengen zu liefern, wenn die Eigenschaft *RequestLive* einer Datenmengenkomponente den Wert *True* hat, gibt es Situationen, in denen dies nicht möglich ist. Weitere Informationen hierzu finden Sie unter »Wann werden zwischengespeicherte Aktualisierungen eingesetzt?« auf Seite 25-1.

In diesen Fällen kann eine Datenmenge folgendermaßen manuell aktualisiert werden:

1. Plazieren Sie eine *TUpdateSQL*-Komponente im Datenmodul der Anwendung.
2. Weisen Sie der Eigenschaft *UpdateObject* der Datenmengenkomponente den Namen der Komponente *TUpdateSQL* im Datenmodul zu.
3. Tragen Sie die Update-Anweisung für die Ergebnismenge in die Eigenschaften *ModifySQL*, *InsertSQL* oder *DeleteSQL* der Update-Komponente ein, oder verwenden Sie den UpdateSQL-Editor.
4. Schließen Sie die Datenmenge.

5 Weisen Sie der Eigenschaft *CachedUpdates* der Datenmengenkomponente den Wert *True* zu.

6 Öffnen Sie die Datenmenge erneut.

Hinweis In vielen Situationen ist es sinnvoll, für die Datenmenge auch eine Ereignisbehandlungsroutine für *OnUpdateRecord* bereitzustellen.

Den Aktualisierungsvorgang steuern

Die Methode *ApplyUpdates* einer Datenmengenkomponente versucht, die Aktualisierungen aller im Zwischenspeicher befindlichen Datensätze in die Basistabelle zu schreiben. Unmittelbar bevor ein geänderter, gelöschter oder eingefügter Datensatz zurückgeschrieben wird, löst die Methode das Ereignis *OnUpdateRecord* der Datenmengenkomponente aus.

In einer Behandlungsroutine für dieses Ereignis können Sie Aktionen definieren, die direkt vor dem Zurückschreiben des Datensatzes ausgeführt werden sollen. Sie können z.B. eine spezielle Validierung der Daten vornehmen, andere Tabelle aktualisieren oder mehrere Update-Objekte ausführen. In einer *OnUpdateRecord*-Ereignisbehandlungsroutine können Sie also umfassend auf den Aktualisierungsvorgang einwirken.

In den folgenden Abschnitten werden Situationen beschrieben, in denen eine Behandlungsroutine für das Ereignis *OnUpdateRecord* bereitgestellt werden muß. Außerdem finden Sie hier detaillierte Erläuterungen zur Entwicklung einer solchen Routine.

Wann muß der Aktualisierungsprozeß überwacht werden?

Manchmal brauchen Sie bei zwischengespeicherten Aktualisierungen nur *ApplyUpdates* aufzurufen, um die Änderungen in die Basistabellen der Datenbank zu schreiben (wenn Sie beispielsweise über eine *TTable*-Komponente exklusiven Zugriff auf eine lokale Paradox- oder dBASE-Tabelle haben). Zusätzliche Operationen können in einer Behandlungsroutine für das Ereignis *OnUpdateRecord* der aktualisierten Datenmengenkomponente implementiert werden.

Dem Ereignis *OnUpdateRecord* können Sie beispielsweise Prüfroutinen zuordnen, die Daten vor dem Eintragen in die Datenbank in die entsprechende Form bringen, oder Routinen, die Datensätze in Haupt- und Detailtabellen bearbeiten, bevor sie in die Datenbank geschrieben werden.

In vielen Fällen sind zusätzliche Bearbeitungsschritte erforderlich. Wenn Sie beispielsweise mit einem Join Abfragen über mehrere Tabellen stellen, müssen Sie für jede Tabelle ein *TUpdateSQL*-Objekt bereitstellen und außerdem in einer Ereignisbehandlungsroutine für *OnUpdateRecord* sicherstellen, daß jedes Update-Objekt ausgeführt und alle Änderungen in die Tabellen geschrieben werden.

Die folgenden Abschnitte beschreiben die Komponente *TUpdateSQL* und das Ereignis *OnUpdateRecord*.

Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen

Das Ereignis *OnUpdateRecord* ist hilfreich, wenn eine bestimmte Update-Komponente die erforderlichen Aktualisierungen nicht durchführen kann oder wenn die Anwendung zusätzliche Kontrolle über die Parameterersetzung benötigt. Das Ereignis *OnUpdateRecord* wird bei jedem Versuch ausgelöst, die Änderungen eines geänderten Datensatzes in die Basistabelle zu schreiben.

Zum Erstellen einer Ereignisbehandlungsroutine für *OnUpdateRecord* müssen folgende Schritte ausgeführt werden:

1. Markieren Sie die Datenmenge.
2. Wählen Sie die Seite *Ereignisse* im Objektinspektor aus.
3. Doppelklicken Sie in der Wertespalte von *OnUpdateRecord*, um den Quelltext-Editor zu öffnen.

Hier das Grundgerüst einer Behandlungsroutine für *OnUpdateRecord*:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   { Hier werden die Aktualisierungen durchgeführt. }
end;
```

Der Parameter *DataSet* zeigt auf die zwischengespeicherte Datenmenge mit den Aktualisierungen.

Mit dem Parameter *UpdateKind* wird der Typ der auszuführenden Aktualisierung festgelegt. *UpdateKind* kann die Werte *ukModify*, *ukInsert* und *ukDelete* annehmen. Bei der Verwendung einer Update-Komponente übergeben Sie diesen Parameter an die Methoden, die für die Ausführung und die Wertzuweisung an die Parameter sorgen. Beispielsweise führt die Übergabe von *ukModify* an die Methode *Apply* zur Ausführung der *ModifySQL*-Anweisung des Update-Objekts. Eventuell muß dieser Parameter auch abgefragt werden, wenn die Behandlungsroutine speziell auf die Art der auszuführenden Aktualisierung reagiert.

Mit dem Parameter *UpdateAction* wird angezeigt, ob eine Aktualisierung stattgefunden hat oder nicht. *UpdateAction* kann die Werte *uaFail* (Vorgabe), *uaAbort*, *uaSkip*, *uaRetry* und *uaApplied* annehmen. In der Ereignisbehandlungsroutine sollte diesem Parameter der Wert *uaApplied* zugewiesen werden, es sei denn, es tritt während der Aktualisierung ein Problem auf. Soll ein bestimmter Datensatz nicht aktualisiert werden und sollen die nicht vorgenommenen Änderungen im Zwischenspeicher erhalten bleiben, wird der Wert *uaSkip* zugewiesen.

Wenn der Wert von *UpdateAction* nicht geändert wird, führt das zum Abruch des gesamten Aktualisierungsvorgangs. Weitere Informationen zu *UpdateAction* finden Sie unter »Die durchzuführende Aktion festlegen« auf Seite 25-28.

Neben diesen Parametern werden üblicherweise auch die Eigenschaften *OldValue* und *NewValue* der mit dem aktuellen Datensatz verbundenen Feldkomponente verwendet. Weitere Informationen zu *OldValue* und *NewValue* finden Sie unter »Die Feldeigenschaften OldValue, NewValue und CurValue« auf Seite 25-29.

Wichtig Wie bei *OnUpdateError* und *OnCalcFields* sollte eine Behandlungsroutine auch für *OnUpdateRecord* keine Methoden aufrufen, die einen anderen Datensatz zum aktuellen machen.

Die folgende Ereignisbehandlungsroutine für *OnUpdateRecord* führt zwei Update-Komponenten mittels ihrer *Apply*-Methoden aus. Durch die Übergabe des Parameters *UpdateKind* an *Apply* wird bestimmt, welche SQL-Anweisung in den einzelnen Update-Objekten ausgeführt werden soll.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   EmployeeUpdateSQL.Apply(UpdateKind);
   JobUpdateSQL.Apply(UpdateKind);
   UpdateAction := uaApplied;
end;
```

Da die Update-Komponenten nicht über die Eigenschaft *UpdateObject* der Datenmengenkomponente mit dieser verknüpft sind, findet der Parameter *DataSet* in diesem Beispiel keine Vewendung.

Fehlerbehandlung

Es vergeht eine gewisse Zeit, bis Änderungen an einem zwischengespeicherten Datensatz in die Datenbank eingetragen werden. Mittlerweile könnte eine andere Anwendung den Datensatz bearbeitet haben. Aktualisierungsfehler können aber auch auftreten, wenn keine Konflikte aufgrund von Datenänderungen anderer Benutzer vorhanden sind. Die Borland Database Engine (BDE) überprüft vor dem Eintragen der Aktualisierungen ausdrücklich, ob Konflikte oder andere Fehlerbedingungen vorliegen, und meldet gegebenenfalls einen Fehler.

Das Abfragen und Beseitigen von Fehlern kann in einer Behandlungsroutine für das Ereignis *OnUpdateError* der Datenmengenkomponente implementiert werden. Sie sollten für alle zwischengespeicherten Aktualisierungen eine derartige Routine schreiben. Existiert sie nicht, schlägt bei Auftreten eines Fehlers die ganze Aktualisierung fehl.

Achtung In einer Behandlungsroutine für *OnUpdateError* darf keine Methode der Datenmenge aufgerufen werden, die einen anderen Datensatz zum aktuellen macht (wie zum Beispiel *Next* oder *Prior*), da sonst die Ereignisbehandlungsroutine in eine Endlosschleife gerät.

Hier die Grundstruktur einer Ereignisbehandlungsroutine für *OnUpdateError*:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   { ... hier folgt die Fehlerbehandlung ... }
end;
```

Die folgenden Abschnitte befassen sich mit der Implementierung einer Fehlerbehandlung in einer *OnUpdateError*-Ereignisbehandlungsroutine und den Parametern für dieses Ereignis.

Referenz auf die betroffene Datenmenge

DataSet referenziert die Datenmenge, die aktualisiert werden soll. Diese Referenz ist Voraussetzung für die Verarbeitung von neuen und alten Datensatzwerten in einer Fehlerbehandlungsroutine.

Feststellen, wie der aktuelle Datensatz geändert wurde

Das Ereignis *OnUpdateRecord* übernimmt den Parameter *UpdateKind*. Dieser hat den Typ *TUpdateKind*. Der Parameter beschreibt die Art der Aktualisierung, die zum Fehler geführt hat. Wenn die Fehlerbehandlungsroutine nicht speziell auf die Art der ausgeführten Aktualisierung reagiert, kommt dieser Parameter vermutlich nicht zum Einsatz.

Die folgende Tabelle enthält die zulässigen Werte für *UpdateKind*:

Tabelle 25.3 Werte von UpdateKind

Wert	Bedeutung
ukModify	Die Bearbeitung eines vorhandenen Datensatzes hat zu einem Fehler geführt.
ukInsert	Das Einfügen eines neuen Datensatzes hat zu einem Fehler geführt.
ukDelete	Das Löschen eines vorhandenen Datensatzes hat zu einem Fehler geführt.

Das folgende Beispiel zeigt ein Konstrukt, in dem aufgrund des Wertes von *UpdateKind* entschieden wird, welche Operation ausgeführt wird.

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
   UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
   case UpdateKind of
    ukModify:
      begin
        { Fehler behandeln, der bei der Aktualisierung eines geänderten Datensatzes auftritt }
      end;
    ukInsert:
      begin
        { Fehler behandeln, der bei der Aktualisierung eines eingefügten Datensatzes auftritt }
      end;
    ukDelete:
      begin
        { Fehler behandeln, der bei der Aktualisierung eines gelöschten Datensatzes auftritt }
      end;
   end;
end;
```

Die durchzuführende Aktion festlegen

Der Parameter *UpdateAction* ist vom Typ *TUpdateAction*. Wenn die Ereignisbehandlungsroutine für Update-Fehler erstmals aufgerufen wird, hat dieser Parameter immer den Wert *uaFail*. In Abhängigkeit von der Fehlerbedingung des Datensatzes, der den Fehler hervorgerufen hat, und von der im Programm implementierten Reaktion wird *UpdateAction* vor dem Verlassen der Bearbeitungsroutine ein anderer Wert zugewiesen. Für *UpdateAction* sind folgende Werte möglich:

Tabelle 25.4 Werte von UpdateAction

Wert	Bedeutung
uaAbort	Die Aktualisierung wird ohne Anzeige einer Fehlermeldung abgebrochen.
uaFail	Die Aktualisierung wird abgebrochen und eine Fehlermeldung angezeigt. Dies ist die Voreinstellung für *UpdateAction*.
uaSkip	Der Datensatz wird nicht aktualisiert, aber seine Aktualisierung bleibt im Zwischenspeicher erhalten.
uaRetry	Die Aktualisierung wird erneut ausgeführt. Die Fehlerbedingung muß beseitigt werden, bevor *UpdateAction* dieser Wert zugewiesen wird.
uaApplied	Wird in Fehlerbehandlungsroutinen nicht verwendet.

Kann der Fehler, der den Aufruf der Behandlungsroutine ausgelöst hat, mit der Fehlerbehandlungsroutine behoben werden, sollte *UpdateAction* die Aktion zugewiesen werden, die nach der Unterbrechung ausgeführt werden soll. Damit die Aktualisierung des Datensatzes erneut versucht werden kann, sollte *UpdateAction* nach dem Beheben von Fehlern den Wert *uaRetry* erhalten.

Wird der Wert *uaSkip* zugewiesen, wird der Datensatz, der den Fehler verursacht hat, bei der Aktualisierung übersprungen. Die Aktualisierung bleibt aber im Zwischenspeicher, auch wenn die restlichen Aktualisierungen bereits vorgenommen wurden.

Die beiden Werte *uaFail* und *uaAbort* beenden die Aktualisierung. Der Wert *uaFail* löst eine Exception aus und bewirkt die Anzeige einer Fehlermeldung. Der Wert *uaAbort* löst eine Exception aus, ohne eine Fehlermeldung anzuzeigen.

Hinweis Wenn während des Zurückschreibens von Aktualisierungen ein Fehler auftritt, wird eine Exception ausgelöst und eine Fehlermeldung angezeigt. Wenn *ApplyUpdates* nicht innerhalb eines **try...except**-Konstrukts aufgerufen wurde und Sie in einer Behandlungsroutine für das Ereignis *OnUpdateError* die Anzeige einer Fehlermeldung festgelegt haben, kann es vorkommen, daß dieselbe Fehlermeldung zweimal angezeigt wird. Sie können doppelte Fehlermeldungen ausschließen, indem Sie *UpdateAction* mit *uaAbort* belegen. Die vom System generierte Fehlermeldung wird dadurch nicht angezeigt.

Der Wert *uaApplied* sollte nur innerhalb eines *OnUpdateRecord*-Ereignisses Verwendung finden. Verwenden Sie diesen Wert nicht in einer Behandlungsroutine für Update-Fehler. Weitere Information zu *OnUpdateRecord* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25.

Eine Fehlermeldung definieren

Der Parameter *E* ist üblicherweise vom Typ *EDBEngineError*. Mit Hilfe dieses Exception-Typs kann eine Fehlermeldung erstellt werden, die dem Benutzer in einer Fehlerbehandlungsroutine angezeigt wird. Die folgende Anweisung zeigt eine Fehlermeldung im Titel eines Dialogfeldes an:

```
ErrorLabel.Caption := E.Message;
```

Der Parameter kann auch zur Ermittlung der eigentlichen Ursache eines Update-Fehlers verwendet werden. Sie können spezielle Fehlercodes aus *EDBEngineError* ableiten und entsprechende Aktionen durchführen. Im folgenden Beispiel wird geprüft, ob der Update-Fehler mit einem Indexfehler in Zusammenhang steht. In diesem Fall wird dem Parameter *UpdateAction* der Wert *uaSkip* zugewiesen:

```
{ Fügen Sie für dieses Beispiel 'Bde' in die uses-Klausel ein }
if (E is EDBEngineError) then
   with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip { Indexfehler, Datensatz überspringen }
    else
      UpdateAction := uaAbort; { Fehlerursache nicht feststellbar, Aktualisierung abbrechen }
   end;
```

Die Feldeigenschaften OldValue, NewValue und CurValue

Ist die Zwischenspeicherung aktiviert, wird der ursprüngliche Wert der Felder in der *TField*-Eigenschaft *OldValue* gespeichert, bevor anstehende Aktualisierungen die Feldwerte ändern. Diese Eigenschaft ist schreibgeschützt. Die neuen Werte werden in der Eigenschaft *NewValue* gespeichert. Bei Client-Datenmengen speichert die zusätzliche *TField*-Eigenschaft *CurValue* den aktuellen Wert eines Feldes. *CurValue* ist mit *OldValue* identisch, es sei denn, ein anderer Benutzer hat den Datensatz geändert. In diesem Fall enthält *CurValue* den aktuellen Feldwert, der von diesem Benutzer gespeichert wurde.

Diese Eigenschaften stellen die einzige Möglichkeit dar, in Ereignisbehandlungsroutinen für *OnUpdateError* und *OnUpdateRecord* Aktualisierungswerte zu überprüfen und zu ändern. Weitere Informationen über *OnUpdateRecord* finden Sie unter »Eine Ereignisbehandlungsroutine für OnUpdateRecord erzeugen« auf Seite 25-25.

In einigen Fällen kann mit den Eigenschaften *OldValue*, *NewValue* und *CurValue* die Ursache eines Fehlers ermittelt und behoben werden. Im folgenden Programmbeispiel wird ein Feld korrigiert, dessen Wert aufgrund einer Server-Beschränkung nur in Schritten von 25 % erhöht werden darf.

```
var
   SalaryDif: Integer;
   OldSalary: Integer;
begin
   OldSalary := EmpTabSalary.OldValue;
   SalaryDif := EmpTabSalary.NewValue - OldSalary;
   if SalaryDif / OldSalary > 0.25 then begin
    { Zuschlag ist zu hoch, auf 25% reduzieren. }
```

```
      EmpTabSalary.NewValue := OldSalary + OldSalary * 0.25;
      UpdateAction := uaRetry;
    end
    else
      UpdateAction := uaSkip;
end;
```

NewValue wird reduziert, wenn sich das Gehalt (*OldSalary*) durch das Zurückschreiben der zwischengespeicherten Aktualisierung um mehr als 25 % erhöhen würde. Anschließend wird ein weiterer Versuch unternommen, die Aktualisierung durchzuführen. Um die Ausführungsgeschwindigkeit zu erhöhen, wird der Parameter *OldValue* in einer lokalen Variablen gespeichert.

Kapitel

26

Datensensitive Steuerelemente

Dieses Kapitel beschreibt, wie mit Hilfe von datensensitiven visuellen Steuerelementen Daten aus Tabellen und Abfragen angezeigt und bearbeitet werden können. Ein Steuerelement wird als *datensensitiv* bezeichnet, wenn es die angezeigten Daten aus einer Datenbankquelle außerhalb der Anwendung bezieht und Änderungen in eine Datenquelle zurückschreiben kann.

Zuerst werden alle Basismerkmale beschrieben, die solchen Steuerelementen gemeinsam sind. Anschließend finden Sie Hinweise zum Einsatz der einzelnen Komponenten.

Die meisten datensensitiven Komponenten stellen Informationen dar, die in einer Datenmenge gespeichert sind. Diese Komponenten können eingeteilt werden in Steuerelemente, die ein einzelnes Feld darstellen, und Steuerelemente, die Datensatzmengen darstellen, wie etwa *DBGrid* oder auch *TDBCtrlGrid*. Zusätzlich gibt das Steuerelement *TDBNavigator* den Benutzern die Möglichkeit, zu bearbeiten.

Die komplexeren datensensitiven Steuerelemente für die Datenanalyse werden beschrieben in Kapitel 27, »Entscheidungskomponenten«.

Datensensitive Steuerelemente im Überblick

Die folgenden Themen sind für die Arbeit mit fast allen Steuerelementen von Bedeutung:

- Datensensitive Steuerelemente einer Datenmenge zuordnen
- Daten bearbeiten und aktualisieren
- Die Datenanzeige aktivieren und deaktivieren
- Datenanzeige aktualisieren
- Maus-, Tastatur- und Timer-Ereignisse

Datensteuerelemente sind datensensitive Komponenten, die über ein *DataSource*-Objekt mit einer Datenmenge verknüpft sind. Sie fügen datensensitive Steuerelemente aus der Registerkarte *Datensteuerung* der Komponentenpalette in die Formulare Ihrer Datenbankanwendung ein. Datensensitive Steuerelemente ermöglichen die Anzeige und Bearbeitung von Feldern mit Daten, die mit dem aktuellen Datensatz einer Datenmenge verbunden sind. Die Tabelle 26.1 gibt einen Überblick über die Steuerelemente, die in der Registerkarte *Datensteuerung* der Komponentenpalette zu finden sind.

Tabelle 26.1 Datensensitive Steuerelemente

Steuerelement	Beschreibung
TDBGrid	*TDBGrid* zeigt Informationen aus einer Datenquelle in einem tabellarischen Format an. Die Spalten im Gitter entsprechen den Spalten in der zugrundeliegenden Tabellen- oder Abfragedatenmenge. Die Zeilen im Gitter entsprechen den Datensätzen.
TDBNavigator	*TDBNavigator* führt Bewegungen durch die Datensätze der Datenmenge aus, um Datensätze zu aktualisieren, einzutragen, zu löschen, zu bearbeiten und die Datenanzeige zu aktualisieren.
TDBText	*TDBText* zeigt Daten aus einem Feld als Beschriftung an.
TDBEdit	*TDBEdit* zeigt Daten aus einem Feld in einem Eingabefeld an und ermöglicht ihre Bearbeitung.
TDBMemo	*TDBMemo* zeigt Daten aus einem Memofeld, einem mehrzeiligen Textfeld oder einem BLOB-Feld in einem bildlauffähigen, mehrzeiligen Eingabefeld an und ermöglicht ihre Bearbeitung.
TDBImage	*TDBImage* zeigt ein Grafikbild oder binäre BLOB-Daten in einem Grafikfeld an und ermöglicht ihre Bearbeitung.
TDBListBox	*TDBListBox* zeigt eine Liste mit Auswahlmöglichkeiten an, mit denen ein Feld des aktuellen Datensatzes aktualisiert werden kann.
TDBComboBox	*TDBComboBox* öffnet eine Dropdown-Liste mit Einträgen, die zur Aktualisierung eines Feldes dienen. Außerdem ermöglicht diese Komponente eine direkte Texteingabe genauso wie ein datensensitives Standardeingabefeld.
TDBCheckBox	*TDBCheckBox* zeigt eine boolesche Feldbedingung in einem Kontrollfeld an und ermöglicht das Festlegen einer Bedingung.
TDBRadioGroup	*TDBRadioGroup* zeigt mehrere sich gegenseitig ausschließende Optionen für ein Feld an und ermöglicht die Festlegung dieser Optionen.
TDBLookupListBox	*TDBLookupListBox* zeigt eine Liste mit Einträgen an, die in einer anderen Datenmenge basierend auf einem Feldwert gesucht werden.
TDBLookupComboBox	*TDBLookupComboBox* zeigt eine Liste mit Einträgen an, die in einer anderen Datenmenge basierend auf einem Feldwert gesucht werden. Diese Komponente ermöglicht außerdem genauso wie ein datensensitives Kombinationsfeld eine direkte Texteingabe.
TDBCtrlGrid	*TDBCtrlGrid* zeigt eine konfigurierbare Menge von gleichartigen datensensitiven Steuerelementen in einem Gitter an.
TDBRichEdit	*TDBRichEdit* dient zur Anzeige formatierter Felddaten.

Die Steuerelemente dieser Gruppe sind während des Entwurfs datensensitiv. Wenn Sie der Eigenschaft *DataSource* eines Steuerelements während der Anwendungsentwicklung eine aktive Datenquelle zuweisen, werden in den Steuerelementen sofort

aktuelle Daten angezeigt. Mit dem Felder-Editor läßt sich zur Entwurfszeit ein Bildlauf durch eine Datenmenge ausführen, um festzustellen, ob die Anwendung die Daten korrekt anzeigt, ohne daß dazu die Anwendung compiliert und ausgeführt werden muß. Weitere Informationen zum Felder-Editor finden Sie unter »Persistente Felder erstellen« auf Seite 19-6.

Zur Laufzeit werden in den Steuerelementen ebenfalls Daten angezeigt. Außerdem können die Daten bearbeitet werden, wenn das Steuerelement, die Anwendung und die mit der Anwendung verbundene Datenbank dies zulassen.

Datensensitive Steuerelemente einer Datenmenge zuordnen

Datensensitive Steuerelemente werden über Datenquellen mit Datenmengen verbunden. Eine Datenquellenkomponente dient als Bindeglied zwischen dem Steuerelement und der Datenmenge, welche die Daten enthält. Diese Komponenten werden im Abschnitt »Datenquellen verwenden« auf Seite 26-6 ausführlich beschrieben.

Folgendermaßen ordnen Sie ein datensensitives Steuerelement einer Datenmenge zu:

1. Fügen Sie eine Datenmenge und eine Datenquelle in ein Datenmodul oder in ein Formular ein. Weisen Sie den einzelnen Eigenschaften die nötigen Werte zu.
2. Fügen Sie ein datensensitives Steuerelement aus der Registerkarte *Datenzugriff* der Komponentenpalette in das Formular ein.
3. Belegen Sie die Eigenschaft *DataSource* des Steuerelements mit dem Namen der Datenquelle, aus der die Daten geholt werden.
4. Belegen Sie die Eigenschaft *DataField* des Steuerelements mit dem Namen des Feldes, das angezeigt werden soll, oder wählen Sie einen Feldnamen aus der Dropdown-Liste der Eigenschaft. Dieser Schritt ist nicht erforderlich, wenn es sich um Objekte der Klassen *TDBGrid*, *TDBCtrlGrid* oder *TDBNavigator* handelt, denn diese greifen auf alle verfügbaren Felder einer Datenmenge zu.
5. Setzen Sie die Eigenschaft *Active* der Datenmenge auf *True*. Die Daten werden nun im Steuerelement angezeigt.

Daten bearbeiten und aktualisieren

Alle Daten-Steuerelemente außer dem Navigator zeigen Daten aus einem Datenbankfeld an. Die angezeigten Daten können bearbeitet und aktualisiert werden, sofern die zugrundeliegende Datenmenge dies zuläßt.

Den Bearbeitungsmodus aktivieren

Eine Datenmenge muß sich im Bearbeitungsmodus befinden, damit die Daten geändert werden können. Die Eigenschaft *AutoEdit* der Datenquelle, mit der ein Steuerelement verknüpft ist, legt fest, ob die zugrundeliegende Datenmenge in den *dsEdit*-Modus wechselt, wenn die Daten in einem Steuerelement als Reaktion auf ein Tastatur- oder Maus-Ereignis geändert werden. Ist *AutoEdit True* (Voreinstellung), wird der *dsEdit*-Modus gesetzt, sobald die Bearbeitung beginnt. Wenn *AutoEdit* den Wert *False*

hat, müssen Sie ein *TDBNavigator*-Steuerelement mit einer *Bearbeiten*-Schaltfläche (oder mit einer anderen Methode) zur Verfügung stellen, damit der Benutzer den *dsEdit*-Status zur Laufzeit aktivieren kann. Weitere Informationen über das Steuerelement *TDBNavigator* finden Sie in »Navigation und Bearbeitung von Datenmengen« auf Seite 26-34.

Daten in einem Steuerelement bearbeiten

Die Eigenschaft *ReadOnly* eines datensensitiven Steuerelements legt fest, ob ein Benutzer die angezeigten Daten bearbeiten kann. Wenn die Eigenschaft den Wert *False* hat (Voreinstellung), kann der Benutzer Daten bearbeiten. Soll der Benutzer daran gehindert werden, setzen Sie *ReadOnly* auf *True*.

Die Eigenschaften der einem Steuerelement zugrundeliegenden Datenquelle und Datenmenge legen außerdem fest, ob der Benutzer den Datenbestand über ein Steuerelement bearbeiten und die Änderungen in die Datenmenge eintragen kann.

Die Eigenschaft *Enabled* der Datenquelle bestimmt, ob in den mit einer Datenquelle verknüpften Steuerelementen Feldwerte aus der Datenmenge angezeigt werden können. Damit wird gleichzeitig festgelegt, ob der Benutzer Werte bearbeiten und eintragen kann. Wenn *Enabled True* ist (Voreinstellung), können in den Steuerelementen Feldwerte angezeigt werden.

Die Eigenschaft *ReadOnly* der Datenmenge legt fest, ob vom Benutzer durchgeführte Änderungen in die Datenmenge eingetragen werden können. Wenn die Eigenschaft den Wert *False* hat (Voreinstellung), sind Änderungen möglich. Hat *ReadOnly* den Wert *True*, kann die Datenmenge nur gelesen werden.

Hinweis Eine weitere Nur-Lesen-Laufzeiteigenschaft namens *CanModify* bestimmt, ob eine Datenmenge geändert werden kann. *CanModify* ist auf *True* gesetzt, wenn eine Datenbank Schreibzugriffe erlaubt. Ist *CanModify False*, kann die Datenmenge nur gelesen werden. Abfragekomponenten, die Einfügungen und Aktualisierungen ausführen, sind per Definition in der Lage, in eine zugrundeliegende Datenbank zu schreiben. Voraussetzung dafür ist, daß die Anwendung und der Benutzer ausreichende Schreibprivilegien für die Datenbank selbst besitzen.

In der folgenden Tabelle sind die Faktoren aufgeführt, die festlegen, ob ein Benutzer Daten in einem Steuerelement bearbeiten und Änderungen in die Datenbank eintragen kann.

Table 26.2 Eigenschaften, die die Bearbeitungsmöglichkeiten in datensensitiven Steuerelementen beeinflussen

Komponente	Eigenschaft				
Datensteuerelement	ReadOnly	False	False	False	True
Datenquelle	Enabled	True	True	False	—
Datenmenge	ReadOnly	False	True	—	—
Datenmenge	CanModify	True	False	—	—
(Datenbank)	Schreibzugriff	Lesen / Schreiben	Nur lesen	—	—
Schreibzugriff auf Datenbank?		Ja	Nein	Nein	Nein

In allen datensensitiven Steuerelementen mit Ausnahme von *TDBGrid* wird jede von Ihnen durchgeführte Änderung in einem Feld in die zugrundeliegende Feldkomponente der Datenmenge kopiert, sobald das Steuerelement verlassen wird. Wenn Sie die Taste *Esc* drücken, bevor Sie das Feld mit *Tab* verlassen, verwirft das Steuerelement die Änderungen, und der Feldwert wird wieder auf seinen ursprünglichen, vor der Änderung gültigen Wert zurückgesetzt.

In *TDBGrid*-Komponenten werden Änderungen nur dann kopiert, wenn Sie zu einem anderen Datensatz wechseln. Sie können vorher die *Esc*-Taste drücken, um alle Änderungen des Datensatzes aufzuheben.

Wenn ein Datensatz eingetragen wird, überprüft Delphi alle mit der Datenmenge verbundenen datensensitiven Komponenten in bezug auf eine Statusänderung. Treten bei der Aktualisierung eines Feldes, das geänderte Daten enthält, Probleme auf, löst Delphi eine Exception aus, und der Datensatz wird nicht geändert.

Die Datenanzeige aktivieren und deaktivieren

Wenn Ihre Anwendung über eine Datenmenge iteriert oder eine Suche ausführt, sollten Sie vorübergehend die Aktualisierung der Werte verhindern, die bei jedem Wechsel des aktuellen Datensatzes in den datensensitiven Steuerelementen erfolgt. Die Unterdrückung dieser Aktualisierung beschleunigt die Iteration bzw. die Suche und verhindert störendes Bildschirmflackern.

Die Methode *DisableControls* deaktiviert die Anzeige aller datensensitiven Steuerelemente, die mit einer Datenmenge verknüpft sind. Unmittelbar nach Beendigung der Iteration oder Suche muß Ihre Anwendung die Methode *EnableControls* aufrufen, um die Anzeige der Steuerelemente wieder zu aktivieren.

Normalerweise deaktivieren Sie die Steuerelemente vor Beginn eines Iterationsprozesses. Die Iteration sollte innerhalb einer **try...finally**-Anweisung stattfinden, so daß Sie die Steuerelemente auch dann wieder aktivieren können, wenn während des Prozesses eine Exception auftritt. Die **finally**-Klausel sollte die Methode *EnableControls* zusätzlich zum Aufruf von *EnableControls* außerhalb des **try..finally**-Blocks aufrufen. Der folgende Quelltext zeigt, wie Sie *DisableControls* und *EnableControls* in diesem Sinne verwenden könnten:

```
CustTable.DisableControls;
try
  CustTable.First; { Ersten Datensatz aufrufen, EOF wird False zugewiesen }
  while not CustTable.EOF do { Datensätze aufrufen, bis EOF True wird }
  begin
    { Datensatzverarbeitung }
    %
    CustTable.Next; { Bei Erfolg ist EOF False, sonst True (letzter Datensatz) }
  end;
finally
  CustTable.EnableControls;
end;
```

Datenanzeige aktualisieren

Die Methode *Refresh* einer Datenmenge leert die lokalen Puffer und ruft die Daten einer geöffneten Datenmenge neu ab. Mit dieser Methode können Sie die Anzeige in datensensitiven Steuerelementen aktualisieren, wenn Sie Grund zu der Annahme haben, daß sich die zugrundeliegenden Daten geändert haben. Dazu kann es kommen, wenn andere Anwendungen zur gleichen Zeit auf die Daten in Ihrer Anwendung zugreifen.

Wichtig Aktualisierungen führen unter Umständen zu unerwünschten Ergebnissen. Wenn beispielsweise ein Benutzer einen Datensatz anzeigt, der zwischenzeitlich von einer anderen Anwendung gelöscht wurde, wird dieser in dem Augenblick ausgeblendet, in dem in Ihrer Anwendung *Refresh* aufgerufen wird. Daten können sich auch ändern, wenn ein anderer Benutzer einen Datensatz bearbeitet, nachdem Sie die Daten abgerufen, aber bevor Sie *Refresh* aufgerufen haben. In bestimmten Situationen kann es sinnvoll sein, vor dem Löschen oder Aktualisieren von Daten die Methode *Refresh* aufzurufen.

Maus-, Tastatur- und Timer-Ereignisse

Die Eigenschaft *Enabled* eines datensensitiven Steuerelements legt fest, ob das Steuerelement auf Maus-, Tastatur- oder Timer-Ereignisse reagiert, und übergibt Informationen an die zugehörige Datenquelle. Die Voreinstellung für diese Eigenschaft lautet *True*.

Wenn Maus-, Tastatur- und Timer-Ereignisse am Zugriff auf ein datensensitives Steuerelement gehindert werden sollen, setzen Sie die Eigenschaft *Enabled* auf *False*. Wenn *Enabled* den Wert *False* hat, empfängt eine Datenquelle keine Informationen vom datensensitiven Steuerelement. Im datensensitiven Steuerelement werden die Daten weiterhin angezeigt, aber der Text wird in dunklerem Grau dargestellt.

Datenquellen verwenden

Eine *TDataSource*-Komponente ist eine nichtvisuelle Datenbankkomponente, die als Bindeglied zwischen einer Datenmenge und datensensitiven Steuerelementen in einem Formular dient, mit deren Hilfe die Daten der Datenmenge angezeigt und bearbeitet werden können. Jedes datensensitive Steuerelement muß einer Datenquelle zugeordnet werden, damit sie Zugriff auf die anzuzeigenden und gegebenenfalls zu bearbeitenden Daten erhält. Entsprechend müssen allen Datenmengen Datenquellen zugeordnet werden, damit die Daten in datensensitiven Steuerelementen in einem Formular angezeigt und bearbeitet werden können.

Hinweis Wenn Sie mit Datenzugriffskomponenten arbeiten, die sich auf der Registerkarte *Interbase* der Komponentenpalette befinden, müssen Sie statt dessen eine *TIBDataSource*-Komponente benutzen. Sie ist mit *TDataSource* weitgehend funktionsgleich.

Hinweis Mit Datenquellenkomponenten können Sie auch Datenmengen in einer Haupt/Detail-Beziehung miteinander verbinden.

Sie können eine Datenquellenkomponente wie andere nicht-visuelle Datenbankkomponenten in ein Datenmodul oder Formular einfügen. Für jede Datenmengenkomponente in einem Datenmodul oder Formular sollten Sie mindestens eine Datenquellenkomponente einfügen.

TDataSource-Eigenschaften verwenden

TDataSource besitzt nur wenige öffentliche Eigenschaften. In den folgenden Abschnitten werden diese Eigenschaften sowie deren Einstellung zur Laufzeit und Entwurfszeit erläutert.

Eigenschaft DataSet einstellen

Die Eigenschaft *DataSet* gibt die Datenmenge an, aus der eine Datenquellenkomponente Daten abruft. Während des Entwurfs können Sie Datenmengen in einer Dropdown-Liste des Objektinspektors auswählen. Zur Laufzeit können Sie die Datenmenge einer Datenquellenkomponente jederzeit ändern. Der folgende Quelltext schaltet zwischen den Datenmengen *Customers* und *Orders* für die Datenquellenkomponente *CustSource* um:

```
with CustSource do
begin
   if DataSet = 'Customers' then
    DataSet := 'Orders'
   else
    DataSet := 'Customers';
end;
```

Sie können der Eigenschaft *DataSet* die Datenmenge eines anderen Formulars zuweisen, um die datensensitiven Steuerelemente der beiden Formulare zu synchronisieren:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
   DataSource1.Dataset := Form1.Table1;
end;
```

Eigenschaft Name einstellen

Mit Hilfe der Eigenschaft *Name* können Sie einen beschreibenden Namen für die Datenquellenkomponente angeben, der sie von allen anderen Datenquellen in einer Anwendung unterscheidet. Der hier für eine Datenquellenkomponente angegebene Name wird unter dem Symbol der Komponente im Datenmodul angezeigt.

Normalerweise stellen Sie in einer Datenquellenkomponente einen Namen bereit, der die zugeordnete Datenmenge bezeichnet. Liegt beispielsweise eine Datenmenge namens *Customers* vor, können Sie diese mit der Datenquellenkomponente verbinden, indem Sie der Eigenschaft *DataSet* dieser Datenquellenkomponente den Wert »Customers« zuweisen. Damit die Verbindung zwischen Datenmenge und Datenquelle im Datenmodul offensichtlich wird, können Sie der Eigenschaft *Name* der Datenquellenkomponente beispielsweise »CustomersSource« zuweisen.

Eigenschaft Enabled einstellen

Die Eigenschaft *Enabled* legt fest, ob eine Datenquellenkomponente mit ihrer Datenmenge verbunden ist. Enthält *Enabled* den Wert *True*, ist die Datenquelle mit der Datenmenge verbunden.

Sie können eine Datenquelle temporär von der Datenmenge trennen, indem Sie der Eigenschaft *Enabled* den Wert *False* zuweisen. Dadurch wird der Inhalt aller datensensitiven Steuerelemente gelöscht, die mit der Datenquellenkomponente verbunden sind. Sie bleiben deaktiviert, bis der Eigenschaft *Enabled* wieder *True* zugewiesen wird. Es wird jedoch empfohlen, den Zugriff auf eine Datenmenge mit den Methoden *DisableControls* und *EnableControls* der Datenmenge zu steuern, da diese alle angeschlossenen Datenquellen berücksichtigen.

Eigenschaft AutoEdit einstellen

Die Eigenschaft *AutoEdit* von *TDataSource* legt fest, ob für die mit eine Datenquelle verbundenen Datenmengen automatisch der Bearbeitungsstatus aktiviert wird, wenn der Benutzer Dateneingaben in den mit den Datenmengen verbundenen datensensitiven Steuerelementen vornimmt. Enthält *AutoEdit* den Wert *True* (Standard), wird automatisch der Bearbeitungsstatus aktiviert, wenn der Benutzer Eingaben in einem zugeordneten datensensitiven Steuerelement vornimmt. Andernfalls wird der Bearbeitungsstatus einer Datenmenge nur aktiviert, wenn die Anwendung explizit deren Methode *Edit* aufruft. Weitere Informationen zu den Stati von Datenmengen finden Sie im Abschnitt »Den Status von Datenmengen bestimmen und einstellen« auf Seite 18-4.

TDataSource-Ereignisse verwenden

TDataSource sind drei Ereignisbehandlungsroutinen zugeordnet:

- Ereignis *OnDataChange* verwenden
- Ereignis *OnUpdateData* verwenden
- Ereignis *OnStateChange* verwenden

Ereignis OnDataChange verwenden

Das Ereignis *OnDataChange* wird ausgelöst, wenn der Datensatzzeiger auf einen neuen Datensatz verschoben wird. Ruft die Anwendung *Next*, *Previous*, *Insert* oder eine andere Methode auf, die zu einer Verschiebung des Datensatzzeigers führt, wird das Ereignis *OnDataChange* ausgelöst.

Dieses Ereignis ist hilfreich, wenn Komponenten in einer Anwendung manuell synchronisiert werden müssen.

Ereignis OnUpdateData verwenden

Das Ereignis *OnUpdateData* tritt auf, wenn die Daten im aktuellen Datensatz aktualisiert werden sollen. Es tritt beispielsweise nach einem Aufruf von *Post* auf, bevor die Daten tatsächlich in die Datenbank eingetragen werden.

Dieses Ereignis ist hilfreich, wenn eine Anwendung Standardsteuerelemente verwendet, die nicht datensensitiv sind, und diese mit einer Datenmenge synchronisieren muß.

Ereignis OnStateChange verwenden

Das Ereignis *OnStateChange* tritt auf, wenn der Status der Datenmenge einer Datenquelle geändert wird. Die Eigenschaft *State* einer Datenmenge zeichnet deren aktuellen Status auf. *OnStateChange* ist nützlich, wenn beispielsweise der Status einer *TDataSource*-Komponente geändert wird.

Im Rahmen einer normalen Datenbanksitzung wird der Status einer Datenmenge häufig geändert. Sie können eine Ereignisbehandlungsroutine für *OnStateChange* schreiben, die den aktuellen Status der Datenmenge im Formular anzeigt. Der folgende Quelltext illustriert eine solche Routine. Zur Laufzeit wird die aktuelle Einstellung der Eigenschaft *State* der Datenmenge angezeigt. Die Anzeige wird aktualisiert, sobald sich deren Status ändert:

```
procedure TForm1.DataSource1.StateChange(Sender:TObject);
var
   S:String;
begin
   case CustTable.State of
    dsInactive: S := 'Inactive';
    dsBrowse: S := 'Browse';
    dsEdit: S := 'Edit';    dsInsert: S := 'Insert';
    dsSetKey: S := 'SetKey';
   end;
   CustTableStateLabel.Caption := S;
end;
```

OnStateChange kann auf ähnliche Weise eingesetzt werden, um Schaltflächen oder Menüoptionen in Abhängigkeit vom aktuellen Status zu aktivieren bzw. deaktivieren:

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
   CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
   CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
   ...
end;
```

Steuerelemente zur Darstellung eines einzelnen Feldes

Viele der Steuerelemente auf der Registerkarte mit den datensensitiven Steuerelementen in der Komponentenpalette repräsentieren ein einzelnes Feld in einer Datenbanktabelle. Erscheinungsbild und Funktion der meisten dieser Steuerelemente entsprechen Windows-Standardsteuerelementen. Die Steuerelemente werden in Formulare eingefügt. Das Steuerelement *TDBEdit* ist beispielsweise eine datensensitive Version des Standardsteuerelements *TEdit*, in dem Benutzer Strings anzeigen und bearbeiten können.

Das jeweils verwendete Steuerelement ist vom Typ der Daten im Feld abhängig (Text, formatierter Text, Grafik, Logisch usw.).

Felder als Beschriftung anzeigen

TDBText ist ein Nur-Lesen-Steuerelement, das den Komponenten *TLabel* auf der Registerkarte *Standard* der Komponentenpalette und der Komponente *TStaticText* ähnelt. Ein *TDBText*-Steuerelement ist von Nutzen, wenn Sie in einem Formular, das dem Benutzer Eingaben in Steuerelemente erlaubt, Daten zur Verfügung stellen wollen, die nur angezeigt werden. Angenommen, Sie haben ein Formular, das für die Felder in einer Kundenliste erstellt wurde. Sobald der Benutzer Adresse, Staat und Bundesland in das Formular eingibt, lassen Sie mit Hilfe einer dynamischen Suche die Postleitzahl in einer anderen Tabelle feststellen. Eine mit der Postleitzahlentabelle verknüpfte *TDBText*-Komponente könnte dann für die Anzeige der Postleitzahl verwendet werden, die zur eingegebenen Adresse gehört.

TDBText ruft den anzuzeigenden Text aus einem bestimmten Feld im aktuellen Datensatz einer Datenmenge ab. Da *TDBText* seinen Text aus einer Datenmenge ermittelt, ist dieser Text dynamisch, d.h., er ändert sich, sobald der Benutzer Bewegungen in der Datenbanktabelle ausführt. Aus diesem Grund können Sie den Anzeigetext für *TDBText* nicht zur Entwurfszeit festlegen, wie dies bei *TLabel* und *TStaticText* möglich ist.

Hinweis Wenn Sie eine *TDBText*-Komponente in ein Formular einfügen, setzen Sie ihre Eigenschaft *AutoSize* auf *True* (Voreinstellung). Damit ist sichergestellt, daß die Größe des Steuerelements an die Anzeige unterschiedlich breiter Daten angepaßt wird. Wenn *AutoSize* auf *False* gesetzt und das Steuerelement zu klein ist, werden die Daten abgeschnitten.

Feldinhalte in Eingabefeldern anzeigen und bearbeiten

TDBEdit ist die datensensitive Version eines Eingabefeldes. Die Komponente zeigt den aktuellen Wert des Datenfeldes an, mit dem es verbunden ist, und ermöglicht die Bearbeitung mit den bei Eingabefeldern üblichen Techniken.

Angenommen, *CustomersSource* ist eine *TDataSource*-Komponente, die aktiv und mit einer geöffneten *TTable*-Komponente namens *CustomersTable* verknüpft ist. Sie können nun eine *TDBEdit*-Komponente in ein Formular einfügen und deren Eigenschaften folgendermaßen festlegen:

- *DataSource*: *CustomersSource*
- *DataField*: *CustNo*

Im datensensitiven Eingabefeld wird sofort (während des Entwurfs und zur Laufzeit) der Wert der aktuellen Zeile in der Spalte *CustNo* der Datenmenge *CustomersTable* angezeigt.

Textfelder in einem Memo-Steuerelement anzeigen und bearbeiten

TDBMemo ist eine datensensitive Komponente (ähnlich der Standardkomponente *TMemo* , mit der ein mehrzeiliges Feld einer Datenmenge oder formatierter Text im BLOB-Format (BLOB = Binary Large Objekt) angezeigt und geändert werden kann. *TDBMemo* zeigt mehrzeiligen Text an und ermöglicht auch dessen Eingabe. Mit *TDBMemo*-Steuerelementen können Sie Memofelder aus dBASE- und Paradox-Tabellen und Textdaten aus BLOB-Feldern anzeigen.

Per Voreinstellung ermöglicht *TDBMemo* dem Benutzer die Bearbeitung von Memotext. Wenn Sie die Bearbeitung verhindern wollen, müssen Sie die Eigenschaft *ReadOnly* von *TDBMemo* auf *True* setzen. Soll dem Benutzer erlaubt werden, Tabulatoren in ein Memofeld einzufügen, setzen Sie die Eigenschaft *WantTabs* auf *True*. Die Anzahl der Zeichen, die der Benutzer in ein Datenbank-Memofeld eingeben kann, läßt sich mit der Eigenschaft *MaxLength* festlegen. Diese ist mit dem Wert 0 voreingestellt, was bedeutet, daß es außer den Auflagen des Betriebssystems keine Einschränkung für die Anzahl der Zeichen gibt, die das Steuerelement aufnehmen kann.

Es gibt mehrere Eigenschaften, die festlegen, wie das Datenbank-Memofeld angezeigt und der Text eingegeben wird. Mit Hilfe der Eigenschaft *ScrollBars* können Sie das Memofeld mit Bildlaufleisten ausstatten. Wenn Sie Zeilenumbrüche verhindern wollen, setzen Sie die Eigenschaft *WordWrap* auf *False*. Die Eigenschaft *Alignment* legt fest, wie der Text innerhalb der Steuerelemente ausgerichtet wird. Folgende Werte stehen für diese Eigenschaft zur Verfügung: *taLeftJustify* (Voreinstellung), *taCenter* und *taRightJustify*. Arbeiten Sie mit der Eigenschaft *Font*, wenn Sie die Schriftart für den Text ändern wollen.

Zur Laufzeit kann der Benutzer Text in einem Memo-Steuerelement ausschneiden, kopieren und einfügen. Diese Aufgaben können mit den Methoden *CutToClipboard*, *CopyToClipboard* und *PasteFromClipboard* im Programm implementiert werden.

Da die *TDBMemo*-Komponente umfangreiche Datenmengen anzeigen kann, wird zur Laufzeit möglicherweise einige Zeit benötigt, bis der gesamte Text dargestellt ist. Um die Zeit, die für den Bildlauf durch die Datensätze erforderlich ist, zu verkürzen, verfügt *TDBMemo* über die Eigenschaft *AutoDisplay*. Diese bestimmt, ob die Daten, auf die zugegriffen wird, automatisch angezeigt werden. Wenn Sie *AutoDisplay* auf *False* setzen, zeigt *TDBMemo* anstelle der tatsächlichen Daten den Feldnamen an. Durch einen Doppelklick im Steuerelement werden die tatsächlichen Daten eingeblendet.

Text in einem RTF-Eingabefeld anzeigen und bearbeiten

TDBRichEdit ist eine datensensitive Komponente ähnlich der Standardkomponente *TRichEdit*. Sie eignet sich zur Darstellung von formatiertem Text, der in einem BLOB-Feld gespeichert ist. *TDBMemo* kann formatierten Text in mehreren Zeilen anzeigen und läßt auch dessen Eingabe bzw. Bearbeitung zu. Das Objekt *TDBRichEdit* kann verwendet werden, um den Inhalt von Memofeldern aus dBASE- und Paradox-Tabellen anzuzeigen sowie Texte, die in BLOB-Feldern abgelegt sind.

Hinweis *TDBRichEdit* stellt zwar Eigenschaften und Methoden bereit, die für die Eingabe und Bearbeitung von RTF-Text benötigt werden, aber keine Oberflächenelemente, die dem Benutzer diese Formatierungsoptionen zugänglich machen. Hier müssen Sie als Programmierer Ihre Anwendung entsprechend ausstatten.

Per Voreinstellung ermöglicht *TDBRichEdit* die Bearbeitung des Memotextes durch den Benutzer. Um dies zu verhindern, setzen Sie die Eigenschaft *ReadOnly* auf *True*. Die Verwendung von Tabulatorzeichen wird mit der Eigenschaft *WantTabs* geregelt. Wenn diese Eigenschaft den Wert *True* hat, sind Tabs zulässig und werden auch angezeigt. Die maximale Zeichenzahl wird mit der Eigenschaft *MaxLength* begrenzt. Der Vorgabewert für *MaxLength* ist 0, was bedeutet, daß es außer den Auflagen des Betriebssystems keine Einschränkung für die Anzahl der Zeichen gibt, die das Steuerelement aufnehmen kann.

TDBRichEdit-Objekte können große Mengen an Daten aufnehmen, so daß es einige Zeit dauern kann, bis der Bildschirmaufbau abgeschlossen ist. Um Bildlaufvorgänge zu beschleunigen, besitzt *TDBRichEdit* deshalb eine Eigenschaft *AutoDisplay*, die festlegt, ob die im Zugriff befindlichen Daten automatisch angezeigt werden sollen. Wenn Sie *AutoDisplay* auf *False* setzen, zeigt *TDBRichEdit* anstelle der tatsächlichen Daten nur den Feldnamen an.

Grafikfelder in einem Bild-Steuerelement anzeigen und bearbeiten

TDBImage ist eine datensensitive Komponente, mit der Bitmap-Grafiken oder BLOB-Daten im Bildformat angezeigt werden. Die Komponente übernimmt BLOB-Grafiken aus einer Datenmenge und speichert sie intern im Windows-Format DIB.

TDBImage läßt standardmäßig zu, daß der Benutzer ein Grafikbild durch Ausschneiden und Einfügen aus der bzw. in die Zwischenablage bearbeitet. Sie können aber auch eigene Bearbeitungsmethoden einsetzen. So können Sie beispielsweise die Methoden *CutToClipboard*, *CopyToClipboard* und *PasteFromClipboard* verwenden. Ebenso lassen sich eigene Bearbeitungsmethoden entwickeln und mit den Ereignisbehandlungsroutinen des Steuerelements verknüpfen.

Setzen Sie die Eigenschaft *Stretch* auf *True*, wenn das Bild automatisch vergrößert oder verkleinert werden soll, damit es in die verfügbare Fläche paßt. Ist *Stretch* auf *True* gesetzt (Voreinstellung), wird so viel von der Grafik im Steuerelement angezeigt, wie darin Platz findet.

Da *TDBImage* in der Lage ist, umfangreiche Datenmengen anzuzeigen, kann der Bildschirmaufbau zur Laufzeit möglicherweise etwas dauern. Um die Zeit, die für einen Bildlauf durch die Datensätze erforderlich ist, zu verkürzen, verfügt *TDBImage* über

die Eigenschaft *AutoDisplay*. Diese bestimmt, ob die Daten, auf die zugegriffen wird, automatisch angezeigt werden sollen. Setzen Sie *AutoDisplay* auf *False*, zeigt *TDBImage* anstelle der tatsächlichen Daten den Feldnamen an. Mit einem Doppelklick im Steuerelement werden die tatsächlichen Daten eingeblendet.

Daten in Listen- und Kombinationsfeldern anzeigen und bearbeiten

Es gibt vier Arten von Steuerelementen, die datensensitive Versionen von gewöhnlichen Listen- und Kombinationsfeldern darstellen. Diese nützlichen Komponenten stellen dem Benutzer eine Reihe von voreingestellten Datenwerten zur Verfügung, aus denen zur Laufzeit eine Auswahl getroffen wird.

Hinweis Datensensitive Listen und Kombinationsfelder können nur mit Datenquellen für Tabellenkomponenten verknüpft werden. Sie lassen sich nicht zusammen mit Abfragekomponenten verwenden.

Die folgende Tabelle enthält eine Beschreibung dieser Steuerelemente:

Tabelle 26.3 Datensensitive Listen- und Kombinationsfelder

Datensensitives Steuerelement	Beschreibung
TDBListBox	*TDBListBox* zeigt eine Liste mit Auswahlmöglichkeiten an, mit denen der Benutzer ein Feld des aktuellen Datensatzes aktualisieren kann. Die Liste der angezeigten Einträge ist eine Eigenschaft des Steuerelements.
TDBComboBox	*TDBComboBox* ist eine Kombination aus einem Eingabe- und einem Listenfeld. Der Benutzer kann ein Feld im aktuellen Datensatz aktualisieren, indem er einen Wert aus der Dropdown-Liste auswählt oder einen Wert eingibt. Die Liste der angezeigten Einträge ist eine Eigenschaft des Steuerelements.
TDBLookupListBox	*TDBLookupListBox* zeigt eine Liste mit Auswahlmöglichkeiten an, mit denen der Benutzer eine Spalte des aktuellen Datensatzes aktualisieren kann. Die Einträge in der Liste stammen aus einer anderen Datenmenge.
TDBLookupComboBox	*TDBLookupComboBox* ist eine Kombination aus einem Eingabe- und einem Listenfeld. Der Benutzer kann ein Feld im aktuellen Datensatz aktualisieren, indem er einen Wert aus der Dropdown-Liste auswählt oder einen Wert eingibt. Die Einträge in der Liste stammen aus einer anderen Datenmenge.

Daten in Listenfeldern anzeigen und bearbeiten

TDBListBox zeigt eine bildlauffähige Liste mit Einträgen an, mit denen der Benutzer ein Datenfeld des aktuellen Datensatzes in einer Datenmenge aktualisieren kann. In einem datensensitiven Listenfeld wird der aktuelle Wert in einem Feld des aktuellen Datensatzes angezeigt und der entsprechende Eintrag in der Liste optisch hervorgehoben. Wenn sich der Feldwert der aktuellen Zeile nicht in der Liste befindet, wird kein Wert im Listenfeld markiert. Wählt der Benutzer einen Listeneintrag aus, wird der entsprechende Feldwert in der zugrundeliegenden Datenmenge geändert.

Mit Hilfe des Stringlisten-Editors läßt sich zur Entwurfszeit die Liste mit den in der Eigenschaft *Items* anzuzeigenden Einträgen zusammenstellen. Die Eigenschaft *Height* legt die vertikale Größe (Höhe) eines Steuerelements in Pixel fest. Mit der Eigenschaft *IntegralHeight* bestimmen Sie, wie das Listenfeld angezeigt wird. Ist *IntegralHeight* auf *False* gesetzt (Voreinstellung), wird das Ende des Listenfeldes von der Eigenschaft *ItemHeight* bestimmt, wobei unter Umständen der letzte Eintrag nicht vollständig angezeigt wird. Wenn *IntegralHeight True* ist, wird der sichtbare Eintrag am Ende des Listenfeldes vollständig eingeblendet.

Daten in Kombinationsfeldern anzeigen und bearbeiten

TDBComboBox ist eine datensensitive Version der Komponente *TComboBox*. Im Steuerelement *TDBComboBox* sind die Merkmale eines datensensitiven Eingabefeldes und einer Dropdown-Liste vereinigt. Der Benutzer kann zur Laufzeit ein Feld im aktuellen Datensatz einer Datenmenge aktualisieren, indem er einen Wert eingibt oder ihn aus der Dropdown-Liste auswählt.

Die Eigenschaft *Items* der Komponente bezeichnet die Elemente, die in der Dropdown-Liste enthalten sein sollen. Zur Entwurfszeit füllen Sie die Liste mit Hilfe des Stringlisten-Editors. Zur Laufzeit sind verschiedene Methoden verfügbar, mit denen sich die Liste bearbeiten läßt.

Wenn ein Steuerelement über seine *DataField*-Eigenschaft mit einem Feld verknüpft ist, zeigt es den Wert des Feldes im aktuellen Datensatz an, unabhängig davon, ob dieser Wert in der *Items*-Liste vorhanden ist oder nicht. Die Eigenschaft *Style* legt die Form der Interaktion zwischen Benutzer und Steuerelement fest. *Style* ist per Voreinstellung mit dem Wert *csDropDown* belegt. Das bedeutet, daß der Benutzer Werte über die Tastatur eingeben oder einen Eintrag aus der Dropdown-Liste auswählen kann. Die folgenden Eigenschaften legen fest, wie die Liste *Items* zur Laufzeit angezeigt wird:

- *Style* bestimmt den Anzeigestil der Komponente:
- *csDropDown* (Voreinstellung): Zeigt eine Dropdown-Liste mit einem Eingabefeld an, in das der Benutzer Text eingeben kann. Alle Einträge sind Strings und haben dieselbe Höhe.
- *csSimple*: Kombiniert ein Eingabefeld mit einer Liste von Einträgen. Die Liste hat eine feste Größe und ist ständig geöffnet. Wenn Sie *Style* auf *csSimple* setzen, müssen Sie auch die Eigenschaft *Height* entsprechend einstellen.
- *csDropDownList*: Zeigt eine Dropdown-Liste und ein Eingabefeld an. Der Benutzer kann jedoch nur diejenigen Werte eingeben oder ändern, die sich zur Laufzeit in der Dropdown-Liste befinden.
- *csOwnerDrawFixed* und *csOwnerDrawVariable*: Diese Eigenschaften ermöglichen die Anzeige von Werten in der *Items*-Liste, die nicht aus Strings bestehen (beispielsweise Bitmaps) oder für die unterschiedliche Schriften verwendet werden sollen.
- *DropDownCount*: Die maximale Anzahl der Einträge, die in der Liste angezeigt werden können. Wenn die Anzahl von *Items* höher als *DropDownCount* ist, kann der Benutzer einen Bildlauf in der Liste durchführen. Ist die Anzahl von *Items*

niedriger als *DropDownCount*, ist die Liste gerade so groß, daß alle Einträge sichtbar sind.

- *ItemHeight*: Die Höhe aller Einträge, wenn *Style* den Wert *csOwnerDrawFixed* hat.
- *Sorted*: Wenn diese Eigenschaft *True* ist, wird die *Items*-Liste in alphabetischer Reihenfolge sortiert.

Daten in Lookup-Listen und -Kombinationsfeldern anzeigen und bearbeiten

TDBLookupListBox und *TDBLookupComboBox* sind datensensitive Steuerelemente, die ihren Inhalt aus einer von zwei möglichen Quellen beziehen:

- Aus einem Lookup-Feld, das für eine Datenmenge definiert ist.
- Aus einer sekundären Datenquelle mit Datenfeld und Schlüssel.

In beiden Fällen erhält der Benutzer eine eingeschränkte Liste von Auswahlmöglichkeiten, mit denen ein gültiger Wert gesetzt werden kann. Wählt der Benutzer einen Listeneintrag aus, wird der entsprechende Feldwert in der zugrundeliegenden Datenmenge geändert.

Angenommen, es existiert ein Bestellformular, dessen Felder mit der Komponente *OrdersTable* verknüpft sind. *OrdersTable* enthält ein Feld namens CUSTNO, das einer Kundennummer entspricht. Weitere Kundeninformationen sind in *OrdersTable* nicht enthalten. Die Tabelle *CustomersTable* enthält ebenfalls ein Feld CUSTNO für die Kundennummer sowie weitere Informationen (z.B. die Firma und die Adresse des Kunden). Es wäre von Vorteil, wenn der Sachbearbeiter bei der Erstellung einer Rechnung die Möglichkeit hätte, im Bestellformular einen Kunden nach der Firma und nicht nach der Kundennummer auszuwählen. Mit Hilfe einer *TDBLookupListBox*-Komponente, die alle Firmennamen enthält, oder einer *TDBLookupComboBox*-Komponente, die eine Dropdown-Liste mit allen in *CustomersTable* vorhandenen Firmennamen anzeigt, könnte der Benutzer den gewünschten Firmennamen aus der Liste auswählen und CUSTNO im Bestellformular die entsprechende Kundennummer zuweisen.

Listen, die auf Lookup-Feldern basieren

Damit Listeneinträge über ein Lookup-Feld erstellt werden können, muß in der Datenmenge, mit der das Steuerelement verknüpft wird, bereits ein Lookup-Feld definiert sein. Weitere Informationen über die Definition von Lookup-Feldern für eine Datenmenge finden Sie unter »Lookup-Felder definieren« auf Seite 19-11.

Zur Definition eines Lookup-Feldes für Listen- oder Dropdown-Listeneinträge führen Sie folgende Arbeitsschritte aus:

1. Belegen Sie die Eigenschaft *DataSource* des Listenfeldes mit der Datenquelle für die Datenmenge, die das gewünschte Lookup-Feld enthält.
2. Wählen Sie das gewünschte Lookup-Feld aus der Dropdown-Liste der Eigenschaft *DataField*.

Wenn Sie eine Tabelle aktivieren, die mit einem Lookup-Listenfeld oder einem Lookup-Kombinationsfeld verbunden ist, erkennt das Steuerelement, daß es sich bei seinem Datenfeld um ein Lookup-Feld handelt, und zeigt die entsprechenden Werte aus dem Suchvorgang an.

Listen, die auf einer sekundären Datenquelle basieren

Wenn kein Lookup-Feld für eine Datenmenge definiert ist, können Sie eine ähnliche Beziehung herstellen. Dazu benötigen Sie eine sekundäre Datenquelle, einen Feldwert, der darin gesucht werden soll, und einen Feldwert, der als Listeneintrag zurückgegeben wird.

Zur Definition einer sekundären Datenquelle für Listenfeld- oder Dropdown-Listeneinträge führen sie folgende Arbeitsschritte aus:

1. Belegen Sie die Eigenschaft *DataSource* des Listenfeldes mit der Datenquelle für die Datenmenge, in der Werte nachgeschlagen werden sollen.
2. Wählen Sie ein Feld aus, in das die nachgeschlagenen Werte aus der Dropdown-Liste der Eigenschaft *DataField* eingefügt werden sollen. Das für diesen Zweck ausgewählte Feld darf kein Lookup-Feld sein.
3. Belegen Sie die Eigenschaft *ListSource* des Listenfeldes mit der Datenquelle für die Datenmenge, die das Feld enthält, dessen Werte nachgeschlagen werden sollen.
4. Wählen Sie aus der Dropdown-Liste der Eigenschaft *KeyField* ein Feld aus, das als Lookup-Schlüssel dienen soll. Die Dropdown-Liste zeigt Felder der Datenmenge an, die mit der in Schritt 3 angegebenen Datenquelle verbunden ist. Das ausgewählte Feld braucht nicht indiziert zu sein. Wenn das aber der Fall ist, erhöht sich die Geschwindigkeit beträchtlich.
5. Wählen Sie ein Feld aus der Dropdown-Liste der Eigenschaft *ListField*, dessen Werte zurückgeliefert werden sollen. Die Dropdown-Liste enthält Felder der Datenmenge, die mit der in Schritt 3 angegebenen Datenquelle verbunden ist. Zur Anzeige mehrerer Felder trennen Sie die Feldnamen mit Semikolons voneinander ab.

Wenn Sie eine Tabelle aktivieren, die mit einem Lookup-Listenfeld oder einem Lookup-Kombinationsfeld verbunden ist, erkennt das Steuerelement, daß seine Dropdown-Listeneinträge von einer sekundären Datenquelle abgeleitet sind, und zeigt die entsprechenden Werte aus dieser Quelle an.

Eigenschaften von Lookup-Listen und Lookup-Kombinationsfeldern

In der folgenden Tabelle sind die für Lookup-Listen und Lookup-Kombinationsfelder wesentlichen Eigenschaften aufgeführt:

Tabelle 26.4 Eigenschaften von TDBLookupListBox und TDBLookupComboBox

Eigenschaft	Zweck
DataField	Diese Eigenschaft gibt das Feld in der Hauptdatenmenge an, das den Schlüsselwert für die Suche in der Lookup-Datenmenge enthält. Dieses Feld ändert sich, wenn der Benutzer einen Eintrag aus einem Listenfeld oder aus einer Dropdown-Liste auswählt. Wenn *DataField* mit einem Lookup-Feld belegt ist, werden die Eigenschaften *KeyField*, ListField und *ListSource* nicht verwendet (z.B. CUSTNO aus *OrdersSource*).
DataSource	Diese Eigenschaft gibt eine Datenquelle für das Steuerelement an. Dabei sollte es sich um die Datenquelle für die Hauptdatenmenge handeln. Wenn sich die Auswahl im Steuerelement ändert, wird diese Datenmenge in den Modus *dsEdit* versetzt (z.B. *OrdersSource*).
KeyField	Diese Eigenschaft bestimmt das Feld in der Lookup-Datenmenge, das dem Wert von *DataField* entspricht. Das Steuerelement sucht nach dem *DataField*-Wert in der Eigenschaft *KeyField* der Lookup-Datenmenge. Die Spalte der Eigenschaft *KeyField* muß dieselben Werte wie die Spalte der Eigenschaft *DataField* enthalten. Die Spaltennamen müssen jedoch nicht identisch sein. Die Lookup-Datenmenge sollte einen Index auf dieses Feld besitzen, um die Suchvorgänge zu erleichtern (z.B. CUSTNO aus *CustomersSource*).
ListField	Diese Eigenschaft legt das Feld der Lookup-Datenmenge fest, das im Steuerelement angezeigt wird (z.B. CUSTNO; COMPANY aus *CustomersSource*).
ListSource	Diese Eigenschaft gibt eine Datenquelle für die Datenmenge an, die das Steuerelement für die Suche nach den anzuzeigenden Informationen verwendet. Die Sortierreihenfolge in der Liste oder der Dropdown-Liste wird von dem Index bestimmt, der mit der Eigenschaft *IndexName* der Lookup-Datenmenge angegeben ist. Dieser Index muß nicht derselbe wie der sein, der von der Eigenschaft *KeyField* verwendet wird (z.B. *CustomersSource*).
RowCount	Diese Eigenschaft gibt die Anzahl der Textzeilen an, die im Listenfeld angezeigt werden. Die Höhe des Listenfeldes wird so angepaßt, daß die Zeilen genau darin Platz finden.
DropDownRows	Die Eigenschaft legt die Anzahl der Textzeilen fest, die in der Dropdown-Liste angezeigt werden.

Inkrementelle Suche in Listen

Der Benutzer kann zur Laufzeit eine inkrementelle Suche nach Listenfeldeinträgen ausführen. Wenn das Steuerelement den Fokus besitzt, wird beispielsweise bei der Eingabe von ROB der erste Eintrag im Listenfeld ausgewählt, der mit den Buchstaben ROB beginnt. Wenn Sie ein weiteres E eingeben, wird der erste Eintrag ausgewählt, der mit den Buchstaben ROBE beginnt. Bei der Suche wird die Groß-/Kleinschreibung berücksichtigt. Mit den Tasten *Rück* und *Esc* wird der aktuelle Such-String verworfen (wobei die Auswahl aber bestehen bleibt). Dieselbe Wirkung hat eine Pause von zwei Sekunden zwischen Tastenanschlägen.

Boolesche Feldwerte und Kontrollfelder

TDBCheckBox ist eine datensensitive Version der Komponente *TCheckBox*. Sie kann für die Anzeige und Bearbeitung der Werte in Booleschen Feldern einer Datenmenge herangezogen werden. Beispielsweise könnte ein Rechnungsformular ein Kontrollfeld enthalten, das im aktivierten Zustand signalisiert, daß der Kunde umsatzsteuerpflichtig ist, während der deaktivierte Zustand bedeutet, daß dies nicht der Fall ist.

Datensensitive Kontrollfelder steuern selbst, ob sie aktiviert oder deaktiviert sind. Dazu wird der Wert des aktuellen Feldes mit den Werten der Eigenschaften *ValueChecked* und *ValueUnChecked* verglichen. Bei diesem Vergleich gibt es immer nur eine Übereinstimmung. Stimmt der Wert von *ValueChecked* mit dem Wert des Feldes überein, aktiviert Delphi das Kontrollfeld. Wenn dagegen der Wert der Eigenschaft *ValueUnchecked* mit dem Wert des Feldes übereinstimmt, deaktiviert Delphi das Kontrollfeld.

Hinweis Die Werte von *ValueChecked* und *ValueUnchecked* dürfen nicht identisch sein.

Belegen Sie die Eigenschaft *ValueChecked* mit einem Wert, der in die Datenbank eingetragen werden soll, wenn das Steuerelement beim Wechsel des Benutzers zu einem anderen Datensatz aktiviert ist. Dieser Wert ist per Voreinstellung *True*. Sie können aber jeden beliebigen alphanumerischen Wert angeben. Es ist auch möglich, eine Liste mit Einträgen, die durch Semikolons voneinander getrennt sind, als Wert für *ValueChecked* einzugeben. Wenn einer der Einträge mit dem Inhalt des Feldes im aktuellen Datensatz übereinstimmt, wird das Kontrollfeld aktiviert. So ließe sich beispielsweise der folgende String für *ValueChecked* verwenden:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

Enthält das Feld des aktuellen Datensatzes einen der Werte *Wahr, Ja* oder *Ein*, wird das Kontrollfeld aktiviert. Beim Vergleich des Feldes mit den Strings von *ValueChecked* wird die Groß-/Kleinschreibung berücksichtigt. Wenn ein Benutzer ein Kontrollfeld aktiviert, für das mehrere Strings definiert sind, wird der erste String als Wert in die Datenbank eingetragen.

Belegen Sie die Eigenschaft *ValueUnchecked* mit einem Wert, der in die Datenbank eingetragen werden soll, wenn das Steuerelement beim Wechsel des Benutzers zu einem anderen Datensatz nicht aktiviert ist. Dieser Wert ist per Voreinstellung auf *False* gesetzt. Sie können aber jeden beliebigen alphanumerischen Wert angeben. Es ist auch möglich, eine Liste mit Einträgen, die durch Semikolons voneinander getrennt sind, als Wert für *ValueUnchecked* einzugeben. Wenn einer der Einträge mit dem Inhalt des Feldes im aktuellen Datensatz übereinstimmt, wird das Kontrollfeld deaktiviert.

Ein datensensitives Kontrollfeld wird in dunklerem Grau dargestellt (nicht *True* und nicht *False*), wenn das Feld des aktuellen Datensatzes keinen der Werte enthält, die in den Eigenschaften *ValueChecked* oder *ValueUnchecked* aufgeführt sind.

Ist das Kontrollfeld mit einem logischen Feld verknüpft, dann ist es immer aktiviert, wenn der Inhalt des Feldes *True* ist, und immer deaktiviert, wenn der Inhalt des Feldes *False* ist. In diesem Fall haben die Strings, die für die Eigenschaften *ValueChecked* und *ValueUnchecked* festgelegt wurden, keinen Einfluß.

Feldwerte mit Optionsfeldern einschränken

TDBRadioGroup ist eine datensensitive Version der Komponente *TRadioGroup*, die es ermöglicht, den Wert eines Datenfeldes über ein Optionsfeld zuzuweisen, das nur eine begrenzte Anzahl von Feldwerten anbietet. Das Optionsfeld besteht aus einer Schaltfläche für jeden Wert, den ein Feld akzeptieren kann. Der Benutzer setzt den Wert für ein Datenfeld, indem er auf das gewünschte Optionsfeld klickt.

Die Eigenschaft *Items* legt die Anzahl der Optionsfelder fest, die in der Gruppe angezeigt werden. Diese Eigenschaft ist eine Stringliste. Für jeden String in *Items* wird ein Optionsfeld angezeigt. Die Strings selbst erscheinen rechts vom Optionsfeld als Beschriftung.

Wenn der aktuelle Wert eines Feldes, das mit einer Gruppe von Optionsfeldern verknüpft ist, mit einem der Strings in der Eigenschaft *Items* übereinstimmt, wird dieses Optionsfeld automatisch ausgewählt. Sind beispielsweise die drei Strings Rot, Gelb und Blau in der Eigenschaft *Items* enthalten und enthält das Feld des aktuellen Datensatzes den Wert Blau, wird das dritte Optionsfeld in der Gruppe ausgewählt.

Hinweis Stimmt das Feld mit keinem der Werte in *Items* überein, kann trotzdem ein Optionsfeld ausgewählt sein, wenn das Feld mit einem String in der Eigenschaft *Values* übereinstimmt. Stimmt das Feld des aktuellen Datensatzes weder mit einem String in *Items* noch in *Values* überein, ist kein Optionsfeld ausgewählt.

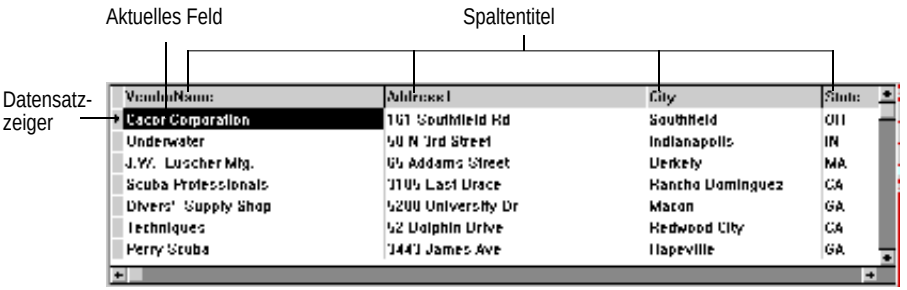
Die Eigenschaft *Values* kann eine optionale Liste mit Strings enthalten, die in die Datenmenge zurückgeschrieben werden, wenn der Benutzer ein Optionsfeld auswählt und damit einen Datensatz einträgt. Strings sind den Optionsfeldern in numerischer Reihenfolge zugeordnet. Der erste String ist mit dem ersten Optionsfeld verbunden, der zweite String mit dem zweiten Optionsfeld usw. Angenommen, *Items* enthält die Einträge Rot, Gelb und Blau, und *Values* die Einträge Magenta, Gelb und Zyan. Wenn der Benutzer nun das Optionsfeld mit der Beschriftung Rot auswählt, wird Magenta in die Datenbank eingetragen.

Falls in *Values* keine Strings vorhanden sind, wird der *Items*-String eines ausgewählten Optionsfeldes in die Datenbank geschrieben, sobald ein Datensatz eingetragen wird.

Daten mit TDBGrid anzeigen und bearbeiten

Mit einem *TDBGrid*-Steuerelement können Sie die Datensätze einer Datenmenge in einem tabellarischen Format anzeigen und bearbeiten.

Abbildung 26.1 TDBGrid-Steuerelement



Drei Faktoren beeinflussen das Erscheinungsbild von Datensätzen in einem Gitter-Steuerelement:

- Persistente Spaltenobjekte, die mit dem Spalteneditor für das Gitterobjekt definiert werden. Persistente Spaltenobjekte ermöglichen eine sehr flexible Steuerung des Erscheinungsbildes von Gitter und Daten.
- Persistente Feldkomponenten, die für die Datenmenge im Gitter angezeigt werden. Weitere Informationen zum Erstellen persistenter Feldkomponenten mit dem Feldeditor finden Sie in Kapitel 19, »Felder«.
- Die Einstellung der Eigenschaft *ObjectView* der Datenmenge für Gitter, die ADT- und Array-Felder anzeigen. Unter »ADT- und Array-Felder anzeigen« auf Seite 26-27 finden Sie weitere Informationen.

Gitter besitzen die Eigenschaft *Columns*, die ein *TDBGridColumns*-Objekt kapselt. *TDBGridColumns* ist eine Kollektion von *TColumn*-Objekten, die alle Spalten eines Gitters repräsentieren. Sie können den Spalteneditor verwenden, um die Spaltenattribute bereits beim Entwurf einzustellen. Mit der Eigenschaft *Columns* des Gitters können Sie dagegen zur Laufzeit auf die Eigenschaften, Ereignisse und Methoden von *TDBGridColumns* zugreifen.

Die Eigenschaft *State* der Eigenschaft *Columns* eines Gitters gibt an, ob persistente Spaltenobjekte für das Gitter existieren. *Columns.State* ist eine nur zur Laufzeit verfügbare Eigenschaft, die automatisch für ein Gitter eingestellt wird. Der Standardstatus *csDefault* bedeutet, daß keine persistenten Spaltenobjekte für das Gitter existieren. In diesem Fall ist die Anzeige der Daten im Gitter entweder von den persistenten Feldkomponenten der im Gitter angezeigten Datenmenge oder von Standard-Anzeigecharakteristika abhängig, wenn die Datenmenge keine persistente Feldkomponenten besitzt.

Gitter im Standardstatus verwenden

Wenn die Eigenschaft *Columns.State* eines Gitters *csDefault* enthält, bestimmen in erster Linie die Eigenschaften der Felder der Datenmenge des Gitters das Erscheinungsbild der Datensätze. Gitterspalten werden dynamisch aus den sichtbaren Feldern der Datenmenge generiert. Die Reihenfolge der Spalten im Gitter entspricht der Reihenfolge der Felder in der Datenmenge. Jede Gitterspalte ist einer Feldkomponente zugeordnet. Änderungen der Eigenschaften von Feldkomponenten werden sofort im Gitter angezeigt.

Der Einsatz eines Gitters mit dynamisch generierten Spalten ist für die Anzeige und Bearbeitung von Tabellen hilfreich, die erst zur Laufzeit ausgewählt werden. Da die Struktur des Gitters nicht festgelegt ist, kann sie dynamisch an verschiedene Datenmengen angepaßt werden. Ein einzelnes Gitter mit dynamisch generierten Spalten kann beispielsweise erst eine Paradox-Tabelle anzeigen und dann zur Anzeige der Ergebnisse einer SQL-Abfrage verwendet werden, indem die Eigenschaft *DataSource* des Gitters oder die Eigenschaft *DataSet* der Datenquelle geändert wird.

Sie können das Erscheinungsbild einer dynamischen Spalte beim Entwurf oder zur Laufzeit ändern. Dies erfolgt jedoch durch Änderung der Eigenschaften der Feldkomponente, die in dieser Spalte angezeigt wird. Eigenschaften einer dynamischen Spalte bleiben verfügbar, bis die Zuordnung einer Spalte zu einem bestimmten Feld in einer einzelnen Datenmenge aufgehoben wird. Wird beispielsweise die Eigenschaft *Width* einer Spalte geändert, ändert sich damit die Eigenschaft *DisplayWidth* des Feldes, das mit der betreffenden Spalte verbunden ist. Änderungen an Spalteneigenschaften, die nicht auf Feldeigenschaften beruhen, wie etwa *Font*, haben nur die Lebensdauer der zugrundeliegenden Spalte.

Eigenschaften dynamischer Spalten bleiben erhalten, bis die zugeordneten Feldkomponenten freigegeben werden. Wenn die Datenmenge eines Gitters aus dynamischen Feldkomponenten besteht, werden die Felder freigegeben, sobald die Datenmenge geschlossen wird. Beim Freigeben der Feldkomponenten werden auch alle zugeordneten dynamischen Spalten freigegeben. Wenn die Datenmenge eines Gitters aus persistenten Feldkomponenten besteht, bleiben diese Feldkomponenten beim Schließen der Datenmenge erhalten. Dies gilt auch für die Eigenschaften der Spalten, die diesen Feldern zugeordnet sind.

Hinweis Wird der Eigenschaft *Columns.State* eines Gitters zur Laufzeit *csDefault* zugewiesen, werden alle Spaltenobjekte im Gitter gelöscht (auch persistente Spalten). Anschließend werden die dynamischen Spalten basierend auf den sichtbaren Feldern der Datenmenge des Gitters neu erstellt.

Angepaßte Gitter erstellen

Ein benutzerdefiniertes Gitter besteht aus persistenten Spaltenobjekten. Diese beschreiben das Erscheinungsbild der Spalten und der Daten in diesen Spalten. Mit benutzerdefinierten Gittern können Sie mehrere Gitter zur Darstellung unterschiedlicher Ansichten derselben Datenmenge konfigurieren (unterschiedliche Spaltenreihenfolge, unterschiedliche Felder, unterschiedliche Farben und Schriften usw.). Außerdem können Sie es Benutzern ermöglichen, das Erscheinungsbild des Gitters

zur Laufzeit zu ändern, ohne die zugrundeliegenden Felder des Gitters oder die Feldreihenfolge der Datenmenge zu beeinflussen.

Benutzerdefinierte Gitter werden insbesondere für Datenmengen eingesetzt, deren Struktur während des Entwurfs bereits bekannt ist. Da vorausgesetzt wird, daß die beim Entwurf angegebenen Feldnamen auch in der Datenmenge existieren, sind benutzerdefinierte Gitter nicht geeignet, wenn Benutzer zur Laufzeit beliebige Tabellen auswählen können.

Persistente Spalten – Grundlagen

Wenn Sie persistente Spaltenobjekte für ein Gitter erstellen, sind diese nur lose mit den zugrundeliegenden Feldern der Datenmenge eines Gitters verknüpft. Die Standard-Eigenschaftswerte für persistente Spalten werden dynamisch aus einer Standardquelle (beispielsweise einem Gitter oder einem zugeordneten Feld) abgerufen, bis der jeweiligen Spalteneigenschaft explizit ein Wert zugewiesen wird. Bis zu dieser expliziten Zuweisung ändert sich der Wert von Spalteneigenschaften nur beim Wechseln der Standardquelle.

So ist beispielsweise die Standardquelle für eine Spaltenüberschrift die Eigenschaft *DisplayLabel* des zugeordneten Feldes. Wenn Sie die Eigenschaft *DisplayLabel* ändern, wird die Änderung sofort im Spaltentitel sichtbar.

Sobald Sie einer Spalteneigenschaft explizit einen Wert zuweisen, wird dieser beim Wechseln der Standardquelle nicht mehr automatisch geändert. Wenn Sie also der Spaltenüberschrift einen String zuweisen, ist die Überschrift von der Eigenschaft *DisplayLabel* des zugeordneten Feldes unabhängig. Spätere Änderungen der Eigenschaft *DisplayLabel* dieses Feldes wirken sich nicht auf den Spaltentitel aus.

Da persistente Spalten unabhängig von den Feldkomponenten existieren, denen sie zugeordnet sind, müssen sie keinen Feldobjekten zugeordnet werden. Wenn die Eigenschaft *FieldName* einer persistenten Spalte leer ist oder der Feldname keinem Feld in der aktuellen Datenmenge des Gitters entspricht, enthält die Eigenschaft *Field* der Spalte den Wert Null. Die Spalte wird in diesem Fall mit leeren Zellen angezeigt. Sie können eine leere Spalte zur Anzeige von Bitmaps oder Balkendiagrammen verwenden, die bestimmte Aspekte eines Datensatzes in einer andernfalls leeren Zelle grafisch hervorheben. Zu diesem Zweck müssen Sie die Standard-Zeichenmethode der Zelle überschreiben.

Einem Feld in einer Datenmenge können mehrere persistente Spalten zugeordnet werden. Sie können beispielsweise auf der linken und auf der rechten Seite eines sehr breiten Gitters dieselbe Teilenummer anzeigen, um die Suche nach bestimmten Teilen zu erleichtern.

Hinweis Da persistente Spalten keinem Feld in einer Datenmenge zugeordnet werden müssen und mehrere Spalten dasselbe Feld referenzieren können, kann die Eigenschaft *FieldCount* eines benutzerdefinierten Gitters kleiner oder gleich der Spaltenanzahl im Gitter sein. Beachten Sie außerdem, daß die Eigenschaft *SelectedField* den Wert Null und die Eigenschaft *SelectedIndex* eines benutzerdefinierten Gitters den Wert –1 hat, wenn die aktuell im Gitter ausgewählte Spalte keinem Feld zugeordnet ist.

Persistente Spalten können zur Anzeige der Zellen einer Spalte in einem Kombinationsfeld mit Lookup-Werten aus einer anderen Datenmenge oder einer statischen

Auswahlliste bzw. als Ellipsen-Schaltfläche (…) in einer Zelle konfiguriert werden, die spezielle Viewer oder Dialogfelder für die aktuelle Zelle aufruft.

Quelle einer Spalteneigenschaft zur Laufzeit festlegen

Zur Laufzeit können Sie die Eigenschaft *AssignedValues* einer Spalte abfragen, um festzustellen, ob eine Spalteneigenschaft Werte aus der zugeordneten Feldkomponente erhält oder ihr ein eigener Wert zugewiesen wurde.

Sie können alle Standardeigenschaften einer Spalte zurücksetzen, indem Sie die Methode *RestoreDefaults* der Spalte aufrufen. Wollen Sie die Standardeigenschaften aller Spalten in einem Gitter wiederherstellen, rufen Sie die Methode *RestoreDefaults* der Spaltenliste auf:

```
DBGrid1.Columns.RestoreDefaults;
```

Mit der Methode *Add* der Spaltenliste fügen Sie eine persistente Spalte ein:

```
DBGrid1.Columns.Add;
```

Eine persistente Spalte wird durch Freigabe des Spaltenobjekts gelöscht:

```
DBGrid1.Columns[5].Free;
```

Indem Sie der Eigenschaft *Column.State* eines Gitters zur Laufzeit *csCustomized* zuweisen, wird der benutzerdefinierte Modus des Gitters aktiviert. Alle im Gitter vorhandenen Spalten werden freigegeben, und für jedes Feld in der Datenmenge des Gitters werden persistente Spalten erstellt.

Persistente Spalten erstellen

Mit dem Spalteneditor können Sie persistente Spaltenobjekte für ein Gitter erstellen, um dessen Erscheinungsbild beim Entwurf festzulegen. Zur Laufzeit wird der Eigenschaft *State* eines Gitters mit persistenten Spaltenobjekten automatisch *csCustomized* zugewiesen.

So erstellen Sie persistente Spalten für ein Gitter:

1. Wählen Sie die Gitterkomponente im Formular aus.
2. Starten Sie den Spalteneditor, indem Sie im Objektinspektor auf die Eigenschaft *Columns* des Gitters doppelklicken.

Das Listenfeld *Spalten* zeigt die persistenten Spalten, die bereits für das ausgewählte Gitter definiert wurden. Beim ersten Starten des Spalteneditors ist diese Liste leer, da der Standardstatus des Gitters aktiviert ist. In diesem Status werden nur dynamische Spalten angezeigt.

Sie können in einem Arbeitsgang für alle Felder einer Datenmenge persistente Spalten erstellen oder dies für einzelne Felder durchführen. Folgendermaßen erstellen Sie persistente Spalten für alle Felder:

1. Klicken Sie mit der rechten Maustaste auf das Gitter, und wählen Sie im lokalen Menü die Option *Alle Felder hinzufügen*. Wenn das Gitter noch keiner Datenquelle zugeordnet wurde, ist diese Option deaktiviert. Ordnen Sie dem Gitter deshalb zunächst eine aktive Datenmenge zu.

2. Wenn das Gitter bereits persistente Spalten enthält, müssen Sie in einem Dialogfeld angeben, ob die vorhandenen Spalten gelöscht oder neue Spalten hinzugefügt werden sollen. Wählen Sie *Ja*, werden alle vorhandenen Informationen zu persistenten Feldern gelöscht. Anschließend werden alle Felder in der aktuellen Datenmenge mit ihren Feldnamen und nach Maßgabe der Reihenfolge in der Datenmenge hinzugefügt. Wählen Sie dagegen *Nein*, bleiben die vorhandenen Definitionen persistenter Spalten erhalten. Die neuen Spalteninformationen werden basierend auf den zusätzlichen Feldern in der Datenmenge angefügt.
3. Klicken Sie auf *Schließen*, um die persistenten Spalten dem Gitter zuzuweisen und das Dialogfeld zu schließen.

Folgendermaßen erstellen Sie einzelne persistente Spalten:

1. Wählen Sie im Spalteneditor die Schaltfläche *Hinzufügen*. Die neue Spalte wird im Listenfeld ausgewählt. Sie erhält eine laufende Nummer und einen Standardnamen (beispielsweise 0 - TColumn).
2. Weisen Sie im Objektinspektor der Eigenschaft *FieldName* der neuen Spalte einen Wert zu, um dieser Spalte ein Feld zuzuordnen.
3. Mit der Option *Caption* der Eigenschaft *Title* können Sie im Objektinspektor einen Spaltentitel festlegen.
4. Schließen Sie den Spalteneditor, um die persistenten Spalten dem Gitter zuzuweisen.

Persistente Spalten löschen

Das Löschen einer persistenten Spalte aus einem Gitter ist hilfreich, um die Anzeige eines Feldes zu verhindern. Folgendermaßen löschen Sie eine persistente Spalte aus einem Gitter:

1. Wählen Sie das zu löschende Feld im Listenfeld *Spalten* aus.
2. Klicken Sie auf *Löschen* (Sie können zum Entfernen einer Spalte auch das lokale Menü oder die Taste *Entf* verwenden).

Hinweis Wenn Sie alle Spalten aus einem Gitter löschen, wird der Eigenschaft *Columns.State* des Gitters wieder der Wert *csDefault* zugewiesen. Für jedes Feld in der Datenmenge werden automatisch dynamische Spalten erstellt.

Reihenfolge persistenter Spalten ändern

Die Anzeigereihenfolge der Spalten im Spalteneditor entspricht der im Gitter. Sie können die Spaltenreihenfolge ändern, indem Sie die Spalten im Listenfeld *Spalten* an eine neue Position ziehen.

Folgendermaßen ändern Sie die Reihenfolge der Spalten:

1. Wählen Sie die Spalte im Listenfeld *Spalten* aus.
2. Ziehen Sie die Spalte im Listenfeld an die gewünschte Position.

Sie können die Spaltenreihenfolge auch durch Ziehen im Gitter ändern (wie zur Laufzeit).

Lookup-Spalte definieren

Damit eine Spalte eine Dropdown-Liste mit Werten aus einer separaten Lookup-Tabelle anzeigt, müssen Sie ein Lookup-Feldobjekt in der Datenmenge definieren. Weitere Informationen zum Erstellen von Lookup-Feldern finden Sie unter »Lookup-Felder definieren« auf Seite 19-11.

Nachdem das Lookup-Feld definiert wurde, weisen Sie der Eigenschaft *FieldName* den Namen des Lookup-Feldes zu. Außerdem müssen Sie der Eigenschaft *ButtonStyle* der Spalte den Wert *cbsAuto* zuweisen. Das Gitter zeigt automatisch eine Dropdown-Schaltfläche im Kombinationsfeldstil an, wenn sich eine Zelle der betreffenden Spalte im Bearbeitungsmodus befindet. Die Dropdown-Liste enthält Werte, die für das Lookup-Feld definiert wurden.

Auswahllistenspalte definieren

Eine Auswahllistenspalte entspricht in Aussehen und Funktion einer Lookup-Listenspalte. Das zugrundeliegende Feld dieser Spalte ist jedoch ein normales Feld, und die Werte in der Liste stammen aus der Eigenschaft *PickList* der Spalte, nicht aus einer Lookup-Tabelle.

Folgendermaßen definieren Sie eine Auswahllistenspalte:

1. Wählen Sie die Spalte im Listenfeld *Spalten* aus.
2. Weisen Sie *ButtonStyle* den Wert *cbsAuto* zu.
3. Klicken Sie im Objektinspektor doppelt auf die Eigenschaft *Picklist*, um den Stringlisten-Editor zu öffnen.

Geben Sie in diesem Editor die Werte ein, die in der Dropdown-Liste dieser Spalte angezeigt werden sollen. Sobald die Auswahlliste Daten enthält, wird aus der Spalte eine Auswahllistenspalte.

Hinweis Sie können die Standardfunktion einer Spalte wiederherstellen, indem Sie den Text aus der Auswahlliste löschen.

Schaltfläche in Spalte einfügen

Eine Spalte kann rechts neben dem normalen Zelleditor eine Ellipsen-Schaltfläche anzeigen. Mit der Tastenkombination *Strg+Return* oder einem Mausklick wird das Ereignis *OnEditButtonClick* des Gitters ausgelöst. Sie können die Ellipsen-Schaltfläche zum Anzeigen von Formularen verwenden, die detaillierte Ansichten der Daten in der Spalte anzeigen. Sie können beispielsweise eine Ellipsen-Schaltfläche in einer Spalte mit Rechnungssummen zur Anzeige eines Formulars konfigurieren, das die Einzelpositionen der jeweiligen Rechnung enthält oder die enthaltene Umsatzsteuer anzeigt. Für grafische Felder können Sie ein Formular aufrufen, das die Grafik darstellt.

Folgendermaßen erstellen Sie eine Ellipsen-Schaltfläche in einer Spalte:

1. Wählen Sie die Spalte im Listenfeld *Spalten* aus.
2. Weisen Sie der Eigenschaft *ButtonStyle* den Wert *cbsEllipsis* zu.
3. Schreiben Sie eine Ereignisbehandlungsroutine für *OnEditButtonClick*.

Spalteneigenschaften beim Entwurf einstellen

Die Spalteneigenschaften bestimmen, wie Daten in den Zellen einer Spalte angezeigt werden. Die meisten Spalteneigenschaften werden aus den Eigenschaften einer anderen Komponente (der sogenannten Standardquelle) übernommen. Dabei kann es sich beispielsweise um ein Gitter oder eine zugeordnete Feldkomponente handeln.

Wählen Sie eine Spalte im Spalteneditor aus, um deren Eigenschaften im Objektinspektor anzuzeigen. Die folgende Tabelle beschreibt die wichtigsten Spalteneigenschaften.

Tabelle 26.5 Spalteneigenschaften

Eigenschaft	Beschreibung
Alignment	Richtet die Felddaten in der Zelle am linken Rand, am rechten Rand oder zentriert aus. Standardquelle: *TField.Alignment*.
ButtonStyle	*cbsAuto* (Standard): Zeigt eine Dropdown-Liste an, wenn das zugeordnete Feld ein Lookup-Feld ist oder die Eigenschaft *PickList* der Spalte Daten enthält. *cbsEllipsis:* Zeigt rechts von der Zelle eine Ellipsen-Schaltfläche an. Durch Klicken auf diese Schaltfläche wird das Ereignis *OnEditButtonClick* ausgelöst. *cbsNone:* Die Spalte verwendet nur das normale Steuerelement zur Bearbeitung der Daten in der Spalte.
Color	Gibt die Hintergrundfarbe für die Zellen der Spalte an. Die Farbe des Textes wird mit der Eigenschaft *Font* eingestellt. Standardquelle: *TDBGrid.Color*.
DropDownRows	Legt die Anzahl der Textzeilen fest, die in der Dropdown-Liste angezeigt werden. Standard: 7.
Expanded	Gibt an, ob die Spalte erweitert ist. Diese Eigenschaft gilt nur für die Darstellung von ADT- oder Array-Feldern.
FieldName	Gibt den Namen des Feldes an, das dieser Spalte zugeordnet ist. Diese Eigenschaft kann leer sein.
ReadOnly	*True:* Die Daten in dieser Spalte können vom Benutzer nicht bearbeitet werden. *False* (Standard): Die Daten in dieser Spalte können bearbeitet werden.
Width	Gibt die Breite der Spalte in Bildschirm-Pixel an. Standardquelle: *TField.DisplayWidth*.
Font	Gibt Schriftart, Schriftgröße und Schriftfarbe an. Standardquelle: *TDBGrid.Font*.
PickList	Enthält eine Liste der Werte, die in der Spalte in einer Dropdown-Liste angezeigt werden sollen.
Title	Gibt die Eigenschaften für den Titel der ausgewählten Spalte an.

Die folgende Tabelle enthält die Optionen der Eigenschaft *Title*.

Tabelle 26.6 Optionen der Eigenschaft TColumn.Title

Eigenschaft	Beschreibung
Alignment	Richtet den Text im Tabellenkopf am linken Rand (Standard), am rechten Rand oder zentriert aus.
Caption	Gibt den Text an, der im Tabellenkopf angezeigt werden soll. Standardquelle: *TField.DisplayLabel*.
Color	Gibt die Hintergrundfarbe für die Zelle mit dem Tabellenkopf an. Standardquelle: *TDBGrid.FixedColor*.
Font	Gibt Schriftart, Schriftgröße und Schriftfarbe an. Standardquelle: *TDBGrid.TitleFont*.

Standardwerte einer Spalte wiederherstellen

Sie können die Änderungen der Eigenschaften einer oder mehrerer Spalten rückgängig machen. Wählen Sie im Spalteneditor die betreffenden Spalten aus, und wählen Sie im lokalen Menü die Option *Standard*. Durch diese Operation werden alle zugewiesenen Eigenschaftseinstellungen verworfen. Anschließend werden den Eigenschaften der Spalten wieder die aus den zugrundeliegenden Feldkomponenten abgeleiteten Werte zugewiesen.

ADT- und Array-Felder anzeigen

In Abhängigkeit vom Wert der Eigenschaft *ObjectView* einer Datenmenge zeigt ein Gitter ADT- und Array-Felder entweder im erweiterten Modus oder im Objektmodus an. Im Objektmodus kann das Feld erweitert oder verkleinert werden. Wenn *ObjectView* den Wert *True* enthält, können die Felder erweitert oder verkleinert werden. Beim Erweitern eines Feldes wird jedes untergeordnete Feld in einer eigenen Spalte mit einer Titelleiste angezeigt, die sich unter der Titelleiste des ADT- bzw. Array-Feldes befindet. Wenn Sie ein Feld verkleinern, wird nur eine Spalte mit einem String angezeigt. Dieser String kann nicht bearbeitet werden. Er gibt die untergeordneten Felder in einer Liste mit Kommas als Trennzeichen an. Eine Spalte kann erweitert oder verkleinert werden, indem Sie auf den Pfeil in der Titelleiste des Feldes klikken. Außerdem muß die Eigenschaft *Expanded* der Spalte gesetzt werden. Wenn die Eigenschaft *ObjectView* der Datenmenge *False* enthält, wird jedes untergeordnete Feld in einer separaten Spalte angezeigt.

Tabelle 26.7 Eigenschaften zur Steuerung der Anzeige von ADT- und Array-Feldern in einem TDBGrid-Objekt

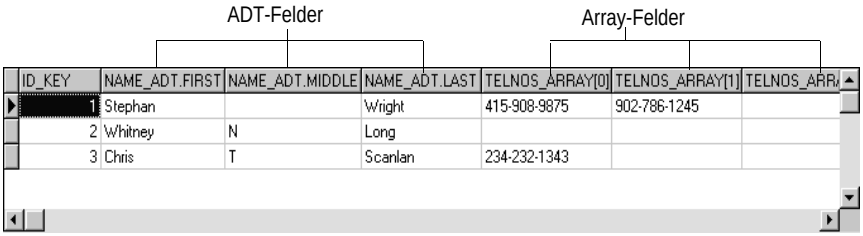
Eigenschaft	Objekt	Beschreibung
Expandable	*TColumn*	Gibt an, ob eine Spalte erweitert werden kann, damit die untergeordneten Felder in separaten Spalten angezeigt und bearbeitet werden können.
Expanded	*TColumn*	Gibt an, ob die Spalte erweitert ist.
MaxTitleRows	*TDBGrid*	Gibt die maximale Anzahl der Titelzeilen im Gitter an.

Tabelle 26.7 Eigenschaften zur Steuerung der Anzeige von ADT- und Array-Feldern in einem TDBGrid-Objekt (Fortsetzung)

Eigenschaft	Objekt	Beschreibung
ObjectView	*TDataSet*	Gibt an, ob Felder erweitert oder im Objektmodus angezeigt werden. Im Objektmodus kann jedes Objektfeld erweitert oder verkleinert werden.
ParentColumn	*TColumn*	Referenziert das *TColumn*-Objekt, dem die Spalte mit dem untergeordneten Feld gehört.

Abbildung 26.2 zeigt ein Gitter mit einem ADT-Feld und einem Array-Feld. Der Eigenschaft *ObjectView* der Datenmenge wurde *False* zugewiesen, damit jedes untergeordnete Feld in einer separaten Spalte angezeigt wird.

Abbildung 26.2 TDBGrid-Steuerelement mit dem Wert *False* für *ObjectView*



Die folgenden Abbildungen zeigen das Gitter mit einem ADT-Feld und einem Array-Feld. In der ersten Abbildung sind die Felder nicht erweitert und können nicht bearbeitet werden. Die zweite Abbildung zeigt dagegen die erweiterten Felder. Die Felder können durch Klicken auf den Pfeil in der Titelleiste des jeweiligen Feldes erweitert oder verkleinert werden.

Abbildung 26.3 TDBGrid-Steuerelement mit dem Wert *False* für *Expanded*

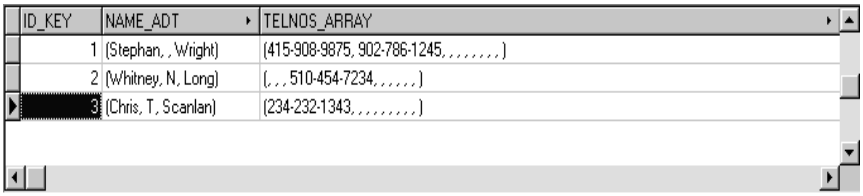
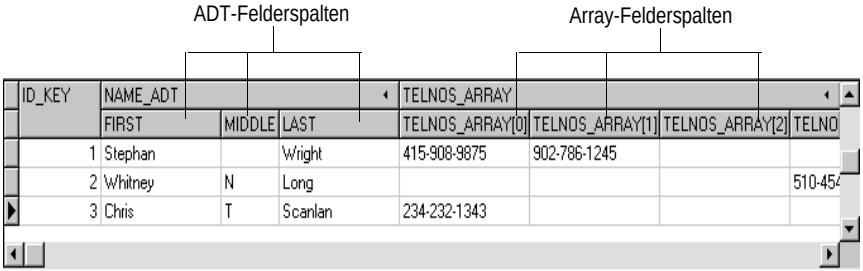


Abbildung 26.4 TDBGrid-Steuerelement mit dem Wert *True* für *Expanded*



Gitteroptionen einstellen

Sie können die Eigenschaft *Options* beim Entwurf verwenden, um Verhalten und Erscheinungsbild des Gitters zur Laufzeit festzulegen. Wenn Sie beim Entwurf eine Gitterkomponente in ein Formular einfügen, wird die Eigenschaft *Options* im Objektinspektor mit einem Plus-Zeichen (+) angezeigt, um die Erweiterungsmöglichkeit darzustellen. Die Eigenschaft *Options* enthält eine Reihe Boolescher Werte, die einzeln eingestellt werden können.

Doppelklicken Sie auf die Eigenschaft *Options*, um diese Eigenschaften anzuzeigen und einzustellen. Die Liste der einstellbaren Optionen wird im Objektinspektor unter der Eigenschaft *Options* angezeigt. Das Plus-Zeichen wird zu einem Minus-Zeichen, die Liste kann also durch einen Doppelklick auf die Eigenschaft *Options* wieder ausgeblendet werden.

Die folgende Tabelle führt die Eigenschaften unter *Options* auf, die eingestellt werden können. Außerdem werden die Auswirkungen auf das Gitter zur Laufzeit beschrieben.

Tabelle 26.8 Optionen der Eigenschaft TDBGrid

Option	Beschreibung
dgEditing	*True* (Standard): Aktiviert das Bearbeiten, Einfügen und Löschen von Datensätzen im Gitter. *False*: Deaktiviert das Bearbeiten, Einfügen und Löschen von Datensätzen im Gitter.
dgAlwaysShowEditor	*True*: Wenn ein Feld ausgewählt wird, befindet es sich im Bearbeitungsmodus. *False* (Standard): Der Bearbeitungsmodus wird beim Auswählen eines Feldes nicht automatisch aktiviert.
dgTitles	*True* (Standard): Zeigt Feldnamen oben im Gitter an. *False*: Die Anzeige der Feldnamen ist deaktiviert.
dgIndicator	*True* (Standard): Die Spalte für den Datensatzzeiger wird auf der linken Seite des Gitters angezeigt, und der Datensatzzeiger (ein Pfeil auf der linken Seite des Gitters) wird aktiviert. Beim Einfügen wird der Pfeil als Stern angezeigt, beim Bearbeiten als I-Cursor. *False*: Die Spalte für den Datensatzzeiger ist deaktiviert.
dgColumnResize	*True* (Standard): Spalten können durch Ziehen der Spaltentrenner im Tabellenkopfbereich vergrößert bzw. verkleinert werden. Die Größenänderung wirkt sich auch auf die zugrundeliegende TField-Komponente aus. *False*: Die Größe der Spalten im Gitter kann nicht geändert werden.
dgColLines	*True* (Standard): Zeigt vertikale Trennlinien zwischen den Spalten an. *False*: Zeigt keine vertikalen Trennlinien zwischen den Spalten an.
dgRowLines	*True* (Standard): Zeigt horizontale Trennlinien zwischen den Datensätzen an. *False*: Zeigt keine horizontalen Trennlinien zwischen den Datensätzen an.
dgTabs	*True* (Standard): Aktiviert den Wechsel der Felder in einem Datensatz mit der Tabulatortaste. *False*: Mit der Tabulatortaste wird das Gitter-Steuerelement verlassen.

Tabelle 26.8 Optionen der Eigenschaft TDBGrid (Fortsetzung)

Option	Beschreibung
dgRowSelect	*True*: Die Auswahlleiste erstreckt sich über die gesamte Breite des Gitters. *False* (Standard): Bei der Auswahl eines Feldes in einem Datensatz wird nur dieses Feld ausgewählt.
dgAlwaysShowSelection	*True* (Standard): Die Auswahlleiste ist im Gitter immer sichtbar, auch wenn ein anderes Steuerelement den Fokus besitzt. *False*: Die Auswahlleiste ist im Gitter nur sichtbar, wenn kein anderes Steuerelement den Fokus besitzt.
dgConfirmDelete	*True* (Standard): Beim Löschen von Datensätzen (*Strg+Entf*) wird eine Bestätigung angefordert. *False*: Datensätze werden ohne Bestätigung gelöscht.
dgCancelOnExit	*True* (Standard): Eine durchzuführende Einfügung wird verworfen, wenn der Fokus an ein anderes Steuerelement übergeben wird. Diese Option verhindert das versehentliche Eintragen nur teilweise ausgefüllter oder leerer Datensätze. *False*: Die durchzuführende Eintragung wird abgeschlossen.
dgMultiSelect	*True*: Der Benutzer kann nicht aufeinanderfolgende Zeilen im Gitter mit den Tasten *Strg+Umschalt* oder *Umschalt+Pfeil* auswählen. *False* (Standard): Das Auswählen nicht aufeinanderfolgender Zeilen ist nicht möglich.

Daten im Gitter bearbeiten

Zur Laufzeit können Sie in einem Gitter vorhandene Daten bearbeiten und neue Datensätze einfügen, wenn die folgenden Standardbedingungen erfüllt sind:

- Die Eigenschaft *CanModify* der *Dataset*-Komponente ist *True*.
- Die Eigenschaft *ReadOnly* des Gitters ist *False*.

Wenn ein Benutzer einen Datensatz im Gitter bearbeitet, werden die Änderungen in den einzelnen Feldern in einen internen Datensatzpuffer eingetragen. Erst wenn der Benutzer den Datensatzzeiger verschiebt, werden die Änderungen in die zugrundeliegende Datenmenge geschrieben. Auch wenn der Fokus an ein anderes Steuerelement im Formular übergeben wird, werden die Änderungen erst eingetragen, wenn im Gitter ein anderer Datensatz aktiviert wird. Beim Eintragen eines Datensatzes prüft die Datenmenge alle zugeordneten datensensitiven Komponenten auf Statusänderungen. Tritt beim Aktualisieren der Felder mit geänderten Daten ein Problem auf, löst Delphi eine Exception aus und ändert den Datensatz nicht.

Sie können alle Änderungen an einem Datensatz verwerfen, indem Sie in einem beliebigen Datensatz *Esc* drücken, bevor der Datensatzzeiger verschoben wird.

Spaltenreihenfolge beim Entwurf ändern

In Gittern mit persistenten Spalten und in Standardgittern, deren Datenmengen persistente Felder enthalten, können Sie die Gitterspalten neu anordnen, indem Sie auf die Titelzelle einer Spalte klicken und diese an die gewünschte Position im Gitter ziehen.

Hinweis Durch die Änderung der Reihenfolge persistenter Felder im Feldeditor wird auch die Reihenfolge der Spalten in einem Standardgitter geändert. Für benutzerdefinierte Gitter gilt dies jedoch nicht.

Wichtig Sie können die Reihenfolge der Spalten in einem Gitter mit dynamischen Spalten und dynamischen Feldern beim Entwurf ändern, da keine Informationen zur geänderten Feld- bzw. Spaltenreihenfolge aufgezeichnet werden müssen.

Spaltenreihenfolge zur Laufzeit ändern

Zur Laufzeit kann ein Benutzer die Maus verwenden, um eine Spalte an eine neue Position im Gitter zu ziehen, wenn die Eigenschaft *DragMode* den Wert *dmManual* enthält. Bei Änderung der Spaltenreihenfolge in einem Gitter mit dem Wert *csDefault* in der Eigenschaft *State* wird auch die Reihenfolge der Feldkomponenten in der Datenmenge geändert, die dem Gitter zugrundeliegt. Die Reihenfolge der Felder in der physikalischen Tabelle wird dagegen nicht geändert.

Das Ereignis *OnColumnMoved* des Gitters wird ausgelöst, nachdem eine Spalte verschoben wurde.

Sie können die Änderung der Spaltenreihenfolge zur Laufzeit durch den Benutzer verhindern, indem Sie der Eigenschaft *DragMode* des Gitters *dmAutomatic* zuweisen.

Gitterdarstellung steuern

Die erste Möglichkeit zur Beeinflussung der Darstellung eines Gitters besteht im Einstellen der Spalteneigenschaften. Das Gitter verwendet automatisch die Werte der Eigenschaften für Schrift, Farbe und Ausrichtung einer Spalte, um die Zellen dieser Spalte anzuzeigen. Der Text der Datenfelder wird basierend auf den Eigenschaften *DisplayFormat* oder *EditFormat* der Feldkomponente dargestellt, die der betreffenden Spalte zugeordnet ist.

Sie können die Standardanzeige eines Gitters mit Quelltext für das Ereignis *OnDrawColumnCell* implementieren. Wenn die Eigenschaft *DefaultDrawing* des Gitters *True* enthält, werden die normalen Zeichenoperationen durchgeführt, bevor die Ereignisbehandlungsroutine für *OnDrawColumnCell* aufgerufen wird. Der Quelltext kann dann über der Standardanzeige zeichnen. Dies ist insbesondere hilfreich, wenn Sie eine leere persistente Spalte definiert haben und spezielle Grafiken in den Zellen dieser Spalte angezeigt werden sollen.

Wollen Sie die Zeichenlogik des Gitters vollständig austauschen, weisen Sie *DefaultDrawing* den Wert *False* zu und fügen den Quelltext zum Zeichnen des Gitters in die Ereignisbehandlungsroutine für dessen Ereignis *OnDrawColumnCell* ein. Wollen Sie die Zeichenlogik dagegen nur für bestimmte Spalten oder Feld-Datentypen ersetzen,

können Sie *DefaultDrawColumnCell* in der Ereignisbehandlungsroutine für *OnDrawColumnCell* aufrufen, damit das Gitter für ausgewählte Spalten die normalen Zeichenoperationen ausführt. Dadurch wird der Arbeitsumfang reduziert, wenn Sie beispielsweise nur die Darstellung Boolescher Werte ändern wollen.

Zur Laufzeit auf Benutzeraktionen reagieren

Sie können das Verhalten eines Gitters ändern, indem Sie Ereignisbehandlungsroutinen schreiben, die zur Laufzeit auf bestimmte Aktionen im Gitter reagieren. Da ein Gitter normalerweise mehrere Felder und Datensätze gleichzeitig anzeigt, können sehr spezielle Anforderungen hinsichtlich der Änderungen in einzelnen Spalten vorliegen. So kann es erforderlich sein, eine Schaltfläche an einer beliebigen Position im Formular zu aktivieren bzw. zu deaktivieren, wenn der Benutzer den Fokus einer bestimmten Spalte zuordnet bzw. dieser entzieht.

Die folgende Tabelle führt die Gitterereignisse auf, die im Objektinspektor verfügbar sind.

Tabelle 26.9 Ereignisse für Gitter-Steuerelemente

Ereignis	Beschreibung
OnCellClick	Tritt auf, wenn der Benutzer auf eine Zelle im Gitter klickt.
OnColEnter	Tritt auf, wenn der Benutzer den Cursor in eine Spalte im Gitter verschiebt.
OnColExit	Tritt auf, wenn der Benutzer eine Spalte im Gitter verläßt.
OnColumnMoved	Tritt auf, wenn der Benutzer eine Spalte an eine neue Position verschiebt.
OnDblClick	Tritt auf, wenn der Benutzer im Gitter doppelklickt.
OnDragDrop	Tritt auf, wenn der Benutzer Komponenten im Gitter zieht und ablegt.
OnDragOver	Tritt auf, wenn der Benutzer mit der Maus über das Gitter zieht.
OnDrawColumnCell	Tritt auf, wenn eine Anwendung einzelne Zellen zeichnen muß.
OnDrawDataCell	(Veraltet) Tritt auf, wenn eine Anwendung einzelne Zellen zeichnen muß und *State* den Wert *csDefault* enthält.
OnEditButtonClick	Tritt auf, wenn der Benutzer auf eine Ellipsen-Schaltfläche in einer Spalte klickt.
OnEndDrag	Tritt auf, wenn der Benutzer das Ziehen mit der Maus im Gitter beendet.
OnEnter	Tritt auf, wenn das Gitter den Fokus erhält.
OnExit	Tritt auf, wenn das Gitter den Fokus abgibt.
OnKeyDown	Tritt auf, wenn der Benutzer eine Taste oder eine Tastenkombination drückt, während das Gitter den Fokus besitzt.
OnKeyPress	Tritt auf, wenn der Benutzer eine einzelne alphanumerische Taste drückt, während das Gitter den Fokus besitzt.
OnKeyUp	Tritt auf, wenn der Benutzer eine Taste losläßt, während das Gitter den Fokus besitzt.
OnStartDrag	Tritt auf, wenn der Benutzer beginnt, mit der Maus über das Gitter zu ziehen.
OnTitleClick	Tritt auf, wenn der Benutzer auf einen Spaltentitel klickt.

Diese Ereignisse können zu unterschiedlichen Zwecken genutzt werden. Sie können beispielsweise eine Ereignisbehandlungsroutine für *OnDblClick* schreiben, die eine Liste zur Auswahl eines Wertes durch den Benutzer anzeigt. Der ausgewählte Wert kann dann in die Spalte eingetragen werden. Eine solche Ereignisbehandlungsroutine kann auf die Eigenschaft *SelectedField* zugreifen, um die aktuelle Zeile und Spalte zu ermitteln.

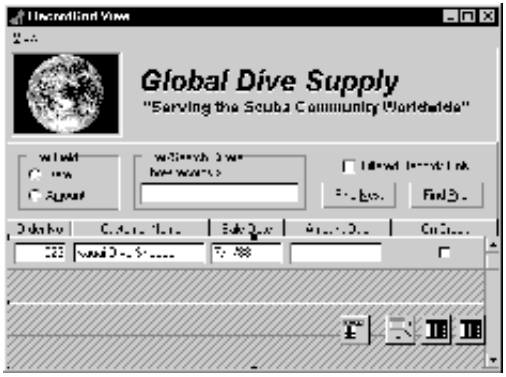
Gitter mit anderen datensensitiven Steuerelementen erstellen

Ein *TDBCtrlGrid*-Steuerelement zeigt mehrere Felder in mehreren Datensätzen in einem tabellarischen Gitterformat an. Jede Zelle in einem Gitter zeigt mehrere Felder aus einer einzelnen Zeile an. Sie können ein Datenbankgitter folgendermaßen verwenden:

1. Fügen Sie ein Datenbankgitter in ein Formular ein.
2. Weisen Sie der Eigenschaft *DataSource* des Gitters den Namen der Datenquelle zu.
3. Fügen Sie die verschiedenen datensensitiven Steuerelemente in die Entwurfszelle für das Gitter ein. Die Entwurfszelle des Gitters ist die Zelle links oben. Nur in diese Zelle können andere Steuerelemente eingefügt werden.
4. Weisen Sie der Eigenschaft *DataField* jedes datensensitiven Steuerelements den Namen eines Feldes zu. Als Datenquelle wird den Steuerelementen automatisch die Datenquelle des Datenbankgitters zugewiesen.
5. Ordnen Sie die Steuerelemente in der gewünschten Reihenfolge in der Zelle an.

Wenn Sie eine Anwendung mit einem Datenbankgitter compilieren und ausführen, wird die Anordnung der datensensitiven Steuerelemente in der Entwurfszelle zur Laufzeit in jeder Gitterzelle repliziert. Jede Zelle zeigt einen anderen Datensatz der Datenmenge an.

Abbildung 26.5 TDBCtrlGrid während des Entwurfs



Die folgende Tabelle führt einige der speziellen Eigenschaften von Datenbankgittern auf, die Sie beim Entwurf einstellen können:

Tabelle 26.10 Ausgewählte Eigenschaften für Datenbankgitter

Eigenschaft	Beschreibung
AllowDelete	*True* (Standard): Datensätze können gelöscht werden. *False*: Es können keine Datensätze gelöscht werden.
AllowInsert	*True* (Standard): Datensätze können eingefügt werden. *False*: Es können keine Datensätze eingefügt werden.
ColCount	Stellt die Anzahl der Spalten im Gitter ein. Standard = 1.
Orientation	*goVertical* (Standard): Zeigt die Datensätze von oben nach unten an. *goHorizontal:* Zeigt die Datensätze von links nach rechts an.
PanelHeight	Stellt die Höhe einer Tafel ein. Standard = 72.
PanelWidth	Stellt die Breite einer Tafel ein. Standard = 200.
RowCount	Stellt die Anzahl der anzuzeigenden Tafeln ein. Standard = 3.
ShowFocus	*True* (Standard): Zeigt die Tafel des aktuellen Datensatzes zur Laufzeit mit einem Fokus-Rechteck an. *False*: Zeigt kein Fokus-Rechteck an.

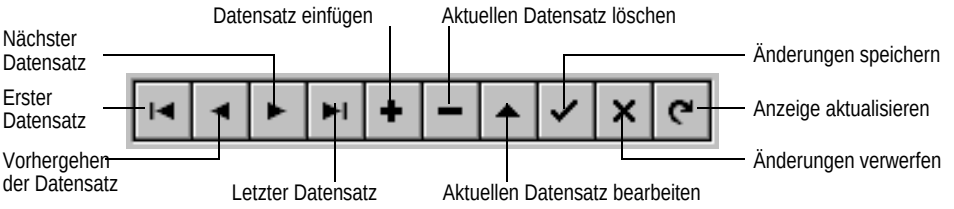
Weitere Informationen zu den Eigenschaften und Methoden von Datenbankgittern finden Sie in der VCL-Referenz.

Navigation und Bearbeitung von Datenmengen

Die Komponente *TDBNavigator* ist ein Steuerelement, mit dessen Hilfe sich der Benutzer durch die Daten einer Datenbanktabelle oder Abfrage bewegen kann. Der Navigator besteht aus einer Reihe von Schaltflächen für folgende Aktionen: Vorwärts und rückwärts durch die Datensätze blättern, den Cursor auf den ersten oder letzten Datensatz positionieren, einen neuen Datensatz einfügen, einen vorhandenen Datensatz aktualisieren, Änderungen an den Daten speichern oder verwerfen, einen Datensatz löschen und die Datensatzanzeige aktualisieren.

Die folgende Abbildung zeigt einen Navigator mit seinem voreingestellten Aussehen. Die Eigenschaft *VisibleButtons* des Navigators ermöglicht es, einzelne Schaltflächen dynamisch ein- und auszublenden.

Abbildung 26.6 Die Schaltflächen eines TDBNavigator-Objekts



In der folgenden Tabelle finden Sie eine Beschreibung der einzelnen Schaltflächen:

Tabelle 26.11 TDBNavigator-Schaltflächen

Schaltfläche	Beschreibung
Erster	Ruft die Methode *First* der Datenmenge auf und macht damit den ersten Datensatz der Datenmenge zum aktuellen Datensatz.
Vorheriger	Ruft die Methode *Prior* der Datenmenge auf und macht damit den vorherigen Datensatz zum aktuellen Datensatz.
Nächster	Ruft die Methode *Next* der Datenmenge auf und macht damit den nächsten Datensatz zum aktuellen Datensatz.
Letzter	Ruft die Methode *Last* der Datenmenge auf und macht damit den letzten Datensatz zum aktuellen Datensatz.
Einfügen	Ruft die Methode *Insert* der Datenmenge auf, die vor dem aktuellen Datensatz einen neuen Datensatz einfügt. Gleichzeitig wird die Datenmenge in den Einfüge- und in den Bearbeitungsmodus versetzt.
Löschen	Löscht den aktuellen Datensatz und macht den darauffolgenden zum aktuellen Datensatz. Wenn die Eigenschaft *ConfirmDelete True* ist, wird vor dem Löschen eine Bestätigung verlangt.
Bearbeiten	Versetzt die Datenmenge in den Bearbeitungsmodus und ermöglicht die Bearbeitung des aktuellen Datensatzes.
Speichern	Speichert die Änderungen am aktuellen Datensatz in der Datenbank.
Verwerfen	Verwirft die Änderungen am aktuellen Datensatz und zeigt ihn wieder in dem Zustand an, den er vor der Bearbeitung hatte. Einfüge- und Bearbeitungsmodus werden deaktiviert (falls sie aktiv sind).
Aktualisieren	Leert die Anzeigepuffer von datensensitiven Steuerelementen und aktualisiert sie mit den Daten aus der physikalischen Tabelle bzw. Abfrage. Dies ist erforderlich, wenn die Möglichkeit besteht, daß die Daten von einer anderen Anwendung geändert wurden.

Navigator-Schaltflächen ein- und ausblenden

Wenn eine *TDBNavigator*-Komponente zum ersten Mal in einem Formular plaziert wird, sind standardmäßig alle Schaltflächen sichtbar. Mit der Eigenschaft *VisibleButtons* können Sie einzelne Schaltflächen ein- und ausblenden. Beispielsweise wäre in einem Formular, das nicht für die Bearbeitung von Daten vorgesehen ist, das Ausblenden der Schaltflächen *Bearbeiten*, *Einfügen*, *Löschen*, *Speichern* und *Verwerfen* sinnvoll.

Navigator-Schaltflächen zur Entwurfszeit ein- und ausblenden

Im Objektinspektor sehen Sie neben der Eigenschaft *VisibleButtons*ein Pluszeichen (+). Es weist darauf hin, daß sich hinter dieser Eigenschaft für jede Schaltfläche des Navigators ein Boolescher Wert verbirgt. Um diese Werte anzuzeigen und zu setzen, doppelklicken Sie auf *VisibleButtons*. Die Liste der Schaltflächen, die ein- und ausgeblendet werden können, erscheint direkt unter dem Eigenschaftsnamen. Wenn die Liste geöffnet ist, wird neben *VisibleButtons* ein Minuszeichen (–) angezeigt, was bedeutet, daß die Liste durch einen erneuten Doppelklick auf den Eigenschaftsnamen wieder geschlossen werden kann.

Der Boolesche Wert zeigt an, ob die betreffende Schaltfläche ein- oder ausgeblendet ist. Bei *True* wird die Schaltfläche im *TDBNavigator*-Objekt angezeigt. Andernfalls ist sie weder zur Entwurfs- noch zur Laufzeit sichtbar.

Hinweis Wenn Sie einen Schaltflächenwert auf *False* setzen, wird die zugehörige Schaltfläche aus der *TDBNavigator*-Komponente entfernt. Die verbleibenden Schaltflächen nehmen an Breite zu, so daß weiterhin der gesamte Navigator ausgefüllt ist. Die Größe der Schaltflächen kann durch Ziehen der Griffe des Navigators mit der Maus geändert werden.

Weitere Informationen über die Schaltflächen und die zugehörigen Methoden finden Sie in der VCL-Referenz.

Navigator-Schaltflächen zur Laufzeit ein- und ausblenden

Zur Laufzeit können die Schaltflächen des Navigators als Reaktion auf Benutzereingaben oder in Abhängigkeit vom Status der Anwendung ein- und ausgeblendet werden. Angenommen, Sie haben einen Navigator für zwei verschiedene Datenmengen definiert. Eine dieser Datenmengen kann vom Benutzer bearbeitet werden, die andere nicht. Wenn der Benutzer in die schreibgeschützte Datenmenge wechselt, sollen die Schaltflächen *Einfügen*, *Löschen*, *Bearbeiten*, *Speichern*, *Verwerfen* und *Aktualisieren* des Navigators deaktiviert werden.

Angenommen, Sie möchten nicht nur die Datenquelle wechseln, sondern zugleich die Bearbeitung von *OrdersTable* verhindern (durch Ausblenden der Navigator-Schaltflächen *Einfügen*, *Löschen*, *Bearbeiten*, *Speichern*, *Verwerfen* und *Aktualisieren*) und die Bearbeitung von *CustomersTable* erlauben. Über die Eigenschaft *VisibleButtons* können Sie festlegen, welche Schaltflächen im Navigator angezeigt werden. Eine Möglichkeit zur Implementierung dieser Funktion wäre die folgende Ereignisbehandlungsroutine für *OnEnter*:

```
    procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
    if Sender = CustomerCompany then
    begin
     DBNavigatorAll.DataSource := CustomerCompany.DataSource;
     DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
    end
    else
    begin
     DBNavigatorAll.DataSource := OrderNum.DataSource;
     DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
       nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
    end;
end;
```

Hilfehinweise anzeigen

Für jede Navigator-Schaltfläche kann zur Laufzeit ein Hilfehinweis angezeigt werden. Dazu setzen Sie die Eigenschaft *ShowHint* des Navigators auf *True*. Wenn *ShowHint True* ist, wird für die Schaltfläche, über der sich der Mauszeiger befindet, ein Hinweistext angezeigt. Der Vorgabewert für *ShowHint* ist *False*.

Der Text der Hilfehinweise für die einzelnen Schaltflächen wird über die Eigenschaft *Hints* festgelegt. Die Vorgabe für *Hints* ist eine leere Stringliste. Wenn Sie diese Voreinstellung beibehalten, zeigt jede Navigator-Schaltfläche einen Standard-Hilfetext an. Mit Hilfe des Stringlisten-Editors können Sie eigene Hinweise verfassen, indem Sie der Eigenschaft für jede Schaltfläche einen einzeiligen Text zuweisen. Dieser Text setzt den Standardtext des Navigators außer Kraft.

Ein Navigator für mehrere Datenmengen

Wie bei allen datensensitiven Steuerelementen bezeichnet auch beim Navigator die Eigenschaft *DataSource* die Datenquelle, die das Steuerelement mit einer Datenmenge verbindet. Diese Eigenschaft kann zur Laufzeit geändert werden. Dadurch ist es möglich, mit einem einzigen Navigator die Datensatznavigation und -bearbeitung für mehrere Datenmengen zu implementieren.

Angenommen, ein Formular enthält zwei *DBEdit*-Steuerelemente, die über die Datenquellen *CustomersSource* und *OrdersSource* mit den Datenmengen *CustomersTable* bzw. *OrdersTable* verknüpft sind. Wenn nun ein Benutzer auf das mit *CustomersSource* verbundene *DBEdit*-Steuerelement klickt, soll der Navigator auf diese Datenmenge umschalten. Klickt der Benutzer dagegen auf das mit *OrdersSource* verbundene Steuerelement, soll der Navigator eben zu dieser Datenmenge wechseln. Sie können für eines der *DBEdit*-Steuerelemente eine Ereignisbehandlungsroutine für *OnEnter* schreiben und diese auch dem anderen Steuerelement zuweisen. Der entsprechende Quelltext könnte wie folgt aussehen:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
   if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
   else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

Kapitel

27

Entscheidungskomponenten

Die Gruppe der Entscheidungskomponenten dient dazu, Tabellen und Diagramme auf Basis von Daten zu erstellen, die im Kreuztabellenformat vorliegen. Diese Tabellen und Diagramme gestatten es, einen Informationsbestand nach verschiedenen Kriterien zusammenzufassen und unter verschiedenen Aspekten auszuwerten. Weitere Informationen hierzu finden Sie unter »Kreuztabellen« auf Seite 27-2

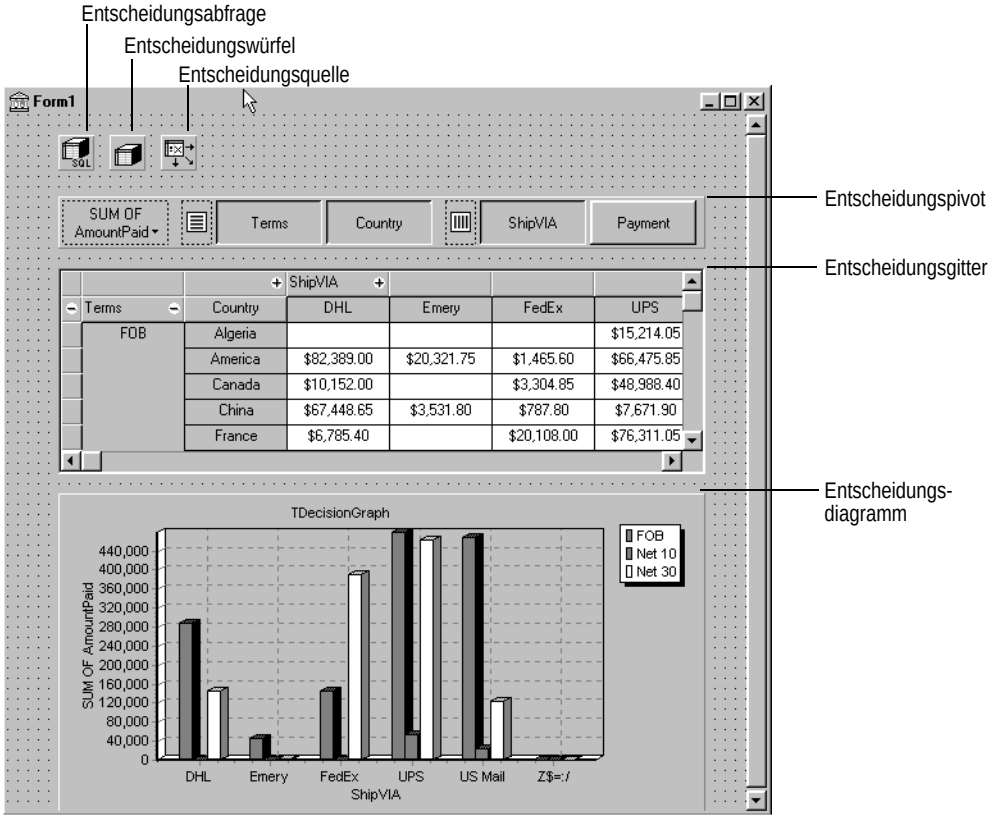
Überblick

In der Registerkarte *Datenanalyse* sind die folgenden Entscheidungskomponenten enthalten:

- Der Entscheidungswürfel *TDecisionCube* ist ein mehrdimensionaler Datenspeicher.
- Die Entscheidungsquelle *TDecisionSource* definiert den aktuellen Pivot-Status eines Entscheidungsgitters oder Entscheidungsgraphen.
- Die Entscheidungsabfrage *TDecisionQuery* ist eine spezialisierte Form von *TQuery* und definiert die Daten in einem Entscheidungswürfel.
- Mit dem Entscheidungspivot *TDecisionPivot* lassen sich die Dimensionen des Entscheidungswürfels (Felder) über Klicks auf Schaltflächen öffnen und schließen.
- Das Entscheidungsgitter *TDecisionGrid* zeigt ein- und mehrdimensionale Daten in Tabellenform an.
- Der Entscheidungsgraph *TDecisionGraph* übernimmt die grafische Aufbereitung der Felder aus einem Entscheidungsgitter.

Die folgende Abbildung 27.1 zeigt ein Formular, in dem alle diese Komponenten angeordnet sind.

Abbildung 27.1 Entscheidungskomponenten zur Entwurfszeit



Kreuztabellen

Kreuztabellen stellen eine Teilmenge des Datenbestandes so dar, daß Abhängigkeitsbeziehungen und Trends deutlicher zu erkennen sind. Tabellenfelder werden zu den Dimensionen der Kreuztabelle, während die Feldwerte die Kategorien und Zusammenfassungen innerhalb einer Dimension definieren.

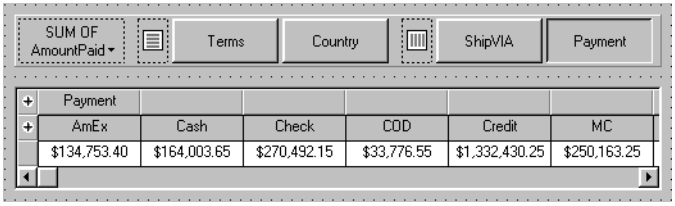
Mit Entscheidungskomponenten lassen sich Kreuztabellen in Formularen realisieren. *TDecisionGrid* bietet eine Darstellung in Tabellenform, und *TDecisionGraph* übernimmt die Diagrammdarstellung. *TDecisionPivot* verfügt über Schaltflächen, mit denen sich die Dimensionen ein- und ausblenden sowie zwischen Zeilen und Spalten verschieben lassen.

Kreuztabellen können ein- und mehrdimensional sein.

Eindimensionale Kreuztabellen

Eindimensionale Kreuztabellen enthalten eine zusammenfassende Auswertungsspalte oder -zeile für die Kategorien einer bestimmten Dimension. In der folgenden Abbildung ist »Payment« die ausgewählte Spaltendimension und »Amount Paid« die Zusammenfassungskategorie. Die Kreuztabelle zeigt die bezahlten Beträge für jede Zahlungsweise.

Abbildung 27.2 Eindimensionale Kreuztabelle



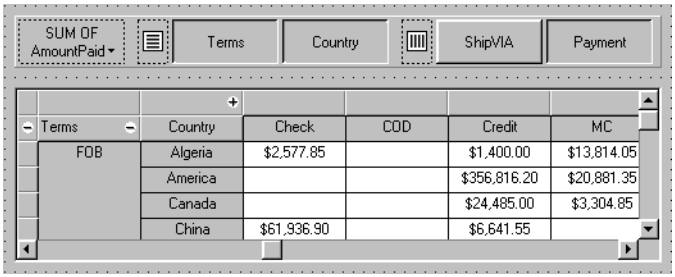
Payment					
AmEx	Cash	Check	COD	Credit	MC
$134,753.40	$164,003.65	$270,492.15	$33,776.55	$1,332,430.25	$250,163.25

Mehrdimensionale Kreuztabellen

In mehrdimensionalen Kreuztabellen gibt es zusätzliche Dimensionen für die Zeilen und/oder Spalten. Beispielsweise könnte in einer zweidimensionalen Kreuztabelle der bezahlte Betrag mit den einzelnen Zahlungsweisen auch noch pro Land dargestellt werden.

Eine dreidimensionale Kreuztabelle könnte dann sogar die Beträge pro Zahlungsweise und die Waren pro Land darstellen. Betrachten Sie hierzu Abbildung 26.3.

Abbildung 27.3 Dreidimensionale Kreuztabelle



Terms	Country	Check	COD	Credit	MC
FOB	Algeria	$2,577.85		$1,400.00	$13,814.05
	America			$356,816.20	$20,881.35
	Canada			$24,485.00	$3,304.85
	China	$61,936.90		$6,641.55	

Entscheidungskomponenten verwenden

Die auf Seite 27-1 vorgestellten Entscheidungskomponenten ermöglichen die Tabellen- oder Diagrammdarstellung von mehrdimensionalen Daten. Jeder Datenmenge können mehrere Diagramme oder Tabellen zugeordnet werden. Das heißt, es kann mehrere Instanzen von *TDecisionPivot* geben, womit zur Laufzeit unterschiedliche Perspektiven auf die Daten geboten werden können.

So erzeugen Sie ein Formular mit Tabellen und Diagrammen, die auf mehrdimensionalen Daten basieren:

1. Legen Sie ein Formular an.
2. Fügen Sie die folgenden Komponenten in das Formular ein. Richten Sie dabei im Objektinspektor die nötigen Verknüpfungen ein:
   - Eine Datenmenge, das heißt ein *TDecisionQuery*-Objekt oder *TQuery*. Details finden Sie unter »Entscheidungsdatenmengen mit dem Entscheidungsabfragen-Editor erzeugen« auf Seite 27-6.
   - Einen Entscheidungswürfel (*TDecisionCube*). Verknüpfen Sie ihn mit der Datenmenge, indem Sie die Eigenschaft *DataSet* entsprechend setzen.
   - Eine Entscheidungsquelle (*TDecisionSource*). Stellen Sie die Verknüpfung zum Entscheidungswürfel über die Eigenschaft *DecisionCube* her.
3. Fügen Sie ein Entscheidungspivot (*TDecisionPivot*) hinzu. Die Verknüpfung zur Entscheidungsquelle nehmen Sie im Objektinspektor vor, indem Sie die Eigenschaft *DecisionSource* mit dem betreffenden Namen belegen. Das Entscheidungspivot ist zwar nicht unbedingt notwendig, aber sehr praktisch: Die angezeigten Dimensionen können zur Laufzeit und auch während des Entwurfs über Schaltflächen ausgewählt werden.

   Standardmäßig werden die Schaltflächen horizontal ausgerichtet. Die Schaltflächen an der linken Seite des Entscheidungsgitters gelten für Zeilen des Entscheidungsgitters, die Schaltflächen am oberen Rand gelten entsprechend für die Spalten des Gitters.

   Die Ausrichtung kann mit der Eigenschaft *GroupLayout* des Entscheidungsgitters im Detail festgelegt werden. Neben der Vorgabe *xtHorizontal* stehen noch *xtVertical* und *xtLeftTop* zur Auswahl. Weitere Informationen hierzu finden Sie im Abschnitt »Mit Entscheidungspivots arbeiten« auf Seite 27-11.
4. Fügen Sie ein oder mehrere Entscheidungsgitter und -graphen hinzu, die mit der Datenquelle verbunden sind. Näheres hierzu finden Sie in den Abschnitten »Entscheidungsgitter erstellen und verwenden« auf Seite 27-12 und»Entscheidungsgraphen erstellen und verwenden« auf Seite 27-15.
5. Die Tabellen, Felder und Zusammenfassungen, die im Gitter oder im Graphen erscheinen sollen, werden entweder im Decision-Query-Editor oder über die Eigenschaft *SQL* von *TDecisionQuery* (bzw. *TQuery*) festgelegt. Das letzte Feld in der SELECT-Anweisung sollte das Zusammenfassungsfeld bezeichnen. Die anderen Felder müssen GROUP BY-Felder sein. Siehe hierzu den Abschnitt »Entscheidungsdatenmengen mit dem Entscheidungsabfragen-Editor erzeugen« auf Seite 27-6.
6. Setzen Sie die Eigenschaft *Active* der Entscheidungsabfrage (oder der alternativ von Ihnen gewählten Datenmenge) auf *True*.
7. Nun können Sie Daten in unterschiedlichster tabellarischer oder grafischer Form darstellen. Siehe auch »Mit Entscheidungsgittern arbeiten« auf Seite 27-12 und »Mit Entscheidungsgraphen arbeiten« auf Seite 27-15.

Abbildung 27.1 on page 27-2 zeigt alle Entscheidungskomponenten im Überblick.

Datenmengen und Entscheidungskomponenten

Die einzige Entscheidungskomponente, die eine direkte Verknüpfung zur Datenmenge besitzt, ist der Entscheidungswürfel *TDecisionCube*. Dieses Objekt erwartet Daten, die mit einer gültigen SQL-Anweisung in Gruppen und Zusammenfassungen gegliedert sind. Die Anweisung GROUP BY muß dieselben Nicht-Zusammenfassungsfelder wie die SELECT-Anweisung enthalten, und zwar in der gleichen Reihenfolge. Zusammenfassungsfelder müssen gekennzeichnet sein.

Die Entscheidungsabfrage *TDecisionQuery* ist eine spezialisierte Form von *TQuery* und vereinfacht die Konfiguration des Entscheidungswürfels *TDecisionCube*. Natürlich kann *TDecisionQuery* auch in seiner ursprünglichen Form als Datenmenge für *TDecisionCube* verwendet werden, allerdings liegt dann die korrekte Einrichtung der Datenmenge in der Verantwortung des Programmierers.

Ebenso kann eine gewöhnliche *TQuery*-Komponente oder eine andere Datenmengenkomponente als Datenmenge für *TDecisionCube* verwendet werden, aber dann liegt die korrekte Einrichtung der Datenmenge und des *TDecisionCube*-Objekts in der Verantwortung des Programmierers.

Damit der Entscheidungswürfel bestimmungsgemäß arbeitet, müssen alle projizierten Felder in der Datenmenge entweder Dimensionen oder Zusammenfassungen sein. Die Zusammenfassungen sollten zweckmäßigerweise additive Werte bilden (Summe oder Anzahl) und außerdem eine Gesamtauswertung für jede Kombination von Dimensionswerten anzeigen. Zur leichteren Konfiguration sollten die Bezeichner mit aussagekräftigen Vorsilben versehen werden (z.B. »Sum...« oder »Count...«).

Im Entscheidungswürfel funktionieren Pivots, Zwischensummen und Fokussierungen (Einzelwertuntersuchungen) nur dann korrekt, wenn es sich um Zusammenfassungen handelt, deren Zellen additiv sind, also kumulativ zu interpretieren sind. SUM und COUNT arbeiten additiv, AVERAGE, MAX und MIN dagegen nicht. Erstellen Sie Kreuztabellen mit Pivot-Funktion nur für solche Gitter, die ausschließlich additive Zusammenfassungen enthalten. Wenn nichtadditive Zusammenfassungen vorliegen, greifen Sie besser auf ein statisches Entscheidungsgitter zurück, das keine Pivots, Fokussierungen oder Zwischensummen enthält.

Da Durchschnittswerte mit Hilfe von SUM und einer anschließenden Division durch COUNT berechnet werden können, wird ein Pivot-Durchschnittswert automatisch hinzugefügt, sobald für ein Feld SUM- und COUNT-Dimensionen in die Datenmenge eingefügt werden. Diese Form der Durchschnittsberechnung ist der Anweisung AVERAGE auf jeden Fall vorzuziehen.

Durchschnittswerte können ebenfalls mit COUNT(*) berechnet werden. Hierzu muß ein Selektor "COUNT(*) COUNTALL" in der Abfrage plaziert werden. Bei der Verwendung von COUNT(*) genügt eine einzige Zusammenfassungsfunktion für alle Felder. Voraussetzung ist allerdings, daß keines der ausgewerteten Felder einen leeren Wert enthält und daß COUNT für alle Felder verfügbar ist.

Entscheidungsdatenmengen mit TQuery oder TTable erzeugen

Wenn Sie als Entscheidungsdatenmenge eine gewöhnliche *TQuery*-Komponente verwenden, müssen Sie die SQL-Anweisung manuell einrichten und dafür sorgen, daß eine GROUP BY-Anweisung übergeben wird, welche dieselben Felder wie die SELECT-Anweisung enthält, und zwar in der gleichen Reihenfolge.

Die SQL-Anweisung sollte folgende Form haben:

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
   ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

Die Reihenfolge der Felder in der SELECT-Anweisung sollte mit der Reihenfolge der GROUP BY-Felder übereinstimmen. Details hierzu finden Sie im Kapitel 21, »Abfragen«.

Bei *TTable*-Objekten müssen an den Entscheidungswürfel Informationen übergeben werden, welche Felder in der Abfrage Gruppierungsfelder und welche Zusammenfassungen sind. Hierzu geben Sie in der Eigenschaft *DimensionMap* des Entscheidungswürfels den Dimensionstyp für jedes Feld an. Die Felder müssen als Dimension oder Zusammenfassung gekennzeichnet werden. Bei Zusammenfassungen ist zusätzlich der Typ erforderlich. Da Pivot-Durchschnittswerte auf SUM / COUNT-Berechnungen basieren, muß außerdem der Basisfeldname übergeben werden, damit der Entscheidungswürfel Zuordnungspaare für SUM und COUNT bilden kann.

Entscheidungsdatenmengen mit dem Entscheidungsabfragen-Editor erzeugen

Alle Daten, die von den Entscheidungskomponenten verwendet werden, durchlaufen den Entscheidungswürfel. Dieser akzeptiert eine speziell formatierte Datenmenge, die am einfachsten mit einer SQL-Anweisung strukturiert werden kann. Details hierzu finden Sie im Abschnitt »Datenmengen und Entscheidungskomponenten« auf Seite 27-5.

Es können zwar auch *TTable* und *TQuery* als Entscheidungsdatenmengen verwendet werden, *TDecisionQuery* ist aber einfacher zu handhaben. Der zugehörige Editor unterstützt Sie bei der Konfiguration der Tabellen, Felder und Zusammenfassungen, die im Entscheidungswürfel erscheinen sollen. Außerdem wird die korrekte Formulierung der SELECT- und GROUP BY-Anweisungen vereinfacht.

Der Editor für Entscheidungsabfragen

So arbeiten Sie mit dem Editor für Entscheidungsabfragen:

1. Markieren Sie die Entscheidungsabfrage im Formular, und klicken Sie mit der rechten Maustaste. Wählen Sie dann im lokalen Menü den Befehl *Editor für Entscheidungswürfel*. Das Editor-Dialogfeld wird geöffnet.
2. Wählen Sie die zu verwendende Datenbank.

3 Bei Abfragen mit nur einer Tabelle wählen Sie diese aus der Liste *Tabelle* aus.

   Wenn es sich um komplexere Abfragen mit mehreren Tabellenverknüpfungen handelt, klicken Sie auf die Schaltfläche *SQL-Builder*, um den SQL-Abfragegenerator zu öffnen, oder geben Sie die SQL-Anweisung in das Eingabefeld auf der Registerkarte *SQL* ein.

4 Kehren Sie wieder in das Dialogfeld *Editor für Entscheidungswürfel* zurück.

5 Im Dialogfeld *Editor für Entscheidungswürfel* wählen Sie die Felder in der Liste *Verfügbare Felder* aus und kennzeichnen sie als *Dimensionen* oder *Zusammenfassungen*. Beim Einfügen von Feldern in die Liste *Zusammenfassungen* wählen Sie im angezeigten Menü den Typ der zu verwendenden Zusammenfassung: Summe, Anzahl oder Druchschnitt.

6 Die Listen *Dimensionen* und *Zusammenfassungen* enthalten per Voreinstellung alle Felder und Zusammenfassungen, die in der Eigenschaft *SQL* der Entscheidungsabfrage definiert sind. Wenn eine Dimension oder eine Zusammenfassung entfernt werden soll, markieren Sie das Element in der Liste und klicken auf die Pfeilschaltfläche links davon. Ebenso können Sie das Element auch doppelt anklicken, um es zu entfernen. Wenn es erneut eingefügt werden soll, markieren Sie es in der Liste *Verfügbare Felder* und klicken auf die Schaltfläche mit dem Rechtspfeil.

Damit ist der Inhalt des Entscheidungswürfels definiert. Mit der Eigenschaft *DimensionMap* und den Schaltflächen von *TDecisionPivot* können Sie die angezeigten Dimensionen konfigurieren. Weitere Informationen finden Sie unter »Mit Entscheidungswürfeln arbeiten«, »Mit Entscheidungsquellen arbeiten« auf Seite 27-10 und »Mit Entscheidungspivots arbeiten« auf Seite 27-11.

Hinweis Wenn Sie den Editor für Entscheidungsabfragen verwenden, wird die Abfrage anfangs in der SQL-Syntax gemäß ANSI-92 verarbeitet und dann bei Bedarf in den Dialekt übersetzt, den der Server verwendet. Der Editor liest und generiert nur ANSI-Standard-SQL. Das Übersetzungsergebnis wird automatisch der Eigenschaft *SQL* von *TDecisionQuery* zugewiesen. Um eine Abfrage zu ändern, modifizieren Sie am besten die in ANSI-92 vorliegende Version in der Entscheidungsabfrage, anstatt die Eigenschaft *SQL*.

Eigenschaften von TDecisionQuery

Das Objekt *TDecisionQuery* besitzt ausschließlich Eigenschaften, die von anderen Objekten geerbt sind. Die wichtigsten davon sind *Active* und *SQL*. Die Eigenschaft *Active* wird in der Hilfe und der *VCL-Referenz* unter *TDataSet* beschrieben, und *SQL* wird unter *TQuery* beschrieben. Kapitel 21, »Abfragen«, beschreibt das Thema Abfragen ausführlich.

Mit Entscheidungswürfeln arbeiten

Der Entscheidungswürfel *TDecisionCube* ist ein mehrdimensionaler Datenspeicher, der seine Daten aus einer Datenmenge bezieht. Bei dieser Datenmenge handelt es sich typischerweise um eine speziell strukturierte SQL-Anweisung, die über *TDecisionQuery* oder *TQuery* erstellt wird. Die Daten werden so gespeichert, daß sie auf einfache Weise reorganisiert und zusammengefaßt werden können, ohne daß dazu die Abfrage erneut ausgeführt werden muß. Diese Umstrukturierung von Daten in einer Kreuztabelle wird als »Schwenken« (engl. »pivoting«) bezeichnet.

Eigenschaften und Ereignisse

Die Eigenschaftsgruppe *DimensionMap* von *TDecisionCube* dient nicht nur dazu, die angezeigten Dimensionen und Zusammenfassungen auszuwählen, sondern gestattet auch die Festlegung von Datenbereichen. Zusätzlich kann die maximale Anzahl von Dimensionen angegeben werden, die der Entscheidungswürfel unterstützt. Auch die Datenanzeige zur Entwurfszeit wird hier gesteuert. Sie können Namen, Werte, Zwischensummen und Daten anzeigen oder ausblenden. Abhängig von der Datenquelle kann die Datenanzeige zur Entwurfszeit sehr zeitintensiv sein.

Wenn Sie im Objektinspektor auf die Ellipsen-Schaltfläche neben der Eigenschaft *DimensionMap* klicken, wird der Editor für den Entscheidungswürfel mit mehreren Registerkarten geöffnet.

Das Ereignis *OnRefresh* wird ausgelöst, wenn der Zwischenspeicher des Entscheidungswürfels neu aufgebaut wird. Als Reaktion darauf kann beispielsweise Speicher freigegeben oder die Dimensionszahl neu bemessen werden.

Der Editor für den Entscheidungswürfel

Im Editor für den Entscheidungswürfel kann die Dimensionszuordnung neu gestaltet werden. Dazu erfolgt ein Zugriff auf die *DimensionMap*-Eigenschaften. Der Editor kann entweder mit dem Objektinspektor geöffnet werden, wie im vorigen Abschnitt beschrieben, oder per Klick mit der rechten Maustaste auf einen Entscheidungswürfel und dem Befehl *Editor für Entscheidungswürfel* aus dem lokalen Menü.

Das Dialogfeld *Editor für Entscheidungswürfel* besteht aus zwei Registerkarten:

- *Dimensionseinstellungen :* Die verfügbaren Dimensionen können aktiviert, deaktiviert, umbenannt, neu formatiert, ausgelagert und mit Bereichen versehen werden.
- *Speicherkontrolle:* Maximalwerte bestimmen die mögliche Anzahl von Dimensionen und Zusammenfassungen, die gleichzeitig aktiv sein können. Außerdem sind Informationen über die Speicherbelegung sowie über die Namen und Daten enthalten, die während des Entwurfs angezeigt werden.

Dimensionseinstellungen anzeigen und ändern

Um die dargestellten Dimensionen zu konfigurieren, öffnen Sie den Editor für den Entscheidungswürfel und klicken auf das Register *Dimensionseinstellungen*. Anschließend wählen Sie in der Liste *Verfügbare Felder* eine Dimension oder Zusammenfassung. Im rechten Bereich des Editors werden die zugehörigen Informationen angezeigt:

- Wenn der Name für eine Dimension oder Zusammenfassung geändert werden soll, der im Entscheidungspivot, Entscheidungsgitter oder Entscheidungsgraphen angezeigt wird, geben Sie im Feld *Namen anzeigen* einen neuen Namen ein.
- Das Eingabefeld *Typ* gibt Auskunft, ob es sich bei dem ausgewählten Feld um eine Dimension oder eine Zusammenfassung handelt, wenn die Datenmenge eine *TTable*-Komponente ist.
- Eine bestimmte Dimension oder Zusammenfassung kann aktiviert und deaktiviert werden, indem Sie die Einstellung in der Dropdown-Liste *Aktiver Typ* ändern: *Aktiv*, *Wie benötigt* oder *Inaktiv*. Es kann Speicher gespart werden, wenn Sie eine Dimension deaktivieren oder auf den Status *Wie benötigt* setzen.
- Im Eingabefeld *Format* kann das Format einer Dimension oder Zusammenfassung mit einem Format-String eingestellt werden.
- Die Dropdown-Liste *Gruppierung* ermöglicht die Bündelung von Dimensionen und Zusammenfassungen nach Jahr, Quartal oder Monat. Eine besondere Funktion übernimmt die Einstellung *Einzelner Wert*. Die betreffende Dimension oder Zusammenfassung wird dann permanent ausgelagert, wodurch Speicher gespart werden kann, wenn eine Dimension sehr viele Werte besitzt. Weitere Informationen hierzu finden Sie im Abschnitt »Entscheidungskomponenten und Speicherverwaltung« auf Seite 27-22.
- Um den Startwert von Bereichen oder den untersuchten Wert einer als Menge vorliegenden Dimension festzulegen, wählen Sie zuerst in der Dropdown-Liste *Gruppierung* eine passende Einstellung und geben dann den Startwert oder den permanent fokussierten Wert in der Liste *Anfangswert* ein.

Grenzwerte für Dimensionen und Zusammenfassungen

In der Registerkarte *Speicherkontrolle* können Sie die maximale Anzahl von Dimensionen und Zusammenfassungen für Entscheidungspivots, Entscheidungsgitter und Entscheidungsgraphen vorgeben, die mit dem betreffenden Entscheidungswürfel verknüpft sind. Passen Sie die aktuellen Einstellungen bei Bedarf mit Hilfe der Steuerelemente des Dialogfeldes an. Hierdurch wird auch der Speicherbedarf eines Entscheidungswürfels bestimmt. Näheres hierzu finden Sie im Abschnitt »Entscheidungskomponenten und Speicherverwaltung« auf Seite 27-22.

Gestaltungsoptionen anzeigen und ändern

Mit der Optionsgruppe *Entwurfsdatenoptionen* können Sie während des Entwurfs den Informationsgehalt festlegen. Markieren Sie die Daten und Feldnamen, die angezeigt werden sollen. Während des Entwurfs kann es sehr zeitaufwendig sein, wenn große Mengen von Daten abgerufen werden müssen.

Mit Entscheidungsquellen arbeiten

Die Datenquelle für Entscheidungskomponenten, *TDecisionSource* definiert den aktuellen Pivot-Status von Entscheidungsgittern und -graphen. Zwei Objekte, denen dieselbe Datenquelle zugeordnet ist, befinden sich immer in demselben Pivot-Status.

Eigenschaften und Ereignisse

Im folgenden finden Sie eine Beschreibung einiger spezieller Eigenschaften und Ereignisse, mit denen das Aussehen und das Verhalten von Entscheidungsdatenquellen festgelegt werden kann:

- Die Eigenschaft *ControlType* von *TDecisionSource* gibt vor, ob die Schaltflächen des Entscheidungspivots mit der Logik von Kontrollfeldern (mehrere Auswahlmöglichkeiten) oder von Optionsfeldern (gegenseitig sich ausschließende Auswahl) funktionieren sollen.
- Die Eigenschaften *SparseCols* und *SparseRows* von *TDecisionSource* steuern die Anzeige von Spalten oder Zeilen, die keine Werte enthalten. Der Wert *True* bewirkt, daß solche Spalten angezeigt werden.
- *TDecisionSource* besitzt die folgenden Ereignisse:
  - *OnLayoutChange* wird ausgelöst, sobald der Benutzer die Daten reorganisiert.
  - *OnNewDimensions* wird ausgelöst, wenn die Daten komplett verändert werden, weil beispielsweise Zusammenfassungs- oder Dimensionsfelder gewechselt werden.
  - *OnSummaryChange* wird ausgelöst, wenn die aktuelle Zusammenfassung geändert wird.
  - *OnStateChange* wird ausgelöst, sobald der Entscheidungswürfel aktiviert oder deaktiviert wird.
  - *OnBeforePivot* wird ausgelöst, wenn Änderungen zwar übergeben, aber in der Darstellung der Benutzeroberfläche noch nicht berücksichtigt wurden. Als Entwickler können Sie Änderungen vornehmen, beispielsweise an der Kapazität oder am Pivot-Status, bevor der Benutzer der Anwendung das Ergebnis seiner Aktion sieht.
  - *OnAfterPivot* wird nach einer Änderung des Pivot-Status ausgelöst. Programmseitig können zu diesem Zeitpunkt Informationen angefordert werden.

Mit Entscheidungspivots arbeiten

Die Komponente *TDecisionPivot* fungiert als »Bedienfeld« für den Entscheidungswürfel: Per Mausklick können die Dimensionen (Felder) des Entscheidungswürfels geöffnet und geschlossen werden. Wenn eine Zeile oder Spalte mit einer der Schaltflächen von *TDecisionPivot* geöffnet wird, werden die zugehörigen Dimensionen in der Komponente *TDecisionGrid* oder *TDecisionGraph* sichtbar. Beim Schließen einer Dimension werden die Detaildaten ausgeblendet und nur noch in die Gesamtwerte anderer Dimensionen eingerechnet. Eine Dimension kann auch fokussiert werden, das heißt, daß nur die Zusammenfassungen eines bestimmten Wertes des Dimensionsfeldes angezeigt werden.

Das Entscheidungspivot dient auch dazu, die Dimensionen zu reorganisieren, die im Entscheidungsgitter und im Entscheidungsgraphen angezeigt werden. Ziehen Sie dazu einfach eine Schaltfläche auf den Zeilen / Spaltenbereich, oder ordnen Sie die darin enthaltenen Schaltflächen neu an.

Die Abbildungen 26.1, 26.2 und 26.3 zeigen Entscheidungspivots zur Entwurfszeit.

Eigenschaften von Entscheidungspivots

Im folgenden werden einige Eigenschaften vorgestellt, die das Aussehen und das Verhalten von Entscheidungspivots steuern:

- Mit der Eigenschaft *ButtonAutoSize* kann verhindert werden, daß die Schaltflächen in der Größe an die Komponente angepaßt werden (*False*).
- Die Eigenschaft *Groups* von *TDecisionPivot* gibt vor, welche Dimensionsschaltflächen angezeigt werden. Es sind beliebige Kombinationen möglich. Zusätzliche Flexibilität bei der Gestaltung ergibt sich daraus, daß ein *TDecisionPivot*-Objekt beispielsweise nur die Schaltflächen für die Zeilen enthalten kann und ein anderes nur die Schaltflächen für die Spalten.
- Typischerweise wird *TDecisionPivot* über *TDecisionGrid* angeordnet. Per Voreinstellung sind die Schaltflächen auf der linken Seite des Entscheidungspivots den Zeilen zugeordnet und die Schaltflächen auf der rechten Seite den Spalten.
- Die Positionierung der Schaltflächen eines *TDecisionPivot*-Objekts kann mit der Eigenschaft *GroupLayout* gesteuert werden. Mögliche Werte sind *xtVertical*, *xtLeftTop* oder *xtHorizontal* (Vorgabewert, siehe oben).

Entscheidungsgitter erstellen und verwenden

Komponenten des Typs *TDecisionGrid* können Daten, die in Kreuztabellenstruktur vorliegen, in Tabellenform darstellen. Details hierzu finden Sie auf Seite 27-2. Abbildung 27.1 on page 27-2 zeigt ein Entscheidungsgitter in der Entwurfsphase.

Entscheidungsgitter erstellen

In folgenden Schritten legen Sie ein Formular mit Kreuztabellendarstellung an:

1. Führen Sie die Schritte 1 bis 3 aus, die im Abschnitt »Entscheidungskomponenten verwenden« auf Seite 27-3 beschrieben sind.
2. Fügen Sie ein oder mehrere Entscheidungsgitter (*TDecisionGrid*) ein. Stellen Sie mit Hilfe des Objektinspektors die notwendigen Verknüpfungen zur Datenquelle (*TDecisionSource*) her, indem Sie der Eigenschaft *DecisionSource* einen Wert zuweisen.
3. Fahren Sie mit den Schritten 5 - 7 fort, die unter »Entscheidungskomponenten verwenden« beschrieben sind.

Die Konfiguration und die Bedienung des Entscheidungsgitters wird unter “Mit Entscheidungsgittern arbeiten” im folgenden Abschnitt beschrieben.

Wenn Sie eine Diagrammdarstellung hinzufügen möchten, finden Sie Hinweise im Abschnitt »Entscheidungsgraphen erstellen« auf Seite 27-15.

Mit Entscheidungsgittern arbeiten

Die Entscheidungsgitterkomponente *TDecisionGrid* übernimmt die Anzeige der Daten, die über einen Entscheidungswürfel (*TDecisionCube*) von den Entscheidungsdatenquellen (*TDecisionSource*) bezogen werden.

Per Voreinstellung befinden sich die Dimensionsfelder des Gitters am linken und / oder am oberen Rand. Maßgebend ist hier die Gruppierung, die in der Datenmenge definiert ist. Unter jedem Feld werden für die einzelnen Datenwerte die Kategorien angezeigt. Ein Entscheidungsgitter bietet die folgenden Funktionen:

- Dimensionen öffnen und schließen
- Zeilen und Spalten reorganisieren
- Werte fokussieren
- Die Dimensionsauswahl auf eine einzige Achse pro Dimension begrenzen

Weitere Details zu den Merkmalen von Entscheidungsgittern finden Sie im Abschnitt »Eigenschaften von Entscheidungsgittern« auf Seite 27-14.

Felder im Entscheidungsgitter öffnen und schließen

Wenn in einem Dimensions- oder Zusammenfassungsfeld ein Pluszeichen (+) angezeigt wird, bedeutet dies, daß rechts von ihm ein oder mehrere Felder geschlossen (verborgen) sind. Mit einem Klick auf das Pluszeichen können also weitere Felder eingeblendet werden. Ein Minuszeichen signalisiert, daß das Feld vollständig geöffnet ist. Wenn Sie das Minuszeichen anklicken, wird das Feld geschlossen. Diese Funktion kann deaktiviert werden (siehe Abschnitt »Eigenschaften von Entscheidungsgittern« auf Seite 27-14.

Entscheidungsgitter reorganisieren

Die Kopfbereiche von Zeilen und Spalten können an eine neue Position auf derselben oder auf einer anderen Achse verschoben werden. Auf diese Weise kann das Gitter reorganisiert werden, um eine neue Perspektive zu gewinnen. Diese Umstrukturierung durch Vertauschen von Positionen wird auch als »Schwenken« bezeichnet. Diese Funktion kann deakitiviert werden. Siehe »Eigenschaften von Entscheidungsgittern« auf Seite 27-14.

Wenn Sie ein Entscheidungspivot im Formular plaziert haben, kann die Anzeige auch mit dessen Schaltlfächen neu konfiguriert werden. Näheres im Abschnitt »Mit Entscheidungspivots arbeiten« auf Seite 27-11.

Detaildaten in Entscheidungsgittern anzeigen

Wenn Detaildaten einer Dimension benötigt werden, kann diese fokussiert und näher untersucht werden.

Wenn Sie beispielsweise eine Kategoriebeschriftung (Zeilenkopf) mit der rechten Maustaste anklicken, läßt sich in einem einzeiligen lokalen Menü die Option *Diesen Wert untersuchen* aktivieren. Sie sehen dann die Kategoriebeschriftungen dieser Dimension nicht, weil nur die Datensätze für einen einzigen Kategoriewert angezeigt werden. Mit einem Entscheidungspivot können Sie dasselbe erreichen.

So fokussieren Sie eine bestimmte Dimension:

- Klicken Sie mit der rechten Maustaste auf eine Kategoriebeschriftung, und wählen Sie *Diesen Wert untersuchen*, oder
- Klicken Sie mit der rechten Maustaste auf eine Pivot-Schaltfläche, und wählen Sie *Untersucht*.

So aktivieren Sie die gesamte Dimension erneut:

- Klicken Sie mit der rechten Maustaste auf die zugehörige Pivot-Schaltfläche oder auf die linke obere Ecke des Entscheidungsgitters, und wählen Sie die Dimension.

Die Dimensionszahl in Entscheidungsgittern begrenzen

Die Entscheidungsdatenquelle besitzt eine Eigenschaft *ControlType*, mit der Sie festlegen können, ob für die Gitterachsen mehrere Dimensionen ausgewählt werden können. Details finden Sie unter »Mit Entscheidungsquellen arbeiten« auf Seite 27-10.

Eigenschaften von Entscheidungsgittern

In einem Entscheidungsgitter *TDecisionGrid* werden die Daten angezeigt, die über einen Entscheidungswürfel (*TDecisionCube*) von der Entscheidungsdatenquelle *TDecisionSource* bezogen werden. Per Voreinstellung befinden sich die Kategoriefelder am linken Rand und oberen Rand des Gitters.

Im folgenden finden Sie die Beschreibung einiger Eigenschaften, mit denen das Aussehen und das Verhalten von Entscheidungsgittern gesteuert werden kann:

- *TDecisionGrid* besitzt für jede Dimension eigene Eigenschaften. Die Einstellungen nehmen Sie im Objektinspektor vor. Klicken Sie neben der Eigenschaft *Dimensions* auf die Ellipsen-Schaltfläche, und wählen Sie in dem erscheinenden Editor eine der vorhandenen Dimensionen aus. Deren Eigenschaften können dann im Objektinspektor bearbeitet werden: Die Eigenschaft *Alignment* bestimmt die Ausrichtung der Kategoriebeschriftung der betreffenden Dimension. Der vorgegebene Name der Dimension kann in der Eigenschaft *Caption* überschrieben werden. Die Eigenschaft *Color* definiert die Farbe der Kategoriebeschriftung, und *FieldName* enthält den Namen der aktiven Dimension. Die Eigenschaft *Format* kann einen Format-String aufnehmen, der für diesen Datentyp gültig ist. Der Wert von *Subtotals* signalisiert, ob für diese Dimension Zwischensummen angezeigt werden sollen oder nicht. In Verbindung mit Zusammenfassungsfeldern werden dieselben Eigenschaften verwendet, um die Datenanzeige im Zusammenfassungsbereich des Gitters zu steuern. Sobald Sie die Eigenschaften der einzelnen Dimensionen eingestellt haben, klicken Sie entweder eine Komponente im Formular an oder wählen eine Komponente aus der Dropdown-Liste unter der Titelleiste des Objektinspektors.
- Die Eigenschaft *Options* eines Entscheidungsgitters beeinflußt die grafische Gestaltung sowie die Funktionalität des Gitters: Wenn *cgGridLines* den Wert *True* hat, erscheinen Gitterlinien; *cgOutliner* steuert, ob das Ein- und Ausblenden von Dimensionen mit den Symbolen + und - aktiv ist. Die Möglichkeit zur Reorganisation per Drag&Drop wird mit *cgPivotable* aktiviert oder deaktiviert.
- Das Ereignis *OnDecisionDrawCell* von *TDecisionGrid* gibt ihnen als Programmierer Gelegenheit, das Aussehen der Zelle zu ändern. Das Ereignis übergibt die Eigenschaften *String*, *Font* und *Color* der aktuellen Zelle als Referenzparameter. Sie können deren Wert ändern, um spezielle Effekte zu realisieren, beispielsweise einen roten Feldhintergrund für negative Zahlen. Zusätzlich zum Anzeigestatus (*Drawstate*), der von *TCustomGrid* übergeben wird, liefert das Ereignis den Wert von *TDecisionDrawState*, mit dem der Typ der gerade dargestellten Zelle ermittelt werden kann. Weitere Informationen über die Zelle können mit den Funktionen *Cells*, *CellValueArray* oder *CellDrawState* abgerufen werden.
- Das Ereignis *OnDecisionExamineCell* von *TDecisionGrid* dient zur Verknüpfung eines Mausklick-Ereignisses (rechte Maustaste) mit einer Datenzelle. Im Programm können dann zum Beispiel erweiterte Informationen wie etwa Detaildatensätze angezeigt werden. Sobald der Benutzer eine Datenzelle mit der rechten Maustaste anklickt, erhält das Ereignis sämtliche Informationen darüber, wie der Datenwert gebildet wurde.

Entscheidungsgraphen erstellen und verwenden

Entscheidungsgraphen (*TDecisionGraph*) übernehmen die grafische Aufbereitung von verwendete Daten, die in Kreuztabellenstruktur vorliegen. Dabei wird der Wert einer bestimmten Zusammenfassung bezogen auf eine oder mehrere Dimensionen in Diagrammform dargestellt. Weitere Informationen über Kreuztabellen finden Sie auf Seite 27-2. Die Abbildungen 26.1 auf Seite 27-2 und 26.4 auf Seite 27-16 zeigen eine solche Komponente zur Entwurfszeit.

Entscheidungsgraphen erstellen

So legen Sie ein Formular mit Entscheidungsgraphen an:

1. Führen Sie die Schritte 1 – 3 aus, die im Abschnitt »Entscheidungskomponenten verwenden« auf Seite 27-3 beschrieben sind.
2. Fügen Sie ein oder mehrere Entscheidungsgraphen hinzu (*TDecisionGraph*). Stellen Sie dann im Objektinspektor mit Hilfe der Eigenschaft *TDecisionSource* die Verknüpfung zur Datenquelle her.
3. Fahren Sie mit den Schritten 5 - 7 fort, die im Abschnitt »Entscheidungskomponenten verwenden« beschrieben sind.
4. Klicken Sie mit der rechten Maustaste auf das Diagramm, und wählen Sie *Diagramm bearbeiten*, wenn die Darstellung der Reihen geändert werden soll. Für jede Diagrammdimension können individuelle Eigenschaftswerte festgelegt werden. Details finden Sie im Abschnitt »Entscheidungsgraphen gestalten« auf Seite 27-17.

Eine Beschreibung dessen, was im Entscheidungsgraphen angezeigt wird, finden Sie unter »Mit Entscheidungsgraphen arbeiten«.

Wenn Sie ein Entscheidungsgitter oder eine Kreuztabelle hinzufügen möchten, finden Sie Hinweise im Abschnitt »Entscheidungsgitter erstellen und verwenden« auf Seite 27-12.

Mit Entscheidungsgraphen arbeiten

Die Komponente *TDecisionGraph* stellt die Felder aus der Entscheidungsdatenquelle (*TDecisionSource*) als dynamisches Diagramm dar, dessen Darstellung sich ändert, wenn Datendimensionen geöffnet, geschlossen, verschoben oder mit dem Entscheidungspivot (*TDecisionPivot*) neu angeordnet werden. Farben, Legenden und Typ des Diagramms können detailliert angepaßt werden.

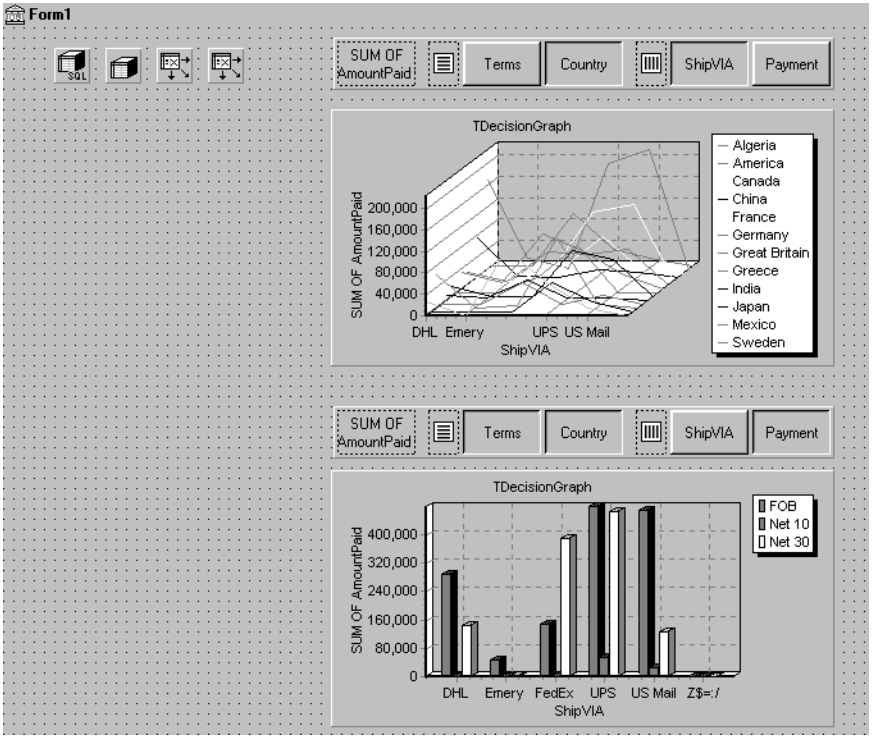
Die Daten werden aus einer speziell formatierten Datenmenge wie *TDecisionQuery* abgerufen. Einen Überblick über diese Entscheidungskomponente und über die Anordnung der Daten finden Sie unter »Entscheidungskomponenten verwenden« auf Seite 27-3.

Per Voreinstellung wird die erste Zeilendimension zur x-Achse und die erste Spaltendimension zur y-Achse.

Entscheidungsgraphen können Entscheidungsgitter entweder ersetzen oder ergänzen. Die beiden Komponenten sind mit derselben Entscheidungsdatenquelle verknüpft und zeigen dieselben Dimensionen an. Wenn verschiedene Zusammenfassungsdaten für dieselben Dimensionen angezeigt werden sollen, können mehrere Entscheidungsgraphen mit derselben Datenquelle verknüpft werden. Wenn umgekehrt verschiedene Dimensionen angezeigt werden sollen, verknüpfen Sie die Entscheidungsgraphen mit mehreren Entscheidungsquellen.

In Abbildung 25.4 sind das erste Entscheidungspivot und der erste Entscheidungsgraph mit der ersten Entscheidungsdatenquelle verbunden. Die zweite Datenquelle ist mit jeweils dem anderen Pivot und Graphen verknüpft. Jedes Diagramm zeigt damit andere Dimensionen.

Abbildung 27.4 Entscheidungsgraphen mit Verknüpfung zu verschiedenen Datenquellen



Details zum Bildinhalt eines Entscheidungsgraphen finden Sie im nächsten Abschnitt »Der Informationsgehalt von Entscheidungsgraphen«.

Hinweise zum Aufbau und zur Konfiguration von Entscheidungsgraphen finden Sie im Abschnitt »Entscheidungsgraphen erstellen«.

Eine Beschreibung der Eigenschaften von Entscheidungsgraphen und der Verfahren zum Ändern von Aussehen und Verhalten finden Sie im Abschnitt »Entscheidungsgraphen gestalten« auf Seite 27-17.

Der Informationsgehalt von Entscheidungsgraphen

Standardmäßig stellt der Entscheidungsgraph Zusammenfassungswerte für die Kategorien im ersten aktiven Zeilenfeld dar (entlang der Y-Achse), und zwar mit Bezug auf die Werte im ersten aktiven Spaltenfeld (entlang der X-Achse). Jede dargestellte Kategorie erscheint als eigene Reihe.

Wenn nur eine einzige Dimension ausgewählt ist (z.B. im Entscheidungspivot), besteht das Diagramm auch nur aus einer Reihe.

Wenn Sie ein Entscheidungspivot integriert haben, können Sie mit dessen Schaltflächen bestimmen, welche Felder des Entscheidungswürfels (Dimensionen) in die Darstellung einbezogen werden sollen. Um die Achsenzuordnung zu vertauschen, ziehen Sie die Schaltflächen von einer Seite des Pivots auf die andere. Bei einem eindimensionalen Graphen können Sie die Zeilen- und Spaltensymbole als Ziel eines Drag&Drop-Vorgangs verwenden. Die Schaltflächen gelangen dann auf die jeweils andere Seite des Pivots, und der Graph wird mehrdimensional.

Wenn immer nur eine Spalte und eine Zeile gleichzeitig aktiv sein sollen, können Sie die Funktionslogik der Pivot-Schaltflächen anpassen. Setzen Sie dazu die Eigenschaft *ControlType* des Objekts *TDecisionSource* auf *xtRadio*. Für jede Achse des Entscheidungswürfels kann es dann immer nur ein aktives Feld geben. Der Inhalt des Entscheidungsgraphen wird entsprechend angepaßt. Die Einstellung *xtRadioEx* entspricht grundsätzlich *xtRadio*, mit dem Unterschied, daß nicht alle Zeilen-/Spaltendimensionen gleichzeitig geschlossen sein können.

Wenn in Ihrem Formular ein Entscheidungsgitter und ein Entscheidungsgraph mit derselben Datenquelle (*TDecisionSource*) verknüpft sind, werden Sie *ControlType* wahrscheinlich auf *xtCheck* setzen, um die Flexibilität von *TDecisionGrid* auszunutzen.

Entscheidungsgraphen gestalten

Die Komponente *TDecisionGraph* stellt die Felder aus der Entscheidungsdatenquelle (*TDecisionSource*) als dynamisches Diagramm dar, dessen Darstellung sich ändert, wenn Datendimensionen geöffnet, geschlossen, verschoben oder mit dem Entscheidungspivot (*TDecisionPivot*) neu angeordnet werden. Farben, Legenden und Typ des Diagramms können detailliert angepaßt werden.

So verändern Sie das Aussehen eines Diagramms:

1. Klicken Sie mit der rechten Maustaste auf das Diagramm, und wählen Sie *Diagramm bearbeiten*. Das Dialogfeld *Diagramm bearbeiten* wird geöffnet.
2. In der Registerkarte *Diagramm* finden Sie in der Liste *Reihen* alle Dimensionen des Entscheidungswürfels (mit Schablone) und eine Kennzeichnung des aktuellen Anzeigestatus.

   Jede Kategorie oder Reihe bildet ein eigenes Objekt, so daß Sie folgende Möglichkeiten haben:

- Reihen, die von vorhandenen Reihen abgeleitet sind, können hinzugefügt oder gelöscht werden.
- Der voreingestellte Diagrammtyp kann ebenso geändert werden wie die Titel von Schablonen und Reihen.

Eine Beschreibung der weiteren Registerkarten finden Sie in der Hilfe.

3 In der Registerkarte *Reihen* können Schablonen für die Dimensionen eingerichtet werden, die Sie dann für die einzelnen Diagrammreihen anpassen.

Per Voreinstellung werden alle Reihen als Balkendiagramm mit bis zu 16 Farben dargestellt. Sie können den Schablonentyp sowie die Eigenschaftswerte ändern und so eigene Schablonen definieren. Schablonen werden verwendet, um für jeden neuen Status eines reorganisierten Diagramms dynamisch die benötigten Reihen zu erstellen. Details hierzu finden Sie im Abschnitt »Standardwerte für Entscheidungsgraphen als Schablonen« auf Seite 27-18.

Wenn Sie einzelne Reihen anpassen müssen, folgen Sie den Schritten im Abschnitt »Reihen in Entscheidungsgraphen gestalten« auf Seite 27-19.

Eine ergänzende Beschreibung der Registerkarte *Reihen* finden Sie in der Online-Hilfe unter »Reihen«.

Standardwerte für Entscheidungsgraphen als Schablonen

Entscheidungsgraphen beziehen ihre Werte von zwei Dimensionen des Entscheidungswürfels. Die eine Dimension wird als Achse des Graphen angezeigt, aus der anderen wird eine Anzahl von Reihen gebildet. In der Schablone für diese Dimension sind Standardeigenschaften für die Reihen hinterlegt (z.B. der Diagrammtyp). Wenn der Benutzer eine Reorganisation durchführt, werden die erforderlichen Reihen der Dimension auf Basis dieser Schablone erzeugt.

Es gibt eine spezielle Schablone für den Fall, daß nur eine einzige Dimension aktiv ist. In diesem Fall wird oft ein Tortendiagramm eingesetzt, weshalb hierzu eine eigene Schablone existiert.

Schablonen bieten folgende Möglichkeiten:

- Standard-Diagrammtyp ändern
- Weitere Eigenschaften in der Schablone für Entscheidungsgraphen ändern
- Übergreifende Eigenschaften für Entscheidungsgraphen anzeigen und erstellen

Den Standard-Typ für Entscheidungsgraphen

So ändern Sie den Standard-Diagrammtyp:

1 Wählen Sie im Diagrammeditor in der Registerkarte *Diagramm* aus der Liste *Reihen* eine Reihe aus.

2 Klicken Sie auf *Ändern*.

3 Wählen Sie einen neuen Typ, und schließen Sie das Dialogfeld.

Weitere Eigenschaften in der Schablone für Entscheidungsgraphen ändern

So ändern Sie für Schablonen die Farbe und weitere Eigenschaften:

1. Öffnen Sie im Diagrammeditor die Registerkarte *Reihen*.
2. Wählen Sie aus der Dropdown-Liste am oberen Rand des Dialogfeldes eine Schablone.
3. Nehmen Sie in den verschiedenen Registerkarten die gewünschten Einstellungen vor.

Weitere Eigenschaften für Entscheidungsgraphen einstellen

Der Diagrammeditor enthält weitere Registerkarten für Eigenschaften:

1. Öffnen Sie im Diagrammeditor die Registerkarte *Diagramm*.
2. Es stehen nun acht weitere Registerkarten zur Verfügung, in denen Sie Einstellungen zur Gestaltung des Diagramms vornehmen können.

Reihen in Entscheidungsgraphen gestalten

Die Schablone enthält viele Standards für jede Dimension des Entscheidungswürfels, etwa den Diagrammtyp und den Anzeigemodus der Reihen. Weitere Vorgaben, wie die Farbe, werden über das Objekt *TDecisionGraph* definiert. Bei Bedarf können die Vorgaben für jede Reihe überschrieben werden.

Das Konzept, das diesen Schablonen zugrundeliegt, sieht vor, daß das Programm die Reihen für bestimmte Kategorien dynamisch bei Bedarf erzeugen und auch wieder verwerfen kann. Sie können benutzerdefinierte Reihen für bestimmte Kategoriewerte einrichten. Dazu reorganisieren Sie den Graphen so, daß in der Anzeige eine Reihe für die Kategorie angezeigt wird, die Sie anpassen wollen. Der Diagrammeditor bietet dann folgende Möglichkeiten:

- Diagrammtyp ändern
- Eigenschaften von Reihen ändern
- Angepaßte Diagrammreihen speichern

Hinweise zur Definition von Reihenschablonen und zur Festlegung von Standardwerten finden Sie im Abschnitt »Standardwerte für Entscheidungsgraphen als Schablonen« auf Seite 27-18.

Den Diagrammtyp für Reihen ändern

Per Voreinstellung besitzt jede Reihe den gleichen Diagrammtyp. Dieser ist in der Schablone für die zugehörige Dimension festgelegt. Wenn der Typ für alle Reihen auf einmal geändert werden soll, ändern Sie die Schablone (siehe auch »Den Standard-Typ für Entscheidungsgraphen« auf Seite 27-18).

So ändern Sie den Diagrammtyp für eine einzelne Reihe:

1. Wählen Sie im Diagrammeditor in der Registerkarte *Diagramm* eine Reihe aus.
2. Klicken Sie auf *Ändern*.

3 Wählen Sie einen neuen Typ, und schließen Sie das Dialogfeld.

4 Markieren Sie das Kontrollfeld zum Speichern von Reihen.

Weitere Eigenschaften von Reihen in Entscheidungsgraphen ändern

So ändern Sie die Farbe oder andere Eigenschaften der Reihen in einem Entscheidungsgraphen:

1 Öffnen Sie die Registerkarte *Reihen* im Diagrammeditor. Doppelklicken Sie auf eine der Reihen.

2 Wählen Sie aus der Dropdown-Liste am oberen Rand der Registerkarte eine Reihe aus.

3 Nehmen Sie in einer der Registerkarten *Format*, *Allgemein*, *Markierungen* die gewünschten Einstellungen vor.

4 Markieren Sie das Kontrollfeld *Reihen speichern*.

Einstellungen für die Reihen speichern

Per Voreinstellung werden zur Entwurfszeit nur die Einstellungen für Schablonen gespeichert. Änderungen an bestimmten Reihen werden nur dann gespeichert, wenn das entsprechende Kontrollfeld markiert ist.

Verwenden Sie diese Option mit Vorsicht, weil sich daraus eventuell sehr hoher Speicherbedarf ergeben könnte.

Entscheidungskomponenten zur Laufzeit

Zur Laufzeit kann der Benutzer mit Hilfe der Maus eine Vielzahl von Operationen ausführen. Im folgenden finden Sie eine Zusammenfassung.

- Entscheidungspivots zur Laufzeit
- Entscheidungsgitter zur Laufzeit
- Entscheidungsgraphen zur Laufzeit

Entscheidungspivots zur Laufzeit

Als Benutzer einer Anwendung haben Sie die folgenden Möglichkeiten:

- Klicken Sie mit der linken Maustaste auf die Zusammenfassungsschaltfläche am linken Ende des Entscheidungspivots. Dadurch wird eine Liste der verfügbaren Zusammenfassungen geöffnet, mit der sich der Inhalt von Entscheidungsgittern und Entscheidungsgraphen ändern läßt.
- Ein Klick mit der rechten Maustaste auf eine Dimensionsschaltfläche öffnet ein lokales Menü, das folgende Möglichkeiten bietet:
  - Die Dimension kann vom Zeilenbereich in den Spaltenbereich verlagert werden und umgekehrt.

  - Die Dimension kann fokussiert werden, wenn Detaildaten gewünscht sind. Dazu dient der Befehl *Untersucht*.
- Wenn eine Dimensionsschaltfläche im Anschluß an den Befehl *Untersucht* mit der linken Maustaste angeklickt wird, stehen die folgenden Möglichkeiten zur Verfügung:
  - *Dimension öffnen:* Die Darstellung setzt wieder an der obersten Ebene dieser Dimension auf.
  - *Alle Werte:* Es werden in Entscheidungsgittern nicht nur die Zusammenfassungen, sondern auch alle anderen Werte angezeigt.
  - *Andere Werte:* Eine Liste der verfügbaren Kategorien für diese Dimension.
- Mit einem Klick der linken Maustaste auf die Dimensionsschaltfläche kann die Dimension geöffnet und geschlossen werden.
- Per Drag&Drop lassen sich Dimensionsschaltflächen vom Spaltenbereich in den Zeilenbereich verschieben. Dabei können sie entweder neben vorhandenen Schaltflächen in diesem Bereich oder auf dem Zeilen-/Spaltensymbol abgelegt werden.

Entscheidungsgitter zur Laufzeit

Als Benutzer einer laufenden Anwendung haben Sie folgende Möglichkeiten:

- Wenn Sie mit der rechten Maustaste auf das Entscheidungsgitter klicken, wird ein lokales Menü geöffnet:
  - Zwischensummen können ein- und ausgeschaltet werden, und zwar für einzelne Datengruppen, für alle Werte einer Dimension oder für das gesamte Gitter.
  - Der Editor für den Entscheidungswürfel kann geöffnet werden (siehe auch Seite 27-8).
  - Dimensionen und Zusammenfassungen können geöffnet und geschlossen werden.
- Mit einem Klick auf die Symbole + und – in den Zeilen- und Spaltenköpfen können Dimensionen geöffnet und geschlossen werden.
- Per Drag&Drop können Dimensionen vom Zeilenbereich in den Spaltenbereich und umgekehrt verschoben werden.

Entscheidungsgraphen zur Laufzeit

Zur Laufzeit kann der Benutzer auf den Diagrammbereich klicken und den angezeigten Ausschnitt in horizontaler und vertikaler Richtung verschieben.

Entscheidungskomponenten und Speicherverwaltung

Wenn eine Dimension oder eine Zusammenfassung in den Entscheidungswürfel geladen wird, belegt sie Speicher. Mit dem Hinzufügen einer neuen Zusammenfassung steigt der Speicherbedarf proportional: Ein Entscheidungswürfel mit zwei Zusammenfassungen benötigt zweimal soviel Speicher wie ein Entscheidungswürfel mit einer Zusammenfassung. Ein Entscheidungswürfel mit drei Zusammenfassungen benötigt analog dazu dreimal soviel Speicher. Beim Hinzufügen von Dimensionen erhöht sich der Speicherbedarf dagegen schneller. Eine Dimension mit 10 Werten belegt den zehnfachen Speicherplatz. Eine Dimension mit 100 Werten belegt den 100fachen Speicherplatz und so fort. Dieser Zusammenhang kann schnell zu Problemen mit der Leistungsfähigkeit der Anwendung führen.

Die Entscheidungskomponenten bieten deshalb eine Reihe von Einstellmöglichkeiten für die mengenmäßige und zeitliche Koordination der Speicherbelegung. Details hierzu finden Sie unter *TDecisionCube* in der Online-Hilfe.

Maximalwerte für Dimensionen, Zusammenfassungen und Zellen

Die Eigenschaften *MaxDimensions* und *MaxSummaries* des Entscheidungswürfels können in Verbindung mit der Eigenschaft *CubeDim.ActiveFlag* verwendet werden, um die maximale Anzahl von Dimensionen und Zusammenfassungen zu bestimmen, die gleichzeitig geladen werden können. Alternativ dazu können Sie in der Registerkarte *Speicherkontrolle* des Editors für den Entscheidungswürfel entsprechende Grenzwerte eingeben.

Mit der Begrenzung der Dimensions- und Zusammenfassungszahl wird der Speicherplatz limitiert, den der Entscheidungswürfel belegen kann. Allerdings handelt es sich nur um eine relativ grobe Vorgabe, denn es wird an dieser Stelle nicht zwischen Dimensionen mit vielen Werten und Dimensionen mit wenigen Werten unterschieden. Wenn Sie eine genauere Kontrolle über den absoluten Speicherbedarf des Entscheidungswürfels wünschen, begrenzen Sie die Anzahl der Zellen im Würfel. Tragen Sie dazu in der Registerkarte *Speicherkontrolle* einen Wert in das Feld *Zellen* ein.

Den Status der Dimensionen einstellen

Die Eigenschaft *ActiveFlag* legt fest, welche Dimensionen geladen werden. Die nötigen Einstellungen nehmen Sie im Editor für den Entscheidungswürfel in der Registerkarte *Dimensionseinstellungen* vor, und zwar im Feld *Aktiver Typ*. Die Einstellung *Aktiv* bewirkt, daß die Dimension ohne Berücksichtigung irgendwelcher Bedingungen geladen wird und somit Speicher belegt. Beachten Sie, daß die Anzahl der Dimensionen in diesem Status immer kleiner als der Wert von *MaxDimensions* sein muß. Analoges gilt für die Anzahl der Zusammenfassungen und den Wert von *MaxSummaries*. Eine Dimension oder Eigenschaft sollte nur dann mit dem Status *Aktiv* versehen werden, wenn deren ständige Verfügbarkeit absolut notwendig ist. Die Einstellung *Aktiv* erschwert die Verwaltung des verfügbaren Speichers durch den Entscheidungswürfel.

Wenn *ActiveFlag* auf *Wie benötigt* gesetzt ist, wird eine Dimension oder Zusammenfassung nur dann geladen, wenn dabei die Grenzwerte *MaxDimensions*, *MaxSummaries* oder *MaxCells* nicht überschritten werden. Der Entscheidungswürfel lagert Dimensionen und Zusammenfassungen mit dem Status *Wie benötigt* aus dem Speicher aus bzw. lädt sie erneut, so daß die Grenzwerte *MaxCells*, *MaxDimensions* und *MaxSummaries* stets eingehalten werden. Nicht benötigte Daten werden somit nicht geladen. Insgesamt verbessert sich die Geschwindigkeit von Umstrukturierungsoperationen, selbst wenn das Laden ausgelagerter Dimensionen etwas Zeit in Anspruch nimmt.

Permanent ausgelagerte Dimensionen

Wenn Sie im Editor für den Entscheidungswürfel in der Registerkarte *Dimensionseinstellungen* unter *Gruppierung* die Einstellung *Einzelner Wert* gewählt haben und der *Anfangswert* ungleich NULL ist, wird die betreffende Dimension als »defokussiert« oder »permanent ausgelagert« bezeichnet. Zu einem bestimmten Zeitpunkt ist dann immer nur der Zugriff auf einen einzigen Wert dieser Dimension möglich, allerdings kann per Programm auf eine Folge von Werten sequentiell zugegriffen werden. Solche Dimensionen können nicht geschwenkt oder geöffnet werden.

Die Einbindung mehrdimensionaler Daten ist immer dann extrem speicherintensiv, wenn sehr viele Werte vorliegen. Indem solche Dimensionen permanent ausgelagert werden, erscheint immer nur die Zusammenfassungsinformation für einen einzigen Wert. Das verbessert einerseits die Lesbarkeit und erleichtert andererseits die Speicherverwaltung.

Teil

III

Verteilte Anwendungen entwickeln

Die Kapitel in Teil III dieses Handbuchs zeigen Konzepte und Techniken auf, die für die Erzeugung von Anwendungen erforderlich sind, die über ein lokales Netzwerk oder über das Internet weitergegeben werden.

Hinweis Die in Kapitel 28 beschriebenen Komponenten sind in der Enterprise-Version von Delphi enthalten. Die in Kapitel 29 behandelten Komponenten für Web-Server-Anwendungen und die in Kapitel 30 behandelten Socket-Komponenten stehen in der Professional- und der Enterprise-Version von Delphi zur Verfügung.

Kapitel

28

CORBA-Anwendungen

Delphi stellt Experten und Klassen zum Erstellen verteilter Anwendungen zur Verfügung, die auf CORBA (Common Object Request Broker Architecture) basieren. CORBA ist eine Spezifikation, die von der OMG (Object Management Group) definiert wurde, um die Komplexität, die das Entwickeln verteilter objektorientierter Anwendungen mit sich bringt, zu meistern.

Wie der Name vermuten läßt, stellt CORBA einen objektorientierten Ansatz für das Schreiben verteilter Anwendungen dar. Dies steht im Gegensatz zu einem nachrichtenorientierten Ansatz, wie er für HTTP-Anwendungen in Kapitel 29, »Internet-Server-Anwendungen« beschrieben ist. Unter CORBA implementieren Server-Anwendungen Objekte, auf die von Remote-Client-Anwendungen über entsprechend definierte Schnittstellen zugegriffen werden kann.

Hinweis COM bietet einen weiteren objektorientierten Ansatz zur Entwicklung verteilter Anwendungen. Weitere Informationen zu COM finden Sie in Kapitel 44, »COM-Technologien im Überblick«. Im Gegensatz zu COM ist CORBA jedoch ein Standard, der auch für Nicht-Windows-Plattformen geeignet ist. Dies bedeutet, daß Sie mit Delphi CORBA-Clients oder -Server schreiben können, die mit CORBA-Anwendungen kommunizieren können, die auf anderen Plattformen ausgeführt werden.

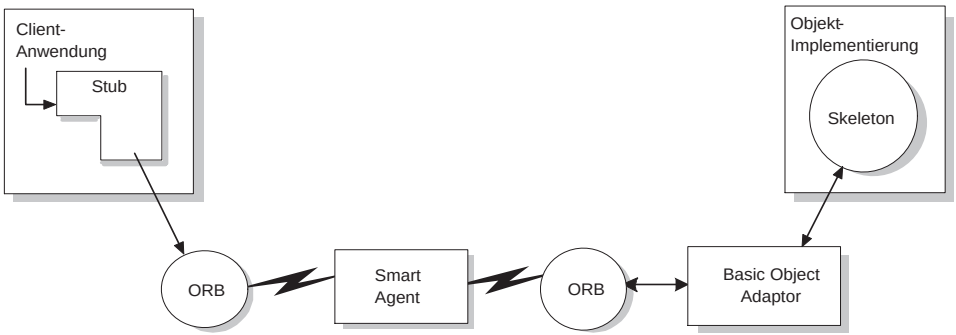
Die CORBA-Spezifikation definiert, wie Client-Anwendungen mit Objekten kommunizieren, die auf einem Server implementiert sind. Die Kommunikation erfolgt über einen ORB (Object Request Broker). Die CORBA-Unterstützung von Delphi basiert auf VisiBroker für C++ ORB (Version 3.3.2) mit einer spezielle Kapselung (orbpas.dll), die eine Teilmenge der ORB-Funktionen für Delphi-Anwendungen verfügbar macht.

Zusätzlich zur grundlegenden ORB-Technologie, welche die Kommunikation von Clients mit Objekten auf Server-Rechnern ermöglicht, werden über den CORBA-Standard eine Reihe von Standarddiensten definiert. Da diese Dienste entsprechend definierte Schnittstellen verwenden, können die Entwickler Clients erstellen, die diese Dienste nutzen, auch wenn die Server von anderen Herstellern geschrieben wurden.

CORBA-Anwendungen im Überblick

Wenn Sie bereits objektorientiert programmieren, erleichtert CORBA das Erstellen verteilter Anwendungen, da Remote-Objekte weitgehend wie lokale Objekte eingesetzt werden können. Das Design einer CORBA-Anwendung ähnelt stark dem anderer objektorientierter Anwendungen, enthält jedoch eine zusätzliche Schicht für die Netzwerkkommunikation, damit ein Objekt auch auf einem anderen Rechner genutzt werden kann. Die Verarbeitung dieser zusätzlichen Schicht erfolgt durch spezielle Objekte, die sogenannten *Stubs* und *Skeletons*.

Abbildung 28.1 Die Struktur einer CORBA-Anwendung



Für CORBA-Clients fungiert der Stub als Proxy für ein Objekt, das durch denselben Prozeß, einen anderen Prozeß oder einen anderen Server implementiert sein kann. Der Client interagiert mit dem Stub wie mit jedem anderen Objekt, das eine Schnittstelle implementiert. Weitere Informationen zur Verwendung von Schnittstellen finden Sie im Abschnitt »Schnittstellen verwenden« auf Seite 3-15.

Im Gegensatz zu den meisten Objekten, über die Schnittstellen implementiert werden, verarbeitet der Stub Schnittstellenaufrufe dadurch, daß er die ORB-Software aufruft, die auf dem Client-Computer installiert ist. VisiBroker ORB verwendet einen Smart Agent (osagent), der an einer beliebigen Stelle im lokalen Netzwerk ausgeführt wird. Bei dem Smart Agent handelt es sich um einen dynamischen, verteilten Verzeichnisdienst, der einen verfügbaren Server sucht, welcher die tatsächliche Objektimplementierung zur Verfügung stellt.

Auf dem CORBA-Server gibt die ORB-Software Schnittstellenaufrufe zu einem automatisch generierten Skeleton weiter. Dieses kommuniziert über den BOA (Basic Object Adaptor) mit der ORB-Software. Über den BOA wird das Objekt durch das Skeleton beim Smart Agent registriert, gibt den Gültigkeitsbereich des Objekts an (Angabe, ob dieses auf Remote-Computern verwendet werden kann) und gibt an, wann Objekte instantiiert werden und auf Client-Anforderungen reagieren können.

Stubs und Skeletons

Stubs und Skeletons stellen den Mechanismus zur Verfügung, mit dem CORBA-Anwendungen Schnittstellenaufrufe übertragen können. Dieser Vorgang wird auch als »Marshaling« bezeichnet. Beim Marshaling laufen folgende Vorgänge ab:

- Ein Schnittstellenzeiger im Server-Prozeß wird dem Code im Client-Prozeß zur Verfügung gestellt.
- Die Argumente eines Schnittstellenaufrufs, die vom Client übergeben wurden, werden im Prozeßraum des Remote-Objekts plaziert.

Bei jedem Schnittstellenaufruf schiebt die aufrufende Funktion die Argumente auf den Stack und führt über den Schnittstellenzeiger einen Funktionsaufruf durch. Wenn sich das Objekt nicht im gleichen Prozeßraum befindet wie der Code, über den seine Schnittstelle aufgerufen wird, wird der Aufruf an einen Stub weitergeleitet, der sich im gleichen Prozeßraum befindet. Der Stub schreibt die Argumente in einen Übertragungspuffer und überträgt den Aufruf in einer bestimmten Struktur an das Remote-Objekt. Das Server-Skeleton packt diese Struktur aus, schiebt die Argumente auf den Stack und ruft die Implementierung des Objekts auf. Im wesentlichen erstellt das Skeleton den Aufruf des Clients in seinem eigenen Adreßraum neu.

Stubs und Skeletons werden bei der Definition der Objektschnittstelle automatisch für Sie generiert. Die Definitionen werden in der Unit _TLB erstellt, die beim Definieren der Schnittstelle erzeugt wurde. Sie können diese Unit anzeigen, indem Sie diese in der uses-Klausel Ihrer Implementierungs-Unit auswählen und dann *Strg-Return* drücken. Weitere Informationen zum Definieren von Objektschnittstellen finden Sie im Abschnitt »Objektschnittstellen definieren« auf Seite 28-6.

Smart Agents

Bei dem Smart Agent (osagent) handelt es sich um einen dynamischen, verteilten Verzeichnisdienst, der einen verfügbaren Server sucht, welcher die tatsächliche Objektimplementierung zur Verfügung stellt. Wenn mehrere Server zur Auswahl stehen, sorgt der Smart Agent für eine ausgeglichene Auslastung. Er schützt auch vor Server-Ausfällen, indem er versucht, den Server neu zu starten, wenn eine Verbindung fehlschlägt, oder notfalls den Server auf einem anderen Host-Rechner sucht.

Ein Smart Agent muß auf mindestens einem Host-Rechner im lokalen Netzwerk gestartet werden, wobei hier unter lokalem Netzwerk ein Netzwerk gemeint ist, in dem Rundsendenachrichten gesendet werden können. Der ORB findet einen Smart Agent über eine Rundsendenachricht. Wenn im Netzwerk mehrere Smart Agents vorhanden sind, verwendet der ORB den ersten, der antwortet. Sobald der Smart Agent gefunden wurde, verwendet der ORB für die Kommunikation mit dem Smart Agent ein Punkt-zu-Punkt-UDP-Protokoll. Für das UDP-Protokoll werden weniger Netzwerk-Ressourcen benötigt als für eine TCP-Verbindung.

Wenn in einem Netzwerk mehrere Smart Agents vorhanden sind, erkennt jeder von ihnen eine Untermenge der verfügbaren Objekte und kommuniziert mit den übrigen Smart Agents, um die Objekte zu finden, die er nicht selbst erkennen kann. Wenn ein

Smart Agent unerwartet beendet wird, erfolgt eine automatische Neuregistrierung von diesem verfolgten Objekt bei einem der übrigen Smart Agents.

Einzelheiten zum Konfigurieren und Verwenden von Smart Agents im lokalen Netzwerk finden Sie im Abschnitt »Smart Agents konfigurieren« auf Seite 28-19.

Server-Anwendungen aktivieren

Beim Starten der Server-Anwendung wird der ORB (über den Basic Object Adaptor) über die Objekte informiert, die Client-Aufrufe entgegennehmen können. Dieser Code, mit dem der ORB initialisiert und darüber informiert wird, daß der Server gestartet wurde und bereit ist, wird automatisch durch den zum Starten der CORBA-Server-Anwendung verwendeten Experten zur Anwendung hinzugefügt.

In der Regel werden CORBA-Server-Anwendungen manuell gestartet. Es ist allerdings auch möglich, die Server mit Hilfe des OAD (Object Activation Daemon) zu starten oder ihre Objekte nur dann zu instantiieren, wenn diese von Clients benötigt werden.

Voraussetzung für die Verwendung des OAD ist, daß Sie Ihre Objekte bei diesem registrieren. Dabei werden die Assoziationen zwischen den Objekten und der Server-Anwendung, mit der diese implementiert werden, in einer Datenbank namens »Implementierungsablage« gespeichert.

Sobald ein Eintrag für ein Objekt in der Implementierungsablage vorhanden ist, simuliert der OAD die zugehörige Anwendung für den ORB. Bei Anforderung des Objekts durch einen Client kontaktiert der ORB den OAD, als ob es sich bei diesem um die Server-Anwendung handeln würde. Der OAD leitet dann die Client-Anforderung an den tatsächlichen Server weiter, wobei bei Bedarf die betreffende Anwendung gestartet wird.

Weitere Informationen zum Registrieren von Objekten beim OAD finden Sie im Abschnitt »Schnittstellen beim Object Activation Daemon registrieren« auf Seite 28-11.

Dynamisches Binden von Schnittstellenaufrufen

Normalerweise erfolgt beim Aufrufen der Schnittstellen von auf dem Server befindlichen Objekten durch die CORBA-Clients ein statisches Binden. Dieser Ansatz hat viele Vorteile: schnellere Anwendungsausführung und Typprüfung beim Compilieren. Es kann allerdings vorkommen, daß erst zur Laufzeit bekannt ist, welche Schnittstelle verwendet werden soll. In diesen Fällen ist es möglich, das Binden an Schnittstellen in Delphi dynamisch zur Laufzeit vorzunehmen.

Bevor Sie das dynamische Binden nutzen können, müssen Sie die Schnittstellen mit Hilfe des Dienstprogramms idl2ir registrieren. Wie Sie hierbei vorgehen müssen, ist im Abschnitt »Schnittstellen bei der Schnittstellenablage registrieren« auf Seite 28-9 beschrieben.

Informationen zur Vorgehensweise beim dynamischen Binden in CORBA-Client-Anwendungen finden Sie im Abschnitt »Die dynamische Aufrufschnittstelle verwenden« auf Seite 28-14.

CORBA-Server schreiben

Zwei Experten auf der Seite *Multi-Tier* des Dialogfelds *Objektgalerie* dienen zum Erstellen von CORBA-Servern:

- Mit dem Experten für CORBA-Datenmodule läßt sich ein CORBA-Server für eine mehrschichtige Datenbankanwendung erstellen.
- Mit dem Experten für CORBA-Objekte läßt sich ein beliebiger CORBA-Server erstellen.

Ferner können Sie einen vorhandenen Automatisierungs-Server problemlos in einen CORBA-Server umwandeln. Hierzu klicken Sie mit der rechten Maustaste und wählen *Als CORBA-Objekt darstellen*. Durch das Umwandeln eines Automatisierungs-Servers in ein CORBA-Objekt wird eine einzelne Anwendung erstellt, die sowohl COM-Clients als auch CORBA-Clients gleichzeitig nutzen können.

CORBA-Experten

Wählen Sie zum Starten des Experten *Datei / Neu*. Das Dialogfeld *Objektgalerie* wird angezeigt. Wählen Sie die Seite *Multi-Tier*, und doppelklicken Sie auf den gewünschten Experten.

Für das CORBA-Objekt muß ein Name angegeben werden. Dies ist der Basisname einer untergeordneten Klasse zu *TCorbaDataModule* oder *TCorbaImplementation*, die von der Anwendung erstellt wird. Dies ist auch der Basisname der Schnittstelle für diese Klasse. Wenn Sie z. B. den Klassennamen *MeinObjekt* festlegen, erstellt der Experte eine neue Unit, indem er *TMeinObjekt* deklariert, mit dem die Schnittstelle *IMeinObjekt* implementiert wird.

Mit dem Experten können Sie angeben, wie die Server-Anwendung die Instanzen dieses Objekts erstellen soll. Es gibt die Möglichkeiten, die Objekte entweder gemeinsam zu verwenden oder pro Client eine eigene Instanz erstellen zu lassen.

- Wenn das Objekt gemeinsam verwendet werden soll, erstellt die Anwendung eine einzelne Instanz des Objekts, die alle Client-Anforderungen verarbeitet. Dieses Modell wird traditionell bei der Entwicklung mit CORBA verwendet. Da die alleinige Objektinstanz von allen Clients gemeinsam verwendet wird, darf sie keine persistenten Statusinformationen wie Eigenschafteneinstellungen aufweisen.
- Wenn pro Client eine Instanz verwendet werden soll, wird für jede Client-Verbindung eine neue Objektinstanz erstellt. Diese Instanz bleibt bestehen, bis ein angegebener Zeitraum abgelaufen ist, ohne daß Meldungen vom Client eingegangen sind. Bei Verwendung dieses Modells können Sie persistente Statusinformationen verwenden, weil die einzelnen Clients die Eigenschafteneinstellungen von anderen Clients nicht stören können. Client-Anwendungen müssen also regelmäßig Aufrufe an den Server übergeben, damit der definierte Zeitraum nicht abläuft.

Hinweis Das Modell mit einer Instanz pro Client ist für die CORBA-Entwicklung nicht typisch, kann aber wie das COM-Modell eingesetzt werden, bei dem die Lebensdauer eines Server-Objekts durch die Client-Nutzung gesteuert wird. Delphi kann also Server erstellen, die gleichzeitig als CORBA- und als COM-Server dienen.

Zusätzlich zum Instantiierungsmodell müssen Sie auch das Threading-Modell angeben: Single- oder Multi-Threading.

- Bei Verwendung von Single-Threading empfängt jede Objektinstanz garantiert immer nur eine Client-Anforderung. Damit ist ein sicherer Zugriff auf die Daten der Objektinstanz (Eigenschaften oder Felder) möglich. In diesem Fall müssen Sie allerdings Vorkehrungen treffen, um Thread-Konflikte bei Verwendung globaler Variablen oder Objekte zu vermeiden.
- Bei Verwendung von Multi-Threading weist jede Client-Verbindung einen eigenen dedizierten Thread (Ablaufstrang) auf. Es kann allerdings vorkommen, daß eine Anwendung auf verschiedenen Threads gleichzeitig von mehreren Clients aufgerufen wird. Sie müssen den gleichzeitigen Zugriff auf Instanzdaten sowie globalen Speicher verhindern. Das Schreiben von Multi-Thread-Servern ist problematisch, wenn eine gemeinsam verwendete Objektinstanz vorhanden ist, denn Sie müssen dann Vorkehrungen gegen Thread-Konflikte für alle in der Anwendung enthaltenen Daten und Objekte treffen.

Objektschnittstellen definieren

Mit den herkömmlichen CORBA-Werkzeugen müssen Sie die Objektschnittstellen getrennt von der Anwendung definieren. Hierzu wird die CORBA-Schnittstellensprache IDL (Interface Definition Language) verwendet. Anschließend muß ein Dienstprogramm ausgeführt werden, das ausgehend von dieser Definition Stub- und Skeleton-Code generiert. In Delphi werden Stub- und Skeleton-Code sowie IDL automatisch für Sie definiert. Das Ändern der Schnittstelle erfolgt einfach mit dem Typbibliothekseditor; die entsprechenden Quelldateien werden von Delphi automatisch aktualisiert. Weitere Informationen zum Definieren von Schnittstellen mit Hilfe des Typbibliothekseditors finden Sie in Kapitel 50, »Mit Typbibliotheken arbeiten«.

Der Typbibliothekseditor dient auch zum Definieren COM-gestützter Typbibliotheken. Aus diesem Grund enthält er viele Optionen und Steuerelemente, die für CORBA-Anwendungen nicht relevant sind. Wenn Sie versuchen, diese Optionen zu nutzen (indem Sie z. B. eine Versionsnummer oder eine Hilfedatei spezifizieren), werden die betreffenden Angaben ignoriert. Wenn Sie einen COM-Automatisierungs-Server erstellen, den Sie dann in einen CORBA-Server umwandeln, gelten diese Festlegungen für den Server für seine Rolle als Automatisierungs-Server.

Im Typbibliothekseditor können Sie die Schnittstelle mit Object Pascal oder mit der Microsoft-IDL definieren, die zum Erstellen von COM-Objekten verwendet wird. Geben Sie die gewünschte Sprache zum Definieren von Schnittstellen auf der Seite *Typbibliothek* des Dialogfelds *Umgebungsoptionen* an. Wenn Sie die IDL verwenden, müssen Sie sich darüber im klaren sein, daß die Microsoft-IDL in einigen Punkten leicht

von der CORBA-IDL abweicht. Beim Definieren der Schnittstellen können Sie nur die in der folgenden Tabelle genannten Typen verwenden:

Tabelle 28.1 In einer CORBA-Schnittstelle zulässige Typen

Typ	Bemerkungen
ShortInt	8-Bit-Integer mit Vorzeichen
Byte	8-Bit-Integer ohne Vorzeichen
SmallInt	16-Bit-Integer mit Vorzeichen
Word	16-Bit-Integer ohne Vorzeichen
Longint, Integer	32-Bit-Integer mit Vorzeichen
Cardinal	32-Bit-Integer ohne Vorzeichen
Single	4-Byte-Gleitkommawert
Double	8-Byte-Gleitkommawert
TDateTime	Wird als Double-Wert übergeben.
PWideChar	Unicode-String
String, PChar	Strings müssen in den Typ PChar umgewandelt werden.
Variante	Wird als CORBA-Typ Any übergeben. Dies ist die einzige Möglichkeit, einen Wert vom Typ Array oder Currency zu übergeben.
Boolescher Wert	Wird als CORBA_Boolean (Byte) übergeben.
Objektreferenz oder Schnittstelle	Wird als CORBA-Schnittstelle übergeben.
Aufzählungstypen	Werden als Integer übergeben.

Hinweis Anstatt den Typbibliothekseditor zu verwenden, können Sie die Schnittstelle auch direkt im Quelltexteditor hinzufügen, indem Sie dort mit der rechten Maustaste klikken und dann *Zum Interface hinzufügen* wählen. Der Typbibliothekseditor wird jedoch zum Speichern der IDL-Datei für die Schnittstelle benötigt.

Sie können keine Eigenschaften hinzufügen, die Parameter verwenden (obwohl es möglich ist, Methoden für solche Eigenschaften abzurufen und festzulegen). Einige Typen (z. B. Arrays, Int64-Werte oder Currency-Typen) müssen als Varianten definiert werden. Records werden in der Client/Server-Version von Delphi nicht unterstützt.

Die Schnittstellendefinition spiegelt sich in der automatisch generierten Stub-und-Skeleton-Unit wider. Diese Unit wird aktualisiert, wenn Sie im Typbibliothekseditor *Aktualisieren* wählen oder den Befehl *Zum Interface hinzufügen* verwenden. Diese automatisch generierte Unit wird zur uses-Klausel der Implementierungs-Unit hinzugefügt. An der Stub-und-Skeleton-Unit dürfen keine Änderungen vorgenommen werden.

Durch das Bearbeiten der Schnittstelle wird ferner die Server-Implementierungs-Unit aktualisiert; dabei werden Deklarationen für die Schnittstellenelemente hinzugefügt und leere Implementierungen für die Methoden zur Verfügung gestellt. Sie können diese Implementierungs-Unit bearbeiten, um für den Hauptteil jeder neuen Schnittstellenmethode sinnvollen Code zur Verfügung zu stellen.

Hinweis Um eine CORBA-IDL-Datei für die Schnittstelle zu speichern, klicken Sie im Typbibliothekseditor auf *Exportieren*. Geben Sie dann an, daß die Datei im CORBA-IDL-

Format und nicht im Microsoft-IDL-Format gespeichert werden soll. Verwenden Sie dann diese IDL-Datei zum Registrieren Ihrer Schnittstelle oder zum Generieren von Stubs und Skeletons für andere Programmiersprachen.

Automatisch generierter Code

Beim Definieren von CORBA-Objektschnittstellen werden automatisch zwei Unit-Dateien aktualisiert, die Ihre Schnittstellendefinitionen widerspiegeln.

Bei der ersten handelt es sich um die Stub-und-Skeleton-Unit. Ihr Name ist nach dem Muster *MeineSchnittstelle_TLB.pas* aufgebaut. Obwohl diese Unit die Stub-Klasse definiert, die nur von den Client-Anwendungen verwendet wird, enthält sie auch die Deklaration für die Schnittstellentypen und die Skeleton-Klassen. Bearbeiten Sie diese Datei nicht direkt. Diese Unit wird automatisch zur uses-Klausel der Implementierungs-Unit hinzugefügt.

Mit der Stub-und-Skeleton-Unit wird für jede vom CORBA-Server unterstützte Schnittstelle ein Skeleton-Objekt definiert. Das Skeleton-Objekt ist ein Nachkomme von *TCorbaSkeleton* und übernimmt die Verarbeitung der Einzelheiten beim Sequenzieren (Marshaling) von Schnittstellenaufrufen. Es dient nicht zur Implementierung der definierten Schnittstellen. Statt dessen übernimmt sein Konstruktor eine Schnittstelleninstanz, mit deren Hilfe alle Schnittstellenaufrufe verarbeitet werden.

Bei der zweiten aktualisierten Datei handelt es sich um die Implementierungs-Unit. Standardmäßig ist ihr Name nach dem Muster *unit1.pas* aufgebaut; es ist allerdings sinnvoll, diesen in einen aussagekräftigeren Namen zu ändern. In dieser Datei nehmen Sie die Änderungen vor.

Für jede definierte CORBA-Schnittstelle wird automatisch eine Implementierungsklassendefinition zur Implementierungs-Unit hinzugefügt. Der Name der Implementierungsklasse hängt vom Namen der Schnittstelle ab. Wenn die Schnittstelle *IMeineSchnittstelle* heißt, lautet der Name der Implementierungsklasse *TMeineSchnittstelle*. Für jede Methode, die Sie zur Schnittstelle hinzufügen, wird auch Code zur Implementierung dieser Klasse hinzugefügt. Sie müssen dann den Hauptteil dieser Methoden ausfüllen, um die Implementierungsklasse fertigzustellen.

Ferner wird Ihnen auffallen, daß Code zum Initialisierungsabschnitt der Implementierungs-Unit hinzugefügt wird. Er dient zum Erstellen eines *TCorbaFactory* -Objekts für jede Objektschnittstelle, die Sie für CORBA-Clients zur Verfügung stellen. Wenn Clients den CORBA-Server aufrufen, erstellt das CORBA-Generatorobjekt eine Instanz der Implementierungsklasse oder sucht eine solche und übergibt sie als Schnittstelle an den Konstruktor für die entsprechende Skeleton-Klasse.

Hinweis Die Verwendung von *TCorbaFactory*-Objekten, die CORBA-Server-Objekte indirekt erstellen, ist im Rahmen der CORBA-Entwicklung unüblich. Statt dessen wird das von COM-Server-Anwendungen verwendete Modell genutzt. Delphi kann deshalb Server erstellen, die gleichzeitig als COM- und als CORBA-Server dienen. Mit Hilfe der *TCorbaFactory*-Objekte können CORBA-Server ein Modell mit einer Instanz pro Client implementieren. Clients der mit Delphi erstellten CORBA-Server verwenden die Funktion *CorbaFactoryCreateStub* . Mit dieser Funktion kann das *TCorbaFactory*-

Objekt auf dem CORBA-Server zum Erstellen eines CORBA-Objekts veranlaßt werden.

Server-Schnittstellen registrieren

Es ist zwar nicht erforderlich, die Server-Schnittstellen zu registrieren, wenn nur statische Bindungen bei Client-Aufrufen von Server-Objekten erfolgen, aber das Registrieren der Schnittstellen wird empfohlen. Es kann mit den folgenden Dienstprogrammen erfolgen:

- **Die Schnittstellenablage**. Durch das Registrieren in der Schnittstellenablage können Clients die Vorteile nutzen, die das dynamische Binden mit sich bringt. Solche Bindungen ermöglichen es dem Server, auf nicht in Delphi geschriebene Clients zu reagieren, sofern diese die dynamische Aufrufschnittstelle DII (Dynamic Invocation Interface) verwenden. Weitere Informationen zur Verwendung der DII finden Sie unter »Die dynamische Aufrufschnittstelle verwenden« auf Seite 28-14. Das Registrieren bei der Schnittstellenablage ist ferner eine praktische Möglichkeit, um anderen Entwicklern beim Schreiben von Client-Anwendungen die Sicht auf Ihre Schnittstellen zu ermöglichen.
- **Der Object Activation Daemon**. Wenn die Registrierung beim Object Activation Daemon (OAD) erfolgt, ist es erst erforderlich, den Server zu starten bzw. die Objekte zu instantiieren, wenn sie wirklich von Clients benötigt werden. Damit werden Ressourcen auf dem Server-System gespart.

Schnittstellen bei der Schnittstellenablage registrieren

Sie können eine Schnittstellenablage für Ihre Schnittstellen erstellen, indem Sie den Schnittstellenablagen-Server ausführen. Zunächst müssen Sie hierzu die IDL-Datei für die Schnittstelle speichern. Hierzu wählen Sie *Ansicht / Typbibliothek* und klicken dann im Typbibliothekseditor auf *Exportieren*, um die Schnittstelle als CORBA-IDL-Datei zu exportieren.

Sobald eine IDL-Datei für die Schnittstelle vorhanden ist, können Sie den Schnittstellenablagen-Server ausführen. Verwenden Sie hierzu die folgende Syntax:

```
irep [-console] IRname [datei.idl]
```

Die Argumente für *irep* sind in der folgenden Tabelle beschrieben:

Tabelle 28.2 Argumente für irep

Argument	Beschreibung
-console	Startet den Schnittstellenablagen-Server als Konsolenanwendung. Standardmäßig wird der Schnittstellenablagen-Server als Windows-Anwendung ausgeführt.

Tabelle 28.2 Argumente für irep (Fortsetzung)

Argument	Beschreibung
IRname	Name der Schnittstellenablage. Während der Server ausgeführt wird, verwenden die Clients diesen Namen, um eine Bindung zur Schnittstellenablage herzustellen, so daß sie die Schnittstelleninformationen für die DII erhalten bzw. zusätzliche Schnittstellen registrieren können.
Datei.IDL	Über die hier angegebene IDL-Datei wird der anfängliche Inhalt der Schnittstellenablage angegeben. Wenn Sie keinen Dateinamen angeben, ist die Ablage zunächst leer. Sie können dann Schnittstellen über die Menüs des Schnittstellenablagen-Servers oder über das Dienstprogramm idl2ir hinzufügen.

Sobald der Schnittstellenablagen-Server läuft, können Sie zusätzliche Schnittstellen hinzufügen, indem Sie *Datei / Laden* wählen und den Namen einer neuen IDL-Datei angeben. Wenn die neue IDL-Datei einen Eintrag enthält, der mit einem vorhandenen IDL-Eintrag übereinstimmt, wird die neue IDL-Datei nicht verwendet.

Sie können den aktuellen Inhalt der Schnittstellenablage jederzeit in einer IDL-Datei speichern, indem Sie *Datei / Speichern* oder *Datei / Speichern unter* wählen. Auf diese Weise können nach dem Beenden der Schnittstellenablage beim nächsten Starten die gespeicherte Datei verwenden und müssen nicht alle Änderungen an der ursprünglichen IDL-Datei erneut einbringen.

Die zusätzlichen Schnittstellen können auch über das Dienstprogramm idl2ir bei der Schnittstellenablage registriert werden. Wenn der Schnittstellenablagen-Server läuft, starten Sie das Dienstprogramm idl2ir durch Verwendung der folgenden Syntax:

```
idl2ir [-ir IRname] {-replace} datei.idl
```

Die Argumente für idl2ir sind in der folgenden Tabelle beschrieben:

Tabelle 28.3 Argumente für idl2ir

Argument	Beschreibung
-ir IRname	Weist das Dienstprogramm an, eine Bindung mit der Schnittstellenablageninstanz IRname herzustellen. Wenn dieses Argument nicht angegeben wird, erfolgt das Binden von idl2ir an eine beliebige Schnittstellenablage, die der Smart Agent liefert.
-replace	Weist das Dienstprogramm an, die Einträge der Schnittstellenablage durch die entsprechenden Einträge in der Datei datei.idl zu ersetzen. Wenn -replace nicht angegeben wird, wird die gesamte Schnittstelle zur Ablage hinzugefügt, es sei denn, diese enthält bereits übereinstimmende Einträge. In diesem Fall wird die gesamte IDL-Datei zurückgewiesen. Wenn -replace angegeben wird, werden nicht übereinstimmende Einträge zurückgewiesen.
datei.idl	Gibt die IDL-Datei an, welche die Aktualisierungen für die Schnittstellenablage enthält.

Während der Schnittstellenablagen-Server läuft, können keine Einträge aus der Schnittstellenablage entfernt werden. Hierzu müssen Sie den Schnittstellenablagen-Server beenden, eine neue IDL-Datei generieren, dann den Server erneut starten und dabei die aktualisierte IDL-Datei angeben.

Schnittstellen beim Object Activation Daemon registrieren

Vor dem Registrieren einer Schnittstelle beim Object Activation Daemon (OAD) muß das OAD-Befehlszeilenprogramm auf zumindest einem Computer im lokalen Netzwerk ausgeführt werden. Starten Sie den OAD durch Eingabe der folgenden Syntax:

```
oad [optionen]
```

Für das Dienstprogramm OAD können die folgenden Befehlszeilenargumente verwendet werden:

Tabelle 28.4 Argumente für OAD

Argument	Beschreibung
-v	Schaltet den Modus für ausführliche Anzeige (Verbose-Modus) ein.
-f	Legt fest, daß der Prozeß nicht fehlschlagen darf, wenn ein weiterer OAD auf diesem Host-Rechner läuft.
-t<n>	Gibt an, wie viele Sekunden der OAD darauf wartet, bis ein erzeugter Server das angeforderte Objekt aktiviert. Das Standard-Zeitlimit beträgt 20 Sekunden. Wenn Sie diesen Wert auf 0 setzen, wartet der OAD unendlich lange. Wenn der erzeugte Server-Prozeß das angeforderte Objekt nicht innerhalb des Zeitlimits aktiviert, beendet der OAD den Server-Prozeß und gibt eine Exception zurück.
-C	Ermöglicht das Ausführen des OAD im Konsolen-Modus, wenn dieser als NT-Dienst installiert wurde.
-k	Legt fest, daß der untergeordnete Prozeß eines Objekts abgebrochen werden muß, sobald keines der zugehörigen Objekte mehr beim OAD registriert ist.
-?	Dient zur Anzeige dieser Argumente.

Sobald der OAD ausgeführt wird, können Sie die Objektschnittstellen über das Befehlszeilenprogramm oadutil registrieren. Zunächst müssen Sie die IDL-Datei für die Schnittstellen exportieren. Klicken Sie hierzu im Typbibliothekseditor auf *Exportieren*, und speichern Sie die Schnittstellendefinitionen als CORBA-IDL-Datei.

Registrieren Sie als nächstes die Schnittstellen mit Hilfe des Programms oadutil. Geben Sie die folgende Syntax ein:

```
oadutil reg [optionen]
```

Zum Registrieren von Schnittstellen mit oadutil stehen die folgenden Argumente zur Verfügung:

Tabelle 28.5 Argumente für oadutil reg

Argument	Beschreibung
-i <Schnittstellenname>	Gibt einen bestimmten IDL-Schnittstellennamen an. Zum Registrieren einer Schnittstelle müssen Sie diese entweder über diese Option oder über -r angeben.
-r <Ablage-id>	Gibt eine bestimmte Schnittstelle über deren Ablage-ID an. Dabei handelt es sich um eine eindeutige Kennung, die der Schnittstelle zugeordnet ist. Zum Registrieren einer Schnittstelle müssen Sie diese entweder über diese Option oder über -i angeben.
-o <Objektname>	Gibt den Namen des Objekts an, das die Schnittstelle unterstützt. Diese Option ist obligatorisch.

Tabelle 28.5 Argumente für oadutil reg (Fortsetzung)

Argument	Beschreibung
-cpp <Dateiname>	Gibt des Namen der ausführbaren Server-Datei an. Diese Option ist obligatorisch.
-host <Host-Name>	Gibt einen entfernten Host-Rechner an, auf dem der OAD ausgeführt wird (optional).
-verbose	Startet das Dienstprogramm im ausführlichen Modus (Verbose-Modus). Alle Nachrichten werden an stdout gesendet (optional).
-d<Referenzdaten>	Gibt Referenzdaten an, die beim Aktivieren an die Server-Anwendung übergeben werden (optional).
-a arg1	Gibt Befehlszeilenargumente an, die an die Server-Anwendung übergeben werden sollen. Durch mehrfache Eingabe von -a können mehrere Argumente übergeben werden (optional).
-e env1	Gibt Umgebungsvariablen an, die an die Server-Anwendung übergeben werden sollen. Durch mehrfache Eingabe von -e können mehrere Argumente übergeben werden (optional).

Beispiel: Mit der folgenden Zeile wird eine Schnittstelle anhand ihrer Ablage-ID registriert:

```
oadutil reg -r IDL:MyServer/MyObjectFactory:1.0 -o TMyObjectFactory -cpp MyServer.exe -p
unshared
```

Hinweis Die Ablage-ID für Ihre Schnittstelle können Sie ermitteln, indem Sie sich den Code ansehen, der zum Initialisierungsabschnitt der Implementierungs-Unit hinzugefügt wurde. Die ID erscheint als drittes Argument des Aufrufs von *TCorbaFactory.Create*.

Wenn eine Schnittstelle nicht mehr verfügbar ist, muß ihre Registrierung aufgehoben werden. Ein nicht mehr registriertes Objekt kann nicht mehr automatisch vom OAD aktiviert werden, wenn es von einem Client angefordert wird. Die Registrierung eines Objekts kann nur aufgehoben werden, wenn es zuvor mit oadutil reg registriert wurde.

Geben Sie zum Aufheben der Registrierung von Schnittstellen die folgende Syntax ein:

```
oadutil unreg [optionen]
```

Zum Aufheben der Registrierung von Schnittstellen mit oadutil stehen die folgenden Argumente zur Verfügung:

Tabelle 28.6 Argumente für oadutil unreg

Argument	Beschreibung
-i <Schnittstellenname>	Gibt einen bestimmten IDL-Schnittstellennamen an. Zum Aufheben der Registrierung einer Schnittstelle müssen Sie diese entweder über diese Option oder über -r angeben.
-r <Ablage-id>	Gibt eine bestimmte Schnittstelle über deren Ablage-ID an. Dabei handelt es sich um eine eindeutige Kennung, die der Schnittstelle beim Registrieren bei der Schnittstellenablage zugeordnet wurde. Zum Aufheben der Registrierung einer Schnittstelle müssen Sie diese entweder über diese Option oder über -i angeben.

Tabelle 28.6 Argumente für oadutil unreg (Fortsetzung)

Argument	Beschreibung
-o <Objektname>	Gibt den Namen des Objekts an, das die Schnittstelle unterstützt. Wenn Sie keinen Objektnamen angeben, wird die Registrierung aller Objekte, welche die angegebene Schnittstelle unterstützen, aufgehoben.
-host <Host-Name>	Gibt einen entfernten Host-Rechner an, auf dem der OAD ausgeführt wird (optional).
-verbose	Startet das Dienstprogramm im ausführlichen Modus (Verbose-Modus). Alle Nachrichten werden an stdout gesendet (optional).
-version	Zeigt die Versionsnummer von oadutil an.

CORBA-Clients schreiben

Beim Schreiben eines CORBA-Clients besteht der erste Schritt darin, die Kommunikation der Client-Anwendung mit der ORB-Software auf dem Client-Computer sicherzustellen. Hierzu fügen Sie einfach CorbaInit in die uses-Klausel der Unit-Datei ein. Nun fahren Sie mit dem Schreiben der Anwendung wie bei jeder anderen Anwendung in Delphi fort. Wenn Sie allerdings Objekte verwenden möchten, die in der Server-Anwendung definiert sind, werden Sie nicht direkt mit einer Objektinstanz arbeiten, sondern eine Schnittstelle für das Objekt abrufen und diese verwenden. Zum Abrufen einer Schnittstelle gibt es zwei Möglichkeiten (für statisches oder dynamisches Binden).

Wenn Sie mit statischem Binden arbeiten möchten, müssen Sie zur Client-Anwendung eine Stub-und-Skeleton-Unit hinzufügen. Diese wird automatisch erzeugt, wenn Sie die Server-Schnittstelle speichern. Das statische Binden erfolgt schneller als das dynamische Binden und bietet weitere Vorteile wie die Typenprüfung und Code-Fertigstellung bei der Compilierung.

Es gibt jedoch Fälle, in denen Sie bis zur Laufzeit noch nicht wissen, welche Objekte oder Schnittstellen Sie verwenden werden. In diesen Fällen können Sie mit dynamischem Binden arbeiten. Beim dynamischen Binden wird keine Stub-Unit benötigt, es müssen aber alle entfernten Objektschnittstellen bei einer Schnittstellenablage registriert sein, die auf dem lokalen Netzwerk ausgeführt wird.

Tip Wenn Sie CORBA-Clients für Server schreiben, die nicht in Delphi geschrieben wurden, ist die Verwendung des dynamischen Bindens empfehlenswert. Auf diese Weise brauchen Sie keine eigene Stub-Klasse zum Marshaling der Schnittstellenaufrufe zu schreiben.

Stubs verwenden

Stub-Klassen werden automatisch generiert, wenn Sie eine CORBA-Schnittstelle definieren. Sie werden in einer Stub- und-Skeleton-Unit definiert, deren Namen das Muster *BaseName_TLB* aufweist (in einer Datei mit einem Namen nach dem Muster *BaseName_TLB.pas*).

Beim Schreiben eines CORBA-Clients, müssen Sie den Code in der Stub-und-Skeleton-Unit nicht bearbeiten. Fügen Sie diese Unit in die uses-Klausel der Unit ein, die eine Schnittstelle für ein auf dem CORBA-Server befindliches Objekt benötigt. Die Stub-und-Skeleton-Unit weist für jedes Server-Objekt eine Schnittstellendefinition und eine Klassendefinition für eine entsprechende Stub-Klasse auf. Wenn z. B. für den Server eine Objektklasse *TServerObj* definiert ist, enthält die Stub-und-Skeleton-Unit eine Definition für die Schnittstelle *IServerObj* und für die Stub-Klasse *TServerObjStub*. Die Stub-Klasse ist eine der Klasse *TCorbaDispatchStub* untergeordnete Klasse; die Implementierung der entsprechenden Schnittstelle erfolgt durch Sequenzierung (Marshaling) von Aufrufen an den CORBA-Server. Zusätzlich zur Stub-Klasse ist in der Stub-und-Skeleton-Unit eine Stub-Generierungsklasse für jede Schnittstelle definiert. Diese Stub-Generierungsklasse wird nie instantiiert: sie definiert eine einzelne Klassenmethode.

In der Client-Anwendung erstellen Sie Instanzen der Stub-Klasse nicht direkt, wenn Sie eine Schnittstelle für das gewünschte Objekt auf dem CORBA-Server benötigen. Statt dessen rufen Sie die Klassenmethode *CreateInstance* der Stub-Generierungsklasse auf. Für diese Methode wird ein Argument, ein optionaler Instanzenname, benötigt; sie liefert eine Schnittstelle zur Objektinstanz auf dem Server. Beispiel:

```
var
  ObjInterface : IServerObj;
begin
  ObjInterface := TServerObjFactory.CreateInstance('');
  ...
end;
```

Beim Aufrufen von *CreateInstance* geschieht folgendes:

1. Die Methode ruft eine Schnittstelleninstanz vom ORB ab.
2. Sie verwendet diese Schnittstelle, um eine Instanz der Stub-Klasse zu erstellen.
3. Sie gibt als Ergebnis die Schnittstelle zurück.

Hinweis Wenn Sie einen Client für einen CORBA-Server schreiben, der nicht mit Delphi geschrieben wurde, müssen Sie eine eigene untergeordnete Klasse von *TCorbaStub* erstellen, welche für den Client die Unterstützung für die Übertragung (Marshaling) bietet. Diese Stub-Klasse muß dann beim globalen *CORBAStubManager* registriert werden. Schließlich müssen Sie zum Instantiieren der Stub-Klasse und zum Abrufen der Server-Schnittstelle die globale Prozedur *BindStub* aufrufen, um eine Schnittstelle abzurufen, die dann an die Methode *CreateStub* des CORBA-Stub-Managers übergeben werden kann.

Die dynamische Aufrufschnittstelle verwenden

Die dynamische Aufrufschnittstelle (DII) ermöglicht es den Client-Anwendungen, Server-Objekte aufzurufen, ohne daß hierzu eine Stub-Klasse benötigt wird, über welche die explizite Übertragung der Schnittstellenaufrufe (Marshaling) erfolgt. Da DII alle Typinformationen vor dem Senden einer Anforderung durch den Client codieren und auf dem Server wieder decodieren muß, ist dieses Verfahren langsamer als die Verwendung einer Stub-Klasse.

Bevor Sie die DII nutzen können, müssen die betreffenden Server-Schnittstellen bei einer Schnittstellenablage registriert werden, die im lokalen Netzwerk ausgeführt wird. Weitere Informationen hierzu finden Sie im Abschnitt »Schnittstellen bei der Schnittstellenablage registrieren« auf Seite 28-9.

Voraussetzung für die Verwendung der DII in einer Client-Anwendung ist, daß Sie eine Serverschnittstelle erhalten und einer Variablen vom Typ *TAny* zuweisen. *TAny* ist eine spezielle, CORBA-spezifische Variante. Anschließend müssen Sie die Methoden der Schnittstelle aufrufen, wobei Sie die *TAny*-Variable verwenden, als würde es sich dabei um eine Schnittstelleninstanz handeln. Der Compiler übernimmt die einzelnen Schritte, die zum Umwandeln der Aufrufe in DII-Anforderungen notwendig sind.

Schnittstelle abrufen

Zum Abrufen einer Schnittstelle für spät gebundene DII-Aufrufe verwenden Sie die globale Funktion *CorbaBind*. *CorbaBind* verwendet entweder die Ablagen-ID des Server-Objekts oder einen Schnittstellentyp zum Anfordern einer Schnittstelle vom ORB. Diese wird dann zum Erstellen eines Stub-Objekts verwendet.

Hinweis Vor dem Aufrufen von *CorbaBind* muß die Zuordnung von Schnittstellentyp zu Ablagen-ID beim globalen *CorbaInterfaceIDManager* registriert werden.

Wenn für Ihre Client-Anwendung eine registrierte Stub-Klasse für den Schnittstellentyp vorhanden ist, erstellt *CorbaBind* einen Stub dieser Klasse. In diesem Fall kann die von *CorbaBind* zurückgegebene Schnittstelle sowohl für frühes Binden (Umwandlung mit dem Operator as) oder für spätes Binden (mit DII) verwendet werden. Wenn keine registrierte Stub-Klasse für den Schnittstellentyp vorhanden ist, liefert *CorbaBind* die Schnittstelle für ein generisches Stub-Objekt. Solche Objekte können nur für spätes Binden (mit DII) verwendet werden.

Damit die von *CorbaBind* gelieferte Schnittstelle für DII-Aufrufe verwendet werden kann, muß sie einer Variablen vom Typ *TAny* zugewiesen werden:

```
var
  IntToCall: TAny;
begin
  IntToCall := CorbaBind('IDL:MyServer/MyServerObject:1.0');
  ...
```

Schnittstellen mit der DII aufrufen

Sobald eine Schnittstelle einer Variablen vom Typ *TAny* zugewiesen wurde, braucht zu ihrem Aufruf mit der DII nur diese Variable so verwendet zu werden, als ob es sich dabei um eine Schnittstelle handeln würde:

```
var
  HR, Emp, Payroll, Salary: TAny;
begin
  HR := CorbaBind('IDL:CompanyInfo/HR:1.0');
  Emp := HR.LookupEmployee(Edit1.Text);
  Payroll := CorbaBind('IDL:CompanyInfo/Payroll:1.0');
  Salary := Payroll.GetEmployeeSalary(Emp);
```

```
    Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit2.Text) / 100));
end;
```

Bei Verwendung der DII müssen Sie zur Angabe der Schnittstellenmethode immer die Groß-/Kleinschreibung beachten. Im Gegensatz zu Aufrufen mit statischem Binden ist hier sicherzustellen, daß die Schreibweise der Methodennamen mit der in der Schnittstellendefinition exakt übereinstimmt.

Beim Aufrufen einer Schnittstelle über die DII wird jeder Parameter als Wert vom Typ *TAny* behandelt. Dies liegt daran, daß in Werten vom Typ *TAny* die Typinformationen enthalten sind. Diese ermöglichen es dem Server, beim Empfang des Aufrufs die Typinformationen zu interpretieren.

Da Parameter immer als *TAny*-Werte behandelt werden, ist es nicht erforderlich, diese in die geeigneten Parametertypen umzuwandeln. Im vorherigen Beispiel wäre es möglich, für den letzten Parameter im Aufruf von *SetEmployeeSalary* anstelle eines Gleitkommawerts einen String zu übergeben:

```
Payroll.SetEmployeeSalary(Emp, Edit2.Text);
```

Es ist immer möglich, einfache Typen direkt als Parameter zu übergeben; sie werden dann beim Compilieren in *TAny*-Werte umgewandelt. Bei strukturierten Datentypen müssen zum Erstellen des geeigneten *TAny*-Typs die Umwandlungsverfahren der globalen ORB-Variablen verwendet werden. Die folgende Tabelle enthält eine Übersicht über die Methoden, die zum Erzeugen der verschiedenen strukturierten Datentypen verwendet werden müssen:

Tabelle 28.7 ORB-Methoden zum Erzeugen strukturierter TAny-Werte

Strukturierter Datentyp	Hilfsfunktion
Record	MakeStructure
Array (feste Länge)	MakeArray
Dynamisches Array (Sequenz)	MakeSequence

Bei Verwendung dieser Hilfsfunktionen müssen Sie den Typcode angeben, der dem zu erstellenden Typ von Record, Array oder dynamischem Array entspricht. Diesen Typ können Sie dynamisch über eine Ablagen-ID aufrufen, indem Sie die ORB-Methode *FindTypeCode* verwenden:

```
var
  HR, Name, Emp, Payroll, Salary: TAny;
begin
  with ORB do
  begin
    HR := Bind('IDL:CompanyInfo/HR:1.0');
    Name := MakeStructure(FindTypeCode('IDL:CompanyInfo/EmployeeName:1.0',
                                       [Edit1.Text,Edit2,Text]);
    Emp := HR.LookupEmployee(Name);
    Payroll := Bind('IDL:CompanyInfo/Payroll:1.0');
  end;
  Salary := Payroll.GetEmployeeSalary(Emp);
  Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit3.Text) / 100));
end;
```

CORBA-Anwendungen anpassen

Mit den beiden globalen Funktionen *ORB* und *BOA* läßt sich die Art der Interaktion Ihrer Anwendung mit der CORBA-Software, die im Netzwerk ausgeführt wird, anpassen.

Client-Anwendungen verwenden den von ORB zurückgegebenen Wert zum Konfigurieren der ORB-Software, zum Beenden der Verbindung zum Server, zum Binden mit Schnittstellen und zum Abrufen von String-Darstellungen von Objekten zum Anzeigen, um Objektnamen in der Benutzeroberfläche anzeigen zu können.

Server-Anwendungen verwenden den von BOA zurückgegebenen Wert zum Konfigurieren der BOA-Software, zum Bereitstellen bzw. Deaktivieren von Objekten und zum Abrufen von benutzerdefinierten Informationen, die einem Objekt von einer Client-Anwendung zugewiesen wurden.

Objekte in der Benutzeroberfläche anzeigen

Beim Schreiben einer CORBA-Client-Anwendung ist es unter Umständen wünschenswert, den Benutzern die Namen der verfügbaren CORBA-Server-Objekte anzuzeigen. Hierzu müssen Sie die Objektschnittstelle in einen String umwandeln. Hierzu dient die Methode *ObjectToString* der globalen *ORB*-Variablen. Im folgenden Beispiel werden die Namen von drei Objekten in einem Listenfeld angezeigt, wobei von den Schnittstelleninstanzen für die zugehörigen Stub-Objekte ausgegangen wird.

```
var
  Dept1, Dept2, Dept3: IDepartment;
  begin
  Dept1 := TDepartmentFactory.CreateInstance('Vertrieb');
  Dept1.SetDepartmentCode(120);
  Dept2 := TDepartmentFactory.CreateInstance('Marketing');
  Dept2.SetDepartmentCode(98);
  Dept3 := TSecondFactory.CreateInstance('Buchhaltung');
  Dept3.SetDepartmentCode(49);
  ListBox1.Items.Add(ORB.ObjectToString(Dept1));
  ListBox1.Items.Add(ORB.ObjectToString(Dept2));
  ListBox1.Items.Add(ORB.ObjectToString(Dept3));
end;
```

Der Vorteil beim Erzeugen der Strings für die Objekte durch den ORB liegt darin, daß Sie diese Prozedur mit Hilfe der Methode *StringToObject* umkehren können:

```
var
  Dept: IDepartment;
begin
  Dept := ORB.StringToObject(ListBox1.Items[ListBox1.ItemIndex]);
  ... { Vorgang mit der ausgewählten Abteilung durchführen }
```

CORBA-Objekte bereitstellen und deaktivieren

Wenn eine CORBA-Server-Anwendung eine Objektinstanz erstellt, kann sie diese über die Methode *ObjIsReady* der von BOA zurückgegebenen globalen Variable für die Clients zur Verfügung stellen.

Jedes mit *ObjIsReady* bereitgestellte Objekt kann von der Server-Anwendung auch wieder deaktiviert werden. Hierzu wird die BOA-Methode *Deactivate* verwendet.

Wenn der Server ein Objekt deaktiviert, kann dadurch die von einer Client-Anwendung verwendete Objektschnittstelle ungültig werden. Client-Anwendungen können dies erkennen, indem Sie die Methode *NonExistent* für das Stub-Objekt aufrufen. Diese liefert den Wert *True*, wenn das Server-Objekt deaktiviert wurde und den Wert *False*, wenn das Server-Objekt noch verfügbar ist.

Client-Informationen an Server-Objekte übergeben

Stub-Objekte in CORBA-Clients können dem zugehörigen Server-Objekt über ein *TCorbaPrincipal*-Objekt Identifikationsinformationen senden. Unter einem *TCorbaPrincipal*-Objekt versteht man ein Byte-Array, das Informationen über die betreffende Client-Anwendung enthält. Für Stub-Objekte kann dieser Wert über die Methode *SetPrincipal* gesetzt werden.

Sobald der CORBA-Client Identifizierungsdaten in die Instanz des Server-Objekts geschrieben hat, kann das Server-Objekt über die BOA-Methode *GetPrincipal* auf diese Informationen zugreifen.

Da es sich bei *TCorbaPrincipal* um ein Byte-Array handelt, können darin jegliche Informationen transportiert werden, die der Entwickler für sinnvoll hält. So können z. B. Clients mit speziellen Privilegien einen Schlüsselwert senden, den der Server überprüft, bevor er den Zugriff auf bestimmte Methoden ermöglicht.

CORBA-Anwendungen weitergeben

Nachdem Sie die Client- bzw. Server-Anwendungen erstellt und gründlich getestet haben, können Sie die Client-Anwendungen an die Desktop-Computer der Endbenutzer und die Server-Anwendungen an die Server-Computer weitergeben. Die folgende Liste beschreibt, welche Dateien zusätzlich zur Client- bzw. Server-Anwendung installiert werden müssen, damit die CORBA-Anwendung weitergegeben werden kann.

- Die ORB-Bibliotheken müssen auf jedem Client- und Server-Computer installiert sein. Sie befinden sich im Bin-Unterverzeichnis des Installationsverzeichnisses von VisiBroker.
- Wenn Ihr Client die dynamische Aufrufschnittstelle (DII) verwendet, muß auf mindestens einem Host-Rechner im lokalen Netzwerk ein Schnittstellenablagen-Server laufen, wobei hier mit lokalem Netzwerk ein Netzwerk gemeint ist, in dem Rundsendenachrichten gesendet werden können. Informationen zum Ausführen

des Schnittstellenablagen-Servers und zum Registrieren der Schnittstellen finden Sie unter »Schnittstellen bei der Schnittstellenablage registrieren« auf Seite 27-9.

- Wenn der Server bei Bedarf gestartet werden soll, muß der Object Activation Daemon (OAD) auf mindestens einem Host-Rechner im lokalen Netzwerk laufen. Informationen zum Ausführen des OAD und zum Registrieren der Schnittstellen finden Sie im Abschnitt »Schnittstellen beim Object Activation Daemon registrieren« auf Seite 28-11.
- Ein Smart Agent (osagent) muß auf mindestens einem Host-Rechner im lokalen Netzwerk installiert sein. Es ist möglich, den Smart Agent an mehrere Computer weiterzugeben.

Zusätzlich müssen Sie zum Weitergeben der CORBA-Anwendung unter Umständen die folgenden Umgebungsvariablen setzen:

Tabelle 28.8 CORBA-Umgebungsvariablen

Variable	Bedeutung
PATH	Hiermit stellen Sie sicher, daß das Verzeichnis, in dem die ORB-Bibliotheken gespeichert sind, in der Pfadangabe enthalten ist.
VBROKER_ADM	Gibt das Verzeichnis mit den Konfigurationsdateien für die Schnittstellenablage, den Object Activation Daemon und den Smart Agent an.
OSAGENT_ADDR	Gibt die IP-Adresse des Host-Rechners an, dessen Smart Agent verwendet werden soll. Wenn Sie diese Variable nicht setzen, verwendet die CORBA-Anwendung den ersten Smart Agent, der auf eine Rundsendenachricht antwortet.
OSAGENT_PORT	Gibt den Anschluß an, an dem der Smart Agent auf Anforderungen wartet.
OSAGENT_ADDR_FILE	Gibt den vollständigen Pfadnamen zu einer Datei an, die Smart Agent-Adressen in anderen lokalen Netzwerken enthält.
OSAGENT_LOCAL_FILE	Gibt den vollständigen Pfadnamen zu einer Datei an, die Netzwerkinformationen über einen Smart Agent enthält, der auf einem Host läuft, für den mehrere IP-Adressen definiert sind.
VBROKER_IMPL_PATH	Gibt das Verzeichnis für die Implementierungsablage an (in welcher der OAD seine Informationen speichert).
VBROKER_IMPL_NAME	Gibt den Standarddateinamen für die Implementierungsablage an.

Hinweis Allgemeine Informationen zum Weitergeben von Anwendungen finden Sie in Kapitel 11, »Anwendungen weitergeben«.

Smart Agents konfigurieren

Wenn Sie eine CORBA-Anwendung weitergeben, muß im lokalen Netzwerk mindestens ein Smart Agent laufen. Durch Weitergabe mehrerer Smart Agents in einem lokalen Netzwerk vermeiden Sie die Probleme, die bei einem Systemabsturz des Computers entstehen, auf dem der Smart Agent läuft.

Durch Weitergabe von Smart Agents wird ein lokales Netzwerk in getrennte ORB-Domänen unterteilt. Es besteht umgekehrt aber auch die Möglichkeit, Smart Agents

in verschiedenen lokalen Netzwerken zu verbinden, um damit die ORB-Domäne zu vergrößern.

Einen Smart Agent starten

Zum Starten des Smart Agent führen Sie das Dienstprogramm osagent aus. Auf mindestens einem Host-Rechner im lokalen Netzwerk muß ein Smart Agent laufen. Für das Dienstprogramm osagent können die folgenden Befehlszeilenargumente eingegeben werden:

Tabelle 28.9 Argumente für osagent

Argument	Beschreibung
-v	Schaltet den Modus für ausführliche Anzeige (Verbose-Modus) ein. Informationen und Diagnosenachrichten werden in eine Protokolldatei namens osagent.log geschrieben, die sich in dem Verzeichnis befindet, das über die Umgebungsvariable VBROKER_ADM angegeben ist.
-p<n>	Gibt den UDP-Anschluß an, an dem der Smart Agent auf Rundsendenachrichten wartet.
-C	Ermöglicht das Ausführen des Smart Agent im Konsolen-Modus, wenn dieser als NT-Dienst installiert wurde.

Geben Sie beispielsweise den folgenden Befehl in einem DOS-Fenster ein, oder klikken Sie die Schaltfläche *Start* an, und wählen Sie *Ausführen*:

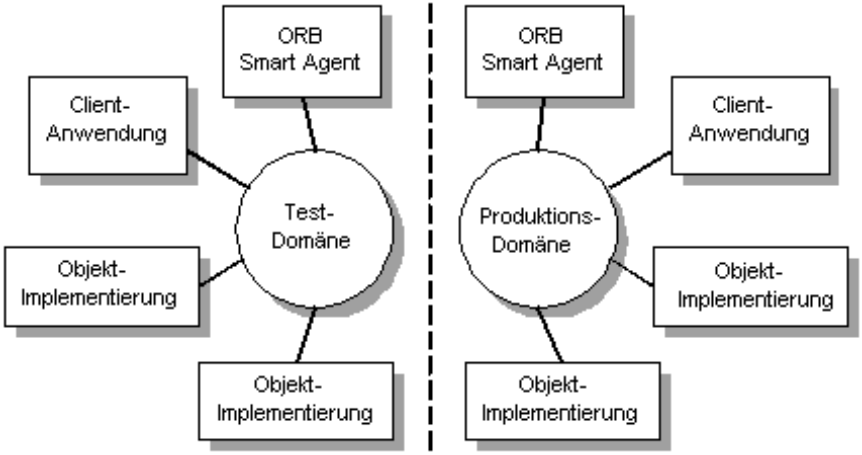
```
osagent -p 11000
```

Damit wird der Smart Agent so gestartet, daß er am UDP-Anschluß 11000 und nicht am Standardanschluß (14000) empfangsbereit ist. Durch Wechseln des Anschlusses, an dem der Smart Agent auf Rundsendenachrichten wartet, können Sie mehrere ORB-Domänen erstellen.

ORB-Domänen konfigurieren

Oft ist es wünschenswert, über zwei oder mehr getrennte, gleichzeitig laufende ORB-Domänen zu verfügen. Eine Domäne könnte beispielsweise die fertige Version der Client-Anwendung und der Objektimplementierungen enthalten, während in der anderen Testversionen derselben Clients sowie Objekte enthalten sein könnten, die noch nicht zur allgemeinen Verwendung freigegeben worden sind. Wenn mehrere Entwickler im gleichen lokalen Netzwerk arbeiten, ist die Einrichtung getrennter ORB-Domänen für sie sinnvoll, so daß die Testaktivitäten der verschiedenen Entwickler sich nicht gegenseitig stören.

Abbildung 28.2 Getrennte ORB-Domänen



Die unterschiedlichen ORB-Domänen innerhalb desselben Netzwerks lassen sich durch eindeutige UDP-Anschlußnummern für die einzelnen osagents in den Domänen unterscheiden.

Die Standard-Anschlußnummer (14000) wird bei der Installation des ORB in die Windows-Registrierung geschrieben. Wenn Sie diesen Wert außer Kraft setzen möchten, ändern Sie die Umgebungsvariable OSAGENT_PORT entsprechend. Ferner können Sie den für OSAGENT_PORT festgelegten Wert umgehen, indem Sie den Smart Agent mit der Option -p starten.

Smart Agents in verschiedenen lokalen Netzwerken verbinden

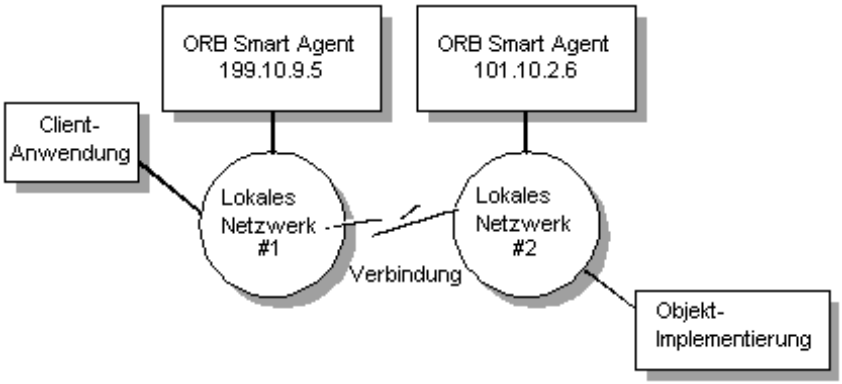
Wenn Sie in Ihrem lokalen Netzwerk mehrere Smart Agents starten, werden diese einander über UDP-Rundsendenachrichten finden. Die Konfiguration ihrer lokalen Netzwerke erfolgt durch Angabe des Bereichs für Rundsendenachrichten über die IP-Teilnetzmaske. Sie können einem Smart Agent das Kommunizieren mit anderen Netzwerken ermöglichen. Hierzu gibt es zwei Möglichkeiten:

- Verwendung einer agentaddr-Datei.
- Verwendung eines Hosts mit mehreren IP-Adressen.

Eine agentaddr-Datei verwenden

Betrachten Sie die beiden in der folgenden Abbildung dargestellten Smart Agents. Der Smart Agent in Netzwerk 1 wartet auf Rundsendenachrichten und verwendet hierzu die IP-Adresse 199.10.9.5, der Smart Agent in Netzwerk 2 verwendet die IP-Adresse 101.10.2.6.

Abbildung 28.3 Zwei Smart Agents in getrennten lokalen Netzwerken



Der Smart Agent in Netzwerk 1 kann mit dem Smart Agent in Netzwerk 2 kommunizieren, wenn er eine Datei namens agentaddr findet, welche die folgende Zeile enthält:

```
101.10.2.6
```

Der Smart Agent sucht diese Datei in dem Verzeichnis, das in der Umgebungsvariablen VBROKER_ADM angegeben ist.

Einen Host mit mehreren IP-Adressen verwenden

Bei Ausführung des Smart Agent auf einem Host-Rechner, für den mehrere IP-Adressen definiert sind, kann der Agent einen leistungsfähigen Mechanismus zur Verfügung stellen, mit dem eine Brücke zwischen Objekten erstellt wird, die sich in verschiedenen lokalen Netzwerken befinden. Alle lokalen Netzwerke, an die der Host angeschlossen ist, können mit einem einzigen Smart Agent kommunizieren, der ohne Verwendung einer agentaddr-Datei eine Brücke zwischen den lokalen Netzwerken bildet.

Auf einem Host mit mehreren IP-Adressen kann der Smart Agent jedoch die korrekte Teilnetzmaske und die Werte für die Rundsendeadresse nicht feststellen. Diese Werte müssen in einer localaddr-Datei angegeben werden. Sie können die zutreffenden Werte der Netzwerkschnittstelle für die localaddr-Datei ermitteln, indem Sie die TCP/IP-Protokolleigenschaften über das Symbol *Netzwerk* in der Systemsteuerung aufrufen. Wenn Ihr Host unter Windows NT läuft, können Sie diese Werte über den Befehl *ipconfig* ermitteln.

Die Datei localaddr enthält eine Zeile für jede Kombination aus IP-Adresse, Teilnetzmaske und Rundsendeadresse, die der Smart Agent verwenden kann. Im folgenden Beispiel ist der Inhalt der Datei localaddr für einen Smart Agent aufgeführt, für den zwei IP-Adressen zur Verfügung stehen:

```
216.64.15.10 255.255.255.0 216.64.15.255
214.79.98.88 255.255.255.0 214.79.98.255
```

Außerdem müssen Sie die Umgebungsvariable OSAGENT_LOCAL_FILE auf den vollständigen Pfadnamen für die Datei localaddr setzen, damit der Smart Agent die Datei finden kann.

Kapitel

29

Internet-Server-Anwendungen

Mit Delphi können Sie Web-Server-Anwendungen als CGI-Anwendung oder als dynamische Link-Bibliothek (DLL) erstellen. Diese Web-Server-Anwendungen können jede verfügbare nichtvisuelle Komponente einsetzen. Spezielle Komponenten der Palettenseite *Internet* erleichtern das Schreiben von Ereignisbehandlungsroutinen, die mit einer bestimmten URI (Uniform Resource Identifier) verbunden sind. Nachdem die Verarbeitung abgeschlossen ist, können im Programm HTML-Dokumente erzeugt und an den Client übertragen werden.

Häufig stammen diese Informationen aus Datenbanken. Mit Hilfe der Internet-Komponenten lassen sich Verbindungen zu Datenbanken automatisch verwalten. Damit ist es möglich, in einer DLL mehrere simultane, thread-sichere Datenbankverbindungen zu verwalten.

Das vorliegende Kapitel beschreibt diese Internet-Komponenten und geht detailliert auf die Programmierung verschiedener Arten von Internet-Anwendungen ein.

Hinweis Sie können auch ActiveForms als Internet-Server-Anwendungen verwenden. Weitere Informationen zu ActiveForms finden Sie unter »ActiveX-Steuerelemente auf der Basis eines VCL-Formulars erstellen« auf Seite 48-8.

Terminologie und Standards

Viele Protokolle zur Steuerung der Aktivität im Internet sind in sogenannten RFC-Dokumenten definiert (RFC = Request for Comment), die von der Internet Engineering Task Force (IETF) erstellt, aktualisiert und verwaltet werden. Es handelt sich dabei um eine Arbeitsgruppe für Protokollentwicklung im Internet. Nachstehend finden Sie einige wichtige RFCs, die Sie bei der Programmierung von Internet-Anwendungen unterstützen:

- RFC822, (Standard für das Format von ARPA-Internet-Textbotschaften), beschreibt die Struktur und den Inhalt eines Botschafts-Headers.

- RFC1521, MIME (Multipurpose Internet Mail Extensions) Teil 1: Mechanisms for Specifying and Describing the Format of Internet Message Bodies (Mechanismen zur Festlegung und Beschreibung des Formats von Internet-Botschaften), beschreibt die Methode, mit der Multipart- und Multiformat-Botschaften gekapselt und transportiert werden.
- RFC1945, Hypertext Transfer Protocol — HTTP / 1.0, beschreibt einen Transfermechanismus, der zur Verteilung kollaborativer Hypermedia-Dokumente benutzt wird.
  Die IETF verwaltet an ihrem Web-Standort (`www.ietf.cnri.reston.va.us`) eine Bibliothek der RFCs.

Bestandteile einer URL

Die URL (Uniform Resource Locator) beschreibt den Standort einer Ressource, die über das Internet zugänglich ist. Sie setzt sich aus mehreren Bestandteilen zusammen, auf die eine Anwendung zugreifen kann. Diese Bestandteile sind in der folgenden Abbildung gezeigt:

Abbildung 29.1 Bestandteile einer URL

Im ersten Teil (der technisch gesehen nicht Bestandteil der URL ist) wird das Protokoll festgelegt (http). Hier können auch andere Protokolle wie https (secure http) oder ftp angegeben werden.

Der Abschnitt *Host* identifiziert den Rechner, auf dem der Web-Server und die Web-Server-Anwendung ausgeführt werden. In diesem Abschnitt kann der Port für den Botschaftsempfang geändert werden (in der obigen Abbildung ist dies nicht dargestellt). Normalerweise muß kein Port angegeben werden, da die Port-Nummer über das Protokoll impliziert ist.

Der Abschnitt *ScriptName* enthält den Namen der Web-Server-Anwendung. An diese Anwendung übergibt der Web-Server die Botschaften.

Nach dem Skriptnamen folgt der Abschnitt *PathInfo*. Hier wird das Ziel in der Web-Server-Anwendung definiert, an das die Botschaften gesendet werden. Dies kann ein bestimmtes Verzeichnis auf den Host-Rechner sein, es kann sich aber auch um Namen von Komponenten handeln, die auf bestimmte Botschaften antworten, oder um andere Mechanismen, mit denen die Web-Server-Anwendung eingehende Botschaften verarbeitet.

Der Abschnitt *Query* enthält einige benannter Werte. Diese Werte und deren Namen werden von der Web-Server-Anwendung definiert.

URI und URL

Die URL stellt eine Teilmenge der URI dar, die im HTTP-Standard (RFC1945) definiert ist. Web-Server-Anwendungen erzeugen ihren Inhalt häufig durch einen Zugriff auf unterschiedliche Quellen, wobei das endgültige Ergebnis nicht an einem bestimmten Standort existiert, sondern bei Bedarf generiert wird. Mit URIs können Ressourcen beschrieben werden, die keinen festen Standort besitzen.

Informationen in den Headern von HTTP-Anforderungen

HTTP-Anforderungsbotschaften enthalten viele Header, die den Client, das Ziel der Anforderung, die Art ihrer Behandlung sowie die mit der Anforderung gesendeten Inhaltsdaten beschreiben. Jeder Header wird durch einen Namen, wie z. B. »Host« identifiziert, hinter dem ein String-Wert steht. Betrachten Sie z. B. die folgende HTTP-Anforderung:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Die erste Zeile identifiziert die Anforderung als GET-Anforderung. Eine solche Anforderungsbotschaft fordert die Web-Server-Anwendung auf, den mit der URI hinter dem Wort GET verbundenen Inhalt (in diesem Fall /art/gallery.dll/animals?animal=doc&color=black) zurückzugeben. Über den letzten Teil der ersten Zeile wird angegeben, daß der Client den Standard HTTP 1.0 verwendet.

Die zweite Zeile stellt den Verbindungs-Header dar. Sie gibt an, daß die Verbindung nicht beendet werden soll, wenn die Anforderung bedient wurde. Die dritte Zeile enthält den Header für den Benutzer-Agenten und liefert Informationen darüber, welches Programm die Anforderung generiert. Die nächste Zeile enthält den Host-Header, in dem der Host-Name und -Port für den Server angegeben sind, über den die Verbindung erfolgt. Die letzte Zeile schließlich enthält den Akzeptanz-Header zur Angabe der Medientypen, die der Client als gültige Antworten akzeptiert.

HTTP-Server-Aktivitäten

Die Client/Server-Struktur von Web-Browsern scheint auf den ersten Blick einfach zu sein. Für die meisten Benutzer stellt sich das Ermitteln von Informationen im World Wide Web als einfacher Vorgang dar: Sie klicken auf einen Link, und die Information wird auf dem Bildschirm angezeigt. In der Regel reichen diese Kenntnisse für die Erstellung einfacher, seitenorientierter Web-Inhalte aus. Es gibt aber auch Anwender, die mit der Materie besser vertraut sind und umfangreichere Kenntnisse über die HTML-Syntax und die Client/Server-Struktur der verwendeten Protokolle besitzen. Für Autoren komplexerer Web-Seiten steht eine Vielzahl von Optionen zur Verfügung, mit denen sich die Präsentation von Informationen automatisieren läßt.

Vor der Erzeugung einer Web-Server-Anwendung ist es hilfreich zu verstehen, wie der Client eine Anforderung ausgibt und wie der Server mit dieser Client-Anforderung umgeht.

Client-Anforderungen zusammenstellen

Wenn ein HTML-Hypertext-Link ausgewählt wird (oder der Benutzer auf andere Art eine URL angibt), sammelt der Browser Informationen über das Protokoll, die angegebene Domäne, den Pfad zu den Daten, das Datum und die Uhrzeit, das Betriebssystem, den Browser selbst sowie weitere Inhaltsinformationen. Anschließend stellt er eine Anforderung zusammen.

Angenommen, es soll eine Seite mit Bildern angezeigt werden, die auf Kriterien basiert, welche durch Anklicken von Schaltflächen in einem Formular ausgewählt wurden. Dazu könnte der Client die folgende URL konstruieren:

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

Diese URL bezeichnet einen HTTP-Server in der Domäne www.TSite.com. Der Client nimmt Kontakt mit www.TSite.com auf, stellt die Verbindung zum HTTP-Server her und übergibt ihm eine Anforderung. Diese könnte wie folgt aussehen:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Client-Anforderungen bedienen

Der Web-Server empfängt eine Client-Anforderung und führt, abhängig von seiner Konfiguration, eine unterschiedliche Anzahl von Aktionen aus. Wenn der Server so konfiguriert ist, daß er den Anforderungsabschnitt /gallery.dll als Programm erkennt, übergibt er Anforderungsinformationen an dieses Programm. Die Art und Weise, in der Informationen über die Anforderung an das Programm übergeben werden, hängt vom Programmtyp der Web-Server-Anwendung ab:

- Wenn es sich um ein CGI-Programm (CGI = Common Gateway Interface) handelt, übergibt der Server die in der Anforderung enthaltene Information direkt an das CGI-Programm. Während das Programm ausgeführt wird, wartet der Server. Bei seiner Beendigung übergibt das CGI-Programm den Inhalt direkt an den Server.
- Bei einem Win-CGI-Programm öffnet der Server eine Datei und schreibt die Anforderungsinformationen in diese Datei. Dann wird das Win-CGI-Programm ausgeführt, wobei die Position der Datei, in der die Client-Informationen enthalten sind, und die Position einer Datei, die das Win-CGI-Programm zum Erstellen des Inhalts verwenden soll, übergeben werden. Während das Programm ausgeführt wird, wartet der Server. Sobald das Programm beendet wird, liest der Server die Daten aus der Inhaltsdatei, in die vom Win-CGI-Programm geschrieben wurde.
- Wenn es sich bei dem Programm um eine dynamische Link-Bibliothek (DLL) handelt, lädt der Server diese (falls erforderlich) und übergibt ihr die Anforderungsin-

formationen als Struktur. Während das Programm ausgeführt wird, wartet der Server. Die DLL übergibt bei Ihrer Beendigung den Inhalt direkt an den Server.

In allen Fällen reagiert das Programm auf die Anforderung und führt die vom Programmierer festgelegten Aktionen aus. Dazu gehören z. B. der Zugriff auf Datenbanken, einfache Lookup-Operationen oder Berechnungen mit Tabellendaten, die Konstruktion bzw. die Auswahl von HTML-Dokumenten usw.

Auf Client-Anforderungen antworten

Wenn eine Web-Server-Anwendung mit der Bearbeitung einer Client-Anforderung fertig ist, generiert sie eine Seite mit HTML-Code oder einem anderen MIME-Inhalt und gibt sie (über den Server) an den Client zur Anzeige zurück. Die Art und Weise, in der diese Antwort gesendet wird, hängt von der Art des Programms ab:

- Bei der Beendigung eines Win-CGI-Skripts wird eine HTML-Seite aufgebaut und in eine Datei geschrieben. Zusätzliche Antwortinformationen werden in einer anderen Datei abgelegt. Dann wird der Standort beider Dateien an den Server zurückgegeben. Der Server öffnet beide Dateien und übergibt die HTML-Seite an den Client.
- Bei der Beendigung einer DLL werden die HTML-Seite und alle Antwortinformationen direkt an den Server zurückgegeben. Dieser übergibt sie dann an den Client.

Durch die Erstellung einer Web-Server-Anwendung als DLL lassen sich Ladezeit und Ressourcenverbrauch durch das System reduzieren, da für die Bedienung einer Anforderung weniger Prozesse und Plattenzugriffe erforderlich sind.

Web-Server-Anwendungen

Durch Web-Server-Anwendungen lassen sich die Fähigkeiten und die Funktionalität vorhandener Web-Server erweitern. Die Web-Server-Anwendung empfängt HTTP-Anforderungsbotschaften vom Web-Server, führt die angeforderten Aktionen aus und formuliert Antworten, die dann an den Web-Server zurückgegeben werden. Alle Operationen, die innerhalb einer Delphi-Anwendung ausgeführt werden können, lassen sich in eine Web-Server-Anwendung einbinden.

Arten von Web-Server-Anwendungen

Mit Hilfe der Internet-Komponenten können Sie vier unterschiedliche Arten von Web-Server-Anwendungen erstellen. Jede Anwendungsart verwendet einen typspezifischen Nachkommen von *TWebApplication*, *TWebRequest* und *TWebResponse*:

Tabelle 29.1 Komponenten für Web-Server-Anwendungen

Anwendungsart	Anwendungsobjekt	Anforderungsobjekt	Antwortobjekt
Microsoft Server-DLL (ISAPI)	*TISAPIApplication*	*TISAPIRequest*	*TISAPIResponse*
Netscape Server-DLL (NSAPI)	*TISAPIApplication*	*TISAPIRequest*	*TISAPIResponse*
CGI-Konsolenanwendung	*TCGIApplication*	*TCGIRequest*	*TCGIResponse*
Windows-CGI-Anwendung	*TCGIApplication*	*TWinCGIRequest*	*TWinCGIResponse*

ISAPI und NSAPI

Eine ISAPI- oder NSAPI-Web-Server-Anwendung ist eine DLL, die vom Web-Server geladen wird. Die Informationen in der Client-Anforderung werden als Struktur an die DLL übergeben und von *TISAPIApplication* ausgewertet. Dieses Objekt erzeugt die *TISAPIRequest*- und *TISAPIResponse*-Objekte. Jede Anforderungsbotschaft wird automatisch in einem separaten Ausführungs-Thread verarbeitet.

CGI-Programme

Eine eigenständige CGI-Web-Server-Anwendung ist eine Konsolenanwendung, die Client-Anforderungen über die Standardeingabe empfängt und die Ergebnisse über die Standardausgabe an den Server zurückgibt. Die Daten werden von *TCGIApplication* ausgewertet. Dieses Objekt erzeugt die *TCGIRequest*- und *TCGIResponse*-Objekte. Jede Anforderungsbotschaft wird von einer gesonderten Instanz der Anwendung bearbeitet.

Win-CGI-Programme

Eine eigenständige Win-CGI-Web-Server-Anwendung ist eine Windows-Anwendung, die Client-Anforderungen aus einer vom Server erstellten INI-Datei (mit Konfigurationseinstellungen) empfängt. Die Ergebnisse werden in eine Datei geschrieben, die der Server dann an den Client zurückgibt. Die INI-Datei wird von *TCGIApplication* ausgewertet. Dieses Objekt erzeugt die *TWinCGIRequest*- und *TWinCGIResponse*-Objekte. Jede Anforderungsbotschaft wird von einer gesonderten Instanz der Anwendung bearbeitet.

Web-Server-Anwendungen erstellen

Zur Erstellung einer neuen Web-Server-Anwendung wählen Sie zunächst *Datei / Neu* aus dem Menü im Hauptfenster und anschließend im Dialogfeld *Objektgalerie* die Option Web-Server-Anwendung. Daraufhin wird ein Dialogfeld eingeblendet, in dem Sie die Art der Web-Server-Anwendung festlegen können:

- *ISAPI* und *NSAPI*: Durch Auswahl dieser Anwendungsart wird Ihr Projekt als DLL mit den vom Web-Server benötigten exportierten Methoden eingerichtet. Der Bibliotheks-Header wird der Projektdatei hinzugefügt. Außerdem werden die erforderlichen Einträge zur uses-Liste und exports-Klausel der Projektdatei hinzugefügt.
- *CGI*, ausführbare Datei: Wenn Sie diese Anwendungsart auswählen, wird Ihr Projekt als Konsolenanwendung eingerichtet und die erforderlichen Einträge werden zur uses-Klausel der Projektdatei hinzugefügt.
- *Win-CGI* standalone: Wenn Sie diese Anwendungsart wählen, wird Ihr Projekt als Windows-Anwendung eingerichtet, und die erforderlichen Einträge werden zur uses-Klausel der Projektdatei hinzugefügt.

Die Art der ausgewählten Web-Server-Anwendung richtet sich nach dem Typ des verwendeten Web-Servers (die Anwendung muß mit diesem kommunizieren können). So wird ein neues Projekt erstellt, das für die Verwendung von Internet-Komponenten konfiguriert ist und ein leeres Web-Modul enthält.

Das Web-Modul

Das Web-Modul (*TWebModule*) ist ein Nachkomme von *TDataModule* und kann zur zentralen Kontrolle von grundlegenden Strukturen der Programmlogik (Business Rules) und nichtvisuellen Komponenten in der Web-Anwendung eingesetzt werden.

In dieses Modul nehmen Sie alle String-Generatoren auf, mit denen Ihre Anwendung Antwortbotschaften erzeugt. Es kann sich hierbei um integrierte String-Generatoren, wie zum Beispiel *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer* und *TMIDASPageProducer* handeln, oder um Nachkommen von *TCustomContentProducer*, die Sie selbst geschrieben haben. Wenn Ihre Anwendung Antwortbotschaften erzeugt, in denen Material aus Datenbanken enthalten ist, können Sie Datenzugriffskomponenten oder spezielle Komponenten hinzufügen und einen Web-Server erstellen, der als Client in einer MIDAS-Anwendung fungiert.

Außer zur Aufnahme von nichtvisuellen Komponenten und grundlegenden Strukturen der Programmlogik wird das Web-Modul auch als DispatcherWeb-Dispatcher eingesetzt, der eingehende HTTP-Anforderungsbotschaften an Aktionselemente übergibt, die Antworten auf die Anforderungen generieren.

Möglicherweise verfügen Sie bereits über ein Datenmodul, das viele der nichtvisuellen Komponenten und grundlegenden Strukturen der Programmlogik enthält, die Sie in Ihrer Web-Anwendung einsetzen wollen. Sie können das Web-Modul in diesem Fall durch dieses Datenmodul ersetzen. Dazu löschen Sie einfach das automatisch erzeugte Web-Modul und ersetzen es durch das betreffende Datenmodul. An-

schließend fügen Sie eine *TWebDispatcher* -Komponente zu dem Datenmodul hinzu. Das Datenmodul kann nun wie ein Web-Modul Anforderungsbotschaften an Aktionselemente weiterleiten. Sie können die Art und Weise ändern, in der Aktionselemente zur Beantwortung eingehender HTTP-Anforderungsbotschaften ausgewählt werden. Dazu leiten Sie eine neue Dispatcher-Komponente von *TCustomWebDispatcher* ab und fügen diese zum Datenmodul hinzu.

Ihr Projekt kann nur einen Dispatcher enthalten. Dies kann entweder das Web-Modul sein, das bei der Erstellung des Projekts automatisch generiert wird, oder die *TWebDispatcher*-Komponente, die Sie zu einem Datenmodul hinzufügen, welches das Web-Modul ersetzt. Wenn während der Ausführung ein zweites Datenmodul erzeugt wird, das einen Dispatcher enthält, erzeugt die Web-Server-Anwendung einen Laufzeitfehler.

Hinweis Das von Ihnen zur Entwurfszeit konfigurierte Web-Modul stellt eine Vorlage dar. In ISAPI- und NSAPI-Anwendungen spaltet jede Anforderungsbotschaft einen eigenen Thread ab. Für jeden Thread wird dynamisch eine separate Instanz des Web-Moduls und dessen Inhalt erzeugt

Achtung Das Web-Modul in einer DLL-basierten Web-Server-Anwendung wird in den Cache geladen, damit die Antwortzeit bei der späteren Wiederverwendung verkürzt wird. Der Status des Dispatchers und dessen Aktionsliste werden zwischen den einzelnen Anforderungen nicht erneut initialisiert. Durch das Aktivieren oder Deaktivieren von Aktionselementen bei der Ausführung kann es zu unerwarteten Ergebnissen kommen, wenn dieses Modul für nachfolgende Client-Anforderungen verwendet wird.

Das Web-Anwendungsobjekt

Das Projekt, das für eine Web-Anwendung konfiguriert wird, enthält eine globale Variable namens *Application*. *Application* ist ein Nachkomme von *TWebApplication* (entweder *TISAPIApplication* oder *TCGIApplication*), welcher der erstellten Anwendungsart entspricht. Diese Anwendung wird als Antwort auf HTTP-Anforderungsbotschaften ausgeführt, die vom Web-Server empfangen werden.

Warnung Fügen Sie die Unit *Form*s nicht nach den Units *CGIApp* oder *ISAPIApp* in die **uses**-Liste des Projekts ein. Die Unit *Forms* deklariert ebenfalls eine globale Variable mit der Bezeichnung Application. Erscheint sie nach den Units *CGIApp* oder *ISAPIApp*, wird *Application* mit einem Objekt des falschen Typs initialisiert.

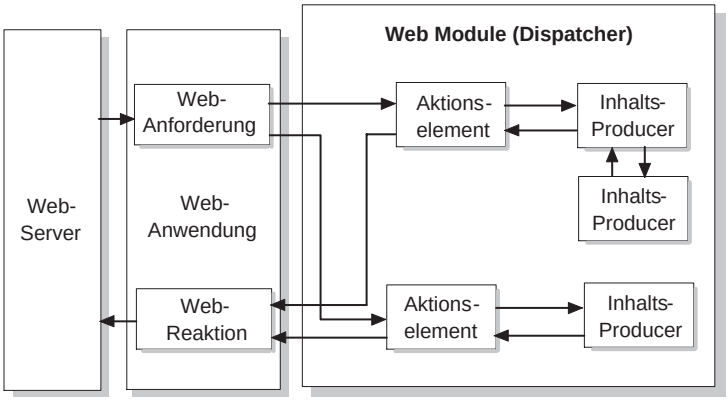
Die Struktur einer Web-Server-Anwendung

Wenn die Web-Anwendung eine HTTP-Anforderungsbotschaft empfängt, erzeugt sie ein *TWebRequest*-Objekt, das die HTTP-Anforderungsbotschaft repräsentiert, und ein *TWebResponse*-Objekt, das die Antwort darstellt, die zurückgegeben werden soll. Diese Objekte werden von der Anwendung an den Web-Dispatcher übergeben (dies ist entweder ein Web-Modul oder eine *TWebDispatcher*-Komponente).

DerWeb-Dispatcher steuert den Ablauf der Web-Server-Anwendung. Er verwaltet eine Gruppe von Aktionselementen (*TWebActionItem*), in denen die Behandlung bestimmter Typen von HTTP-Anforderungsbotschaften festgelegt ist. Der Dispatcher

identifiziert die Aktionselemente oder die Auto-Dispatch-Komponenten, die zur Behandlung der HTTP-Anforderungsbotschaft eingesetzt werden müssen, und übergibt die Anforderungs- und Antwortobjekte an die entsprechende Behandlungsroutine. Diese führt daraufhin die angeforderten Aktionen durch oder formuliert eine Antwortbotschaft. Der gesamte Vorgang wird im Abschnitt »Der Web-Dispatcher« auf Seite 29-10 beschrieben.

Abbildung 29.2 Die Struktur einer Server-Anwendung



Die Aktionselemente sind für den Zugriff auf die Anforderung und das Erstellen der Antwortbotschaft zuständig. Spezielle String-Generatorkomponenten helfen den Aktionselementen dabei, dynamisch den Inhalt von Antwortbotschaften zu generieren , die benutzerdefinierten HTML-Code oder anderen MIME-Inhalt enthalten können. Die String-Generatoren können bei der Erzeugung des Inhalts für die Antwortbotschaft auf andere String-Generatoren bzw. auf Nachkommen von *THTMLTagAttributes* zurückgreifen. Weitere Informationen über String-Generatoren finden Sie im Abschnitt »Den Inhalt von Antwortbotschaften generieren« auf Seite 29-20.

Wenn Sie den Web-Client in einer mehrschichtigen Datenbankanwendung erzeugen , kann die Web-Server-Anwendung weitere Auto-Disptach-Komponenten enthalten, die in XML codierte Datenbankinformationen und in Javascript codierte Klassen zur Datenbankbearbeitung repräsentieren. Der Dispatcher ruft diese Komponenten nur dann auf, wenn die Anforderungsbotschaft nicht mit den »normalen« Aktionselementen verarbeitet werden kann.

Wenn alle Aktionselemente (oder Auto-Dispatch-Komponenten) das Ausfüllen des *TWebResponse*-Objekts abgeschlossen haben, und die Antworterstellung damit beendet ist, gibt der Dispatcher das Ergebnis an die Web-Anwendung weiter. Die Anwendung sendet die Antwort dann über den Web-Server an den Client.

Der Web-Dispatcher

Wenn Sie ein Web-Modul einsetzen, fungiert dieses als Web-Dispatcher. Verwenden Sie dagegen ein vorhandenes Datenmodul, müssen Sie eine (einzelne) Dispatcher-Komponente (*TWebDispatcher)* zu diesem Modul hinzufügen. Der Dispatcher verwaltet eine Gruppe von Aktionselementen, in denen die Behandlung bestimmter Anforderungsbotschaften festgelegt ist. Wenn die Web-Anwendung ein Anforderungsobjekt und ein Antwortobjekt an den Dispatcher übergibt, wählt dieser ein oder mehrere Aktionselemente aus, die auf die Anforderung antworten.

Aktionen zum Dispatcher hinzufügen

Öffnen Sie den Aktionseditor im Objektinspektor, indem Sie auf die Ellipsenschaltfläche der Eigenschaft *Actions* des Dispatchers klicken. Klicken Sie anschließend im Aktionseditor auf die Schaltfläche *Hinzufügen*, um die gewünschten Aktionselemente hinzuzufügen.

Aktionselemente können auf unterschiedliche Arten eingerichtet werden. Sie können zum Beispiel mit Aktionselementen beginnen, die eine Vorverarbeitung der Anforderungen ausführen, und mit einer Standardaktion aufhören, die überprüft, ob die Anwort vollständig ist und sie anschließend sendet oder gegebenenfalls einen Fehlercode zurückliefert. Es ist aber auch möglich, für jede Anforderungsart ein eigenes Aktionselement hinzuzufügen, das die Anforderung vollständig bearbeitet.

Eine ausführliche Beschreibung von Aktionselementen finden Sie im Abschnitt »Aktionselemente« auf Seite 29-11.

Anforderungsbotschaften verteilen

Wenn der Dispatcher die Client-Anforderung empfängt, erzeugt er das Ereignis *BeforeDispatch*. Die Anwendung hat damit Gelegenheit, eine Vorverarbeitung der Anforderungsbotschaft durchzuführen, bevor sie zu einem Aktionselement gelangt.

Anschließend prüft der Dispatcher die Liste seiner Aktionselemente nach einen Element, das dem *PathInfo*-Teil in der Ziel-URL der Anforderungsbotschaft entspricht und anzeigt, daß es den Dienst bereitstellen kann, der als Methode der Anforderungsbotschaft angegeben ist. Dazu vergleicht der Dispatcher die Eigenschaften *PathInfo* und *MethodType* des *TWebRequest*-Objekts mit den Eigenschaften des identischen Namens im Aktionselement

Wenn der Dispatcher ein entsprechendes Aktionselement findet, löst er das Aktionselement aus. Dieses führt dann eine der folgenden Aktionen aus:

- Es legt den Antwortinhalt fest und sendet die Antwort oder zeigt an, daß die Anforderung vollständig verarbeitet wurde.
- Es erstellt einen Teil der Antwort und überläßt die restliche Verarbeitung anderen Aktionselementen.
- Es leitet die Anforderung an andere Aktionselemente weiter.

Wenn der Dispatcher alle Aktionselemente überprüft hat, die Anforderungsbotschaft aber immer noch nicht verarbeitet wurde, testet er alle registrierten Auto-Dispatch-Komponenten, die keine Aktionselemente verwenden. Diese Komponenten für mehrschichtige Datenbankanwendungen werden unter »Web-Anwendungen mit InternetExpress erstellen« auf Seite 14-34.

Wenn der Dispatcher alle Aktionselemente sowie die speziell registrierten Auto-Dispatch-Komponenten überprüft hat, die Anforderungsbotschaft aber immer noch nicht vollständig verarbeitet wurde, ruft er das Standard-Aktionselement auf. Das Standard-Aktionselement muß nicht mit der Ziel-URL oder der Methode der Anforderung identisch sein.

Wenn der Dispatcher das Ende der Aktionsliste erreicht (einschließlich des Standard-Aktionselements, falls vorhanden) und keine Aktion ausgelöst wurde, erfolgt keine Rückgabe an den Server. Der Server beendet dann die Verbindung zum Client.

Wurde die Anforderung dagegen von den Aktionselementen verarbeitet, erzeugt der Dispatcher ein *AfterDispatch* -Ereignis. Die Anwendung hat dann zum letzten Mal die Gelegenheit, die erzeugte Antwort zu überprüfen und noch Änderungen durchzuführen.

Aktionselemente

Jedes Aktionselement (*TWebActionItem)* führt als Reaktion auf eine bestimmte Art von Anforderungsbotschaft eine spezifische Operation aus.

Ein Aktionselement kann eine Anforderung vollständig bearbeiten oder nur einen Teil der Antwort erstellen und die restliche Verarbeitung anderen Aktionselementen überlassen. Sie können die HTTP-Antwortbotschaft für die Anforderung selbst senden oder nur einen Teil der Antwort vorbereiten und die Komplettierung dann von anderen Aktionselementen ausführen lassen. Wenn eine Antwort von einem Aktionselement fertiggestellt, aber nicht gesendet wird, übernimmt die Web-Server-Anwendung das Senden der Antwortbotschaft.

Das Auslösen von Aktionselementen festlegen

Die meisten Eigenschaften eines Aktionselements legen fest, wann es vom Dispatcher zur Bearbeitung einer HTTP-Anforderungsbotschaft ausgewählt wird. Damit Sie die Eigenschaften eines Aktionselements definieren können, starten Sie den Aktionseditor. Wählen Sie dazu im Objektinspektor die Eigenschaft *Actions* des Dispatchers aus, und klicken Sie auf die Ellipsenschaltfläche. Wenn eine Aktion im Aktionseditor ausgewählt ist, können ihre Eigenschaften im Objektinspektor geändert werden.

Die Ziel-URL

Der Dispatcher vergleicht die Eigenschaft *PathInfo* eines Aktionselements mit *PathInfo* der Ziel-URL der Anforderungsbotschaft. Der Wert dieser Eigenschaft sollte mit der Information im Pfadabschnitt der URL aller Anforderungen identisch sein, die

von dem Aktionselement verarbeitet werden können. Angenommen, in der folgenden URL stellt der Abschnitt `/gallery.dll` die Web-Server-Anwendung dar:

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
Der Abschnitt mit der Pfadinformation lautet dann:

/mammals
```

Über die Pfadinformation geben Sie an, wo die Web-Server-Anwendung beim Bedienen von Anforderungen nach Informationen suchen soll, oder Sie legen damit fest, daß die Web-Anwendung in logische Teildienste unterteilt werden soll.

Der Anforderungsmethodentyp

Die Eigenschaft *MethodType* eines Aktionselements legt fest, welche Art von Anforderungsbotschaft das Element verarbeiten kann. Der Dispatcher vergleicht die Eigenschaft *MethodType* eines Aktionselements mit dem Wert von *MethodType* der Anforderungsbotschaft. *MethodType* kann folgende Werte annehmen:

Tabelle 29.2 MethodType-Werte

Wert	Bedeutung
mtGet	Die Anforderung fragt nach der Information, die mit der Ziel-URI verbunden ist und in einer Antwortbotschaft zurückgeliefert werden soll.
mtHead	Die Anforderung fragt nach den Header-Eigenschaften einer Antwort (dies entspricht der Bedienung einer *mtGet*-Anforderung), läßt aber den Inhalt der Antwort weg.
mtPost	Die Anforderung stellt Informationen bereit, die an die Web-Anwendung zurückgesendet werden müssen.
mtPut	Die Anforderung verlangt, daß die mit der Ziel-URI verbundene Ressource durch den Inhalt der Anforderungsbotschaft ersetzt werden soll.
mtAny	Das Aktionselement kann jede Art von Anforderungsmethode bearbeiten, einschließlich *mtGet*, *mtHead*, *mtPut* und *mtPost*.

Aktionselemente aktivieren und deaktivieren

Jedes Aktionselement besitzt die Eigenschaft *Enabled*. Über diese Eigenschaft lassen sich Aktionselemente aktivieren und deaktivieren. Durch Setzen von *Enabled* auf *False* deaktivieren Sie das Aktionselement. Es wird dann vom Dispatcher nicht berücksichtigt, wenn dieser nach einem Aktionselement zur Bearbeitung einer Anforderung sucht.

In einer Ereignisbehandlungsroutine für *BeforeDispatch* können Sie festlegen, welche Aktionselemente eine Anforderung bearbeiten. Dazu wird die Eigenschaft *Enabled* der Aktionselemente entsprechend gesetzt, bevor der Dispatcher beginnt, sie für die Anforderungsbotschaft zu berücksichtigen.

Achtung Eine Änderung der Eigenschaft *Enabled* während der Ausführung kann bei nachfolgenden Anforderungen zu unerwünschten Ergebnissen führen. Wenn die Web-Server-Anwendung eine DLL ist, die Web-Module zwischenspeichert, erfolgt keine Neuinitialisierung des Anfangsstatus für die nächste Anforderung. Stellen Sie mit

Hilfe des Ereignisses *BeforeDispatch* sicher, daß alle Aktionselemente korrekt für ihren entsprechenden Anfangsstatus initialisiert sind.

Ein Standard-Aktionselement festlegen

Nur eines der Aktionselemente kann als Standard-Aktionselement fungieren. Sie machen ein Aktionselement zum Standard-Aktionselement, indem Sie seine Eigenschaft *Default* auf *True* setzen. Dies hat zur Folge, daß die *Default*-Eigenschaft des bisherigen Standard-Aktionselement (falls vorhanden) auf *False* gesetzt wird.

Wenn der Dispatcher auf der Suche nach einem Aktionselement zur Bearbeitung einer Anforderung seine Aktionsliste durchsucht, speichert er den Namen des Standard-Aktionselements. Erreicht der Dispatcher das Ende der Aktionsliste und wurde die Anforderung noch nicht vollständig bearbeitet, wird das Standard-Aktionselement ausgeführt.

Der Dispatcher überprüft weder die Eigenschaften *PathInfo* und *MethodType* des Standard-Aktionselements noch dessen Eigenschaft *Enabled*. Sie können deshalb sicherstellen, daß das Standard-Aktionselement erst am Ende aufgerufen wird, indem Sie seine Eigenschaft *Enabled* auf *False* setzen.

Das Standard-Aktionselement sollte jede angetroffene Anforderung behandeln können. Es sollte aber mindestens einen Fehlercode zurückliefern, der auf eine ungültige URI oder *MethodType*-Eigenschaft hinweist. Wenn das Standard-Aktionselement die Anforderung nicht bearbeitet, wird keine Antwort an den Web-Client gesendet.

Achtung Eine Änderung der Eigenschaft *Default* während der Ausführung kann zu unerwünschten Ergebnissen für die aktuelle Anforderung führen. Wenn Sie die Eigenschaft Default einer bereits ausgelösten Aktion auf *True* setzen, wird diese Aktion nicht neu ausgewertet, und der Dispatcher löst die Aktion nicht aus, wenn er das Ende der Aktionsliste erreicht.

Mit Aktionselementen auf Anforderungsbotschaften antworten

Die Hauptarbeit der Web-Server-Anwendung wird von den Aktionselementen geleistet. Nachdem der Web-Dispatcher ein Aktionselement ausgelöst hat, kann dieses auf zwei Arten auf die Anforderungsbotschaft antworten:

- Wenn in der Eigenschaft *Producer* des Aktionselements eine Generatorkomponente angegeben ist, weist dieser Generator automatisch mit seiner *Content* -Methode der Eigenschaft *Content* der Antwortbotschaft einen Wert zu. Die Registerkarte Internet der Komponentenpalette stellt String-Generatoren bereit, mit denen Sie eine HTML-Seite generieren können, die den Inhalt der Antwortbotschaft bildet.
- Nachdem der Generator (falls vorhanden) einen Inhalt für die Antwortbotschaft erzeugt hat, empfängt das Aktionselement das Ereignis *OnAction*. An die Ereignisbehandlungsroutine für OnAction wird das *TWebRequest* -Objekt übergeben, das die HTTP-Anforderungsbotschaft darstellt, und zusätzlich ein *TWebResponse*-Objekt, das mit den Antwortinformationen gefüllt wird.

Wenn der Inhalt für das Aktionselement von einem String-Generator erzeugt werden kann, reicht es aus, diesen in der Eigenschaft *Producer* des Aktionselements anzuge-

ben. In der Ereignisbehandlungsroutine für *OnAction* kann allerdings auch auf andere String-Generatoren zugegriffen werden. Eine Ereignisbehandlungsroutine für *OnAction* bietet eine größere Flexibilität, da sie den Einsatz mehrerer String-Generatoren, das Setzen von Eigenschaften für Antwortbotschaften usw. ermöglicht.

In der Behandlungsroutine für *OnAction* können Sie alle Objekte bzw. Methoden der Laufzeitbibliothek einsetzen, um auf Anforderungsbotschaften zu antworten. Dies gilt auch bei Verwendung einer String-Generatorkomponente. Sie können z. B. auf Datenbanken zugreifen, Berechnungen durchführen, HTML-Dokumente generieren oder auswählen usw. Weitere Informationen zum Erstellen des Antwortinhalts mit Hilfe von String-Generatoren finden Sie unter »Den Inhalt von Antwortbotschaften generieren« auf Seite 29-20.

Die Antwort senden

Eine Ereignisbehandlungsroutine für *OnAction* kann zum Senden der Antwort an den Web-Client die Methoden des *TWebResponse*-Objekts einsetzen. Wenn kein Aktionselement eine Antwort an den Client schickt, übernimmt die Web-Server-Anwendung das Senden. Voraussetzung dafür ist aber, daß das letzte Aktionselement, dem die Anforderung vorgelegt wurde, anzeigt, daß sie bearbeitet worden ist.

Mehrere Aktionselemente verwenden

Sie können auf eine Anforderung innerhalb eines einzelnen Aktionselements antworten oder aber die Arbeit auf mehrere Aktionselemente verteilen. Wenn ein Aktionselement die Antwortbotschaft nicht vollständig bereitstellt, muß es dies in der Ereignisbehandlungsroutine für *OnAction* signalisieren. Dazu wird der Parameter *Handled* auf *False* gesetzt.

Wenn die Beantwortung von Anforderungsbotschaften auf viele Aktionselemente verteilt wird und jedes Element durch Setzen von *Handled* auf *False* anzeigt, daß ein anderes Element mit der Bearbeitung fortfahren kann, muß das Standard-Aktionselement den Parameter *Handled* auf *True* gesetzt lassen. Andernfalls wird keine Antwort an den Web-Client gesendet.

Wenn die Arbeit auf mehrere Aktionselemente verteilt wird, muß entweder die *OnAction*-Ereignisbehandlungsroutine für das Standard-Aktionselement oder die Behandlungsroutine für *AfterDispatch* des Dispatchers überprüfen, ob die Anforderung vollständig bearbeitet wurde, und bei Bedarf einen entsprechenden Fehlercode generieren.

Auf Client-Anforderungsinformationen zugreifen

Wenn eine Web-Server-Anwendung eine HTTP-Anforderungsbotschaft empfängt, werden die Header der Client-Anforderung in die Eigenschaften eines *TWebRequest*-Objekts geladen. In NSAPI- und ISAPI-Anwendungen wird die Anforderungsbotschaft in einem *TISAPIRequest* -Objekt gekapselt. Konsolenanwendungen benutzen *TCGIRequest*-Objekte, und Win-CGI-Anwendungen benutzen *TWinCGIRequest*-Objekte.

Die Eigenschaften des Anforderungsobjekts sind vom Typ Nur-Lesen. Über diese Eigenschaften können die gesamten Informationen zusammengestellt werden, die in der Client-Anforderung verfügbar sind.

Eigenschaften, die Header-Informationen zur Anforderung enthalten

Die meisten Eigenschaften eines Anforderungsobjekts enthalten Informationen über die Anforderung, die aus dem HTTP-Anforderungs-Header übernommen werden. Allerdings stellen nicht alle Anforderungen Werte für alle diese Eigenschaften bereit. Anforderungen können außerdem Header-Felder enthalten, die nicht in einer Eigenschaft des Anforderungsobjekts abgebildet sind. Dies gilt speziell im Hinblick auf eine Weiterentwicklung des HTTP-Standards. Mit der Methode *GetFieldByName* können Sie den Wert eines Header-Feldes ermitteln, das nicht als Eigenschaft des Anforderungsobjekts abgebildet wird.

Eigenschaften, die das Ziel bezeichnen

Das vollständige Ziel der Anforderungsbotschaft wird über die Eigenschaft *URL* angegeben. In der Regel handelt es sich hierbei um eine URL, die aus folgenden Teilen besteht: *Protokoll* (HTTP), *Host* (Server-System), *ScriptName* (Server-Anwendung), *PathInfo* (Position auf dem Host) und *Query* (Abfrage).

Jeder dieser Bestandteile erscheint in einer eigenen Eigenschaft. Beim Protokoll handelt es sich immer um HTTP, über *Host* und *ScriptName* wird die Web-Server-Anwendung identifiziert. Der Dispatcher verwendet den Abschnitt *PathInfo* bei der Zuordnung von Aktionselementen an Anforderungsbotschaften. Der Abschnitt *Query* wird von einigen Anforderungen zur Angabe der Details zu den angeforderten Informationen verwendet. Dieser Wert dient nach seiner Zerlegung in Felder (Parsen) zum Definieren der Eigenschaft *QueryFields*.

Eigenschaften, die den Web-Client beschreiben

Zur Anforderung gehören auch Eigenschaften, die Informationen über ihre Herkunft enthalten. Dazu gehört alles, von der E-Mail-Adresse des Senders (Eigenschaft *From*) bis zur URI, von der die Botschaft stammt (Eigenschaft *Referer* oder *RemoteHost*). Umfaßt die Anforderung einen Inhalt, der nicht aus derselben URI wie die Anforderung selbst stammt, wird die Quelle des Inhalts in der Eigenschaft *DerivedFrom* angegeben. Sie können auch die IP-Adresse des Clients (Eigenschaft *RemoteAddr)* sowie den Namen und die Version der Anwendung feststellen, von der die Anforderung gesendet wurde (Eigenschaft *UserAgent).*

Eigenschaften, die auf den Zweck der Anforderung hinweisen

Die Eigenschaft *Method* besteht aus einem String, in dem angegeben ist, zu welcher Aktion die Server-Anwendung von der Anforderungsbotschaft aufgefordert wird. Im Standard HTTP 1.1 sind folgende Methoden definiert:

Wert	Bedeutung
OPTIONS	Fordert Informationen über die verfügbaren Kommunikationsoptionen an.
GET	Fordert Informationen an, die in der Eigenschaft *URL* hinterlegt sind.
HEAD	Fordert Header-Informationen aus einer entsprechenden GET-Botschaft an (ohne den Inhalt der Antwort).
POST	Fordert die Server-Anwendung auf, die in der Eigenschaft *Content* enthaltenen Daten zu senden.
PUT	Fordert die Server-Anwendung auf, die in der Eigenschaft *URL* angegebene Ressource durch die Daten zu ersetzen, die in der Eigenschaft *Content* enthalten sind.
DELETE	Fordert die Server-Anwendung auf, die in der Eigenschaft *URL* angegebene Ressource zu löschen oder zu verbergen.
TRACE	Fordert die Server-Anwendung auf, den Erhalt der Anforderung durch Senden eines Loopback zu bestätigen.

Die Eigenschaft *Method* kann auf jede andere Methode verweisen, die der Web-Client vom Server anfordert.

Die Web-Server-Anwendung muß nicht für jeden möglichen Wert von *Method* eine Antwort bereitstellen. Jedoch fordert der HTTP-Standard, daß GET- und HEAD-Anforderungen bedient werden.

Die Eigenschaft *MethodType* zeigt an, ob der Wert von *Method* GET (*mtGet*), HEAD (*mtHead*), POST (*mtPost*) oder PUT (*mtPut*) lautet, oder ob sie einen anderen String (*mtAny*) enthält. Der Dispatcher vergleicht den Wert der Eigenschaft *MethodType* mit der *MethodType*-Eigenschaft jedes Aktionselements.

Eigenschaften, die die erwartete Antwort beschreiben

Die Eigenschaft *Accept* zeigt die Medientypen an, die der Web-Client als Inhalt der Antwortbotschaft akzeptiert. Die Eigenschaft *IfModifiedSince* gibt an, ob der Client nur Informationen anfordert, die sich kürzlich geändert haben. In der Eigenschaft *Cookie* sind Statusinformationen (die in der Regel zuvor von der Anwendung hinzugefügt wurden) enthalten, welche die Antwort ändern können.

Eigenschaften, die den Inhalt beschreiben

Zu den meisten Anforderungen gehört keinerlei Inhalt, da sie nur Informationen anfordern. Bestimmte Anforderungen (z. B. mit der Methode POST) stellen einen Inhalt bereit, von dem erwartet wird, daß ihn die Web-Server-Anwendung benutzt. Der Medientyp des Inhalts wird in der Eigenschaft *ContentType* angegeben, seine Länge in der Eigenschaft *ContentLength*. Falls der Inhalt der Botschaft verschlüsselt war (beispielsweise aufgrund einer Datenkompression), ist diese Information in der Eigen-

schaft *ContentEncoding* enthalten. Der Name und die Versionsnummer der Anwendung, die den Inhalt erzeugt hat, wird in der Eigenschaft *ContentVersion* angegeben. Die Eigenschaft *Title* kann ebenfalls Informationen über den Inhalt enthalten.

Der Inhalt von HTTP-Anforderungsbotschaften

Neben den Header-Feldern enthalten manche Anforderungsbotschaften einen Inhaltsabschnitt, den die Web-Server-Anwendung verarbeiten soll. In einer POST-Anforderung können beispielsweise Informationen enthalten sein, die zu einer von der Web-Server-Anwendung verwalteten Datenbank hinzugefügt werden sollen.

Der unverarbeitete Wert des Inhalts ist in der Eigenschaft *Content* enthalten. Wenn der Inhalt in Felder zerlegt (geparst) werden kann, die durch das Zeichen & voneinander getrennt sind, ist die zerlegte Version über die Eigenschaft *ContentFields* verfügbar.

HTTP-Anwortbotschaften erzeugen

Wenn die Web-Server-Anwendung für eine eingehende HTTP-Anforderungsbotschaft ein *TWebRequest* -Objekt erstellt, erzeugt sie gleichzeitig ein zugehöriges *TWebResponse*-Objekt für die Antwortbotschaft, die an den Client zurückgesendet wird. In NSAPI- und ISAPI-Anwendungen wird die Antwortbotschaft in einem *TISAPIResponse*-Objekt gekapselt. CGI-Konsolenanwendungen benutzen *TCGIResponse*-Objekte, während Windows-CGI-Anwendungen *TWinCGIResponse*-Objekte verwenden.

Die Aktionselemente, welche die Antwort für die Anforderung eines Web-Client generieren, füllen die Eigenschaften des Antwortobjekts aus. In bestimmten Fällen kann die Antwort aus der Rückgabe eines Fehlercodes oder der Weiterleitung der Anforderung an eine andere URI bestehen. Es können aber auch komplizierte Berechnungen erforderlich sein, für die das Aktionselement Informationen aus anderen Quellen abrufen und in einem Formular zusammenführen muß. Bei den meisten Anforderungsbotschaften muß eine Antwort erstellt werden, auch wenn sie nur aus der Bestätigung besteht, daß die angeforderte Aktion ausgeführt wurde.

Den Antwort-Header füllen

Die meisten Eigenschaften des Objekts *TWebResponse* stellen die Header-Informationen der HTTP-Antwortbotschaft dar, die an den Web-Client zurückgesendet wird. Diese Eigenschaften werden in der *OnAction*-Ereignisbehandlungsroutine des Aktionselements gesetzt.

Nicht alle Antwortbotschaften müssen Werte für alle Header-Eigenschaften bereitstellen. Eigenschaften, die gesetzt werden müssen, sind von der Art der Anforderung und dem Status der Antwort abhängig.

Den Antwortstatus anzeigen

Jede Antwortbotschaft muß einen Statuscode enthalten, der den Status der Antwort anzeigt. Der Statuscode wird über die Eigenschaft *StatusCode* spezifiziert. Der HTTP-Standard definiert eine Anzahl von Standard-Statuscodes, die eine feste Bedeutung haben. Daneben können Sie aber auch eigene Statuscodes definieren. Verwenden Sie dazu Werte, die noch nicht belegt sind.

Ein Statuscode besteht aus einer Nummer mit drei Ziffern, wobei die signifikanteste Ziffer die Klasse der Antwort bezeichnet:

- 1xx: Information (die Anforderung wurde empfangen, aber nicht vollständig verarbeitet).
- 2xx: Erfolg (die Anforderung wurde empfangen, verstanden und akzeptiert).
- 3xx: Umleitung (der Client muß weitere Aktionen ausführen, damit die Anforderung fertiggestellt werden kann).
- 4xx: Client-Fehler (die Anforderung wurde nicht verstanden und kann deshalb nicht bearbeitet werden).
- 5xx: Server-Fehler (die Anforderung war gültig, konnte aber vom Server nicht verarbeitet werden).

Zu jedem Statuscode gehört ein String, der seine Bedeutung erläutert. Dieser String ist in der Eigenschaft *ReasonString* enthalten. Bei den vordefinierten Statuscodes muß diese Eigenschaft nicht festgelegt werden. Wenn Sie eigene Statuscodes definieren, sollten Sie die Eigenschaft ReasonString aber definieren.

Auf eine erforderliche Client-Aktion hinweisen

Wenn der Statuscode im Bereich von 300 bis 399 liegt, muß der Client weitere Aktionen ausführen, bevor die Web-Server-Anwendung ihre Anforderung fertigstellen kann. Falls eine Umleitung des Client an eine andere URI erforderlich ist, oder wenn angezeigt werden soll, daß eine neue URI zur Verarbeitung der Anforderung erzeugt wurde, setzen Sie die Eigenschaft *Location* entsprechend. Wenn der Client zur Weiterarbeit ein Kennwort angeben muß, setzen Sie die Eigenschaft *WWWAuthenticate*.

Die Server-Anwendung beschreiben

Bestimmte Antwort-Header-Eigenschaften beschreiben die Fähigkeiten der Web-Server-Anwendung. Die Eigenschaft *Allow* bezeichnet die Methoden, auf die die Anwendung antworten kann. Die Eigenschaft *Server* enthält den Namen und die Versionsnummer der Anwendung, mit der die Antwort generiert wird. Die Eigenschaft *Cookies* kann Statusinformationen darüber enthalten, wie der Client die Server-Anwendung nutzt. Diese Informationen sind in den nachfolgenden Anforderungsbotschaften enthalten.

Den Inhalt beschreiben

Verschiedene Eigenschaften beschreiben den Inhalt der Antwort. *ContentType* enthält den Medientyp der Antwort. *ContentVersion* bezeichnet die Versionsnummer dieses Medientyps. Die Länge der Antwort ist in der Eigenschaft *ContentLength* enthalten. Wenn der Inhalt verschlüsselt ist (beispielsweise aufgrund einer Datenkompression), weisen Sie in der Eigenschaft *ContentEncoding* darauf hin. Stammt der Inhalt aus einer anderen URI, sollte darauf in der Eigenschaft *DerivedFrom* hingewiesen werden. Wenn der Wert des Inhalts zeitsensitiv ist, zeigen die Eigenschaften *LastModified* und *Expires* an, ob der Wert noch gültig ist. Mit Hilfe der Eigenschaft *Title* kann eine Beschreibung des Inhalts angegeben werden.

Den Antwortinhalt festlegen

Bei manchen Anforderungen ist die Antwortbotschaft vollständig in den Header-Eigenschaften der Antwort enthalten. In den meisten Fällen weist aber ein Aktionselement der Antwortbotschaft einen Inhalt zu. Dabei kann es sich um statische Informationen handeln, die in einer Datei gespeichert sind, oder um Informationen, die vom Aktionselement bzw. dessen String-Generator dynamisch erzeugt wurden.

Sie können den Inhalt der Antwortbotschaft entweder mit der Eigenschaft *Content* oder *ContentStream* festlegen.

Bei der Eigenschaft *Content* handelt es sich um einen String. Da Delphi-Strings nicht auf Textwerte beschränkt sind, kann die Eigenschaft *Content* aus einem String mit HTML-Befehlen, Grafiken (z. B. einem Bit-Stream) oder jedem anderen MIME-Inhaltstyp bestehen.

Verwenden Sie die Eigenschaft *ContentStream*, wenn der Inhalt der Antwortbotschaft aus einem Stream gelesen werden kann. Soll die Antwortbotschaft zum Beispiel den Inhalt einer Datei senden, verwenden Sie für die Eigenschaft *ContentStream* ein *TFileStream*-Objekt. Ebenso wie die Eigenschaft *Content* kann auch *ContentStream* aus einem String mit HTML-Befehlen oder einem anderen MIME-Inhalt bestehen. Wenn Sie die Eigenschaft *ContentStream* einsetzen, dürfen Sie den Stream nicht selbst freigeben, da die Freigabe automatisch durch das Web-Antwortobjekt erfolgt.

Hinweis Wenn der Wert der Eigenschaft *ContentStream* nicht **nil** ist, wird die Eigenschaft *Content* ignoriert.

Die Antwort senden

Wenn zur Beantwortung einer Anforderungsbotschaft keine weiteren Aktionen erforderlich sind, können Sie die Antwort senden. Dies kann direkt von einer *OnAction*-Ereignisbehandlungsroutine aus geschehen. Das Antwortobjekt stellt zwei Methoden für das Senden einer Antwort bereit: *SendResponse* und *SendRedirect*. Durch einen Aufruf von *SendResponse* senden Sie die Antwort mit dem angegebenen Inhalt und den gesamten Header-Informationen des *TWebResponse*-Objekts. Wenn dagegen nur die Umleitung des Web-Client an eine andere URI erforderlich ist, verwenden Sie die Methode *SendRedirect*.

Falls keine der Ereignisbehandlungsroutinen die Antwort sendet, übernimmt das Web-Anwendungsobjekt das Senden, sobald der Dispatcher seine Arbeit beendet hat. Zeigt allerdings keines der Aktionselemente an, daß die Anforderung bearbeitet wurde, beendet die Anwendung die Verbindung zum Web-Client, ohne eine Antwort zu senden.

Den Inhalt von Antwortbotschaften generieren

Verschiedene Delphi-Objekte unterstützen die Aktionselemente bei der Erstellung des Inhalts von HTTP-Antwortbotschaften. Sie können mit diesen Objekten Strings mit HTML-Befehlen erstellen, die in einer Datei gespeichert oder direkt an den Web-Client gesendet werden. Sie haben auch die Möglichkeit, eigene String-Generatoren zu entwickeln, die von *TCustomContentProducer* bzw. dessen Nachfahren abgeleitet sind.

TCustomContentProducer stellt eine grundlegende Schnittstelle zur Erstellung jedes beliebigen MIME-Typs bereit. Dazu gehört auch der Inhalt einer HTTP-Antwortbotschaft. Zu seinen Nachkommen gehören auch Seiten- und Tabellengeneratoren:

- Seitengeneratoren durchsuchen HTML-Dokumente nach bestimmten Tags, die sie durch bereitgestellten HTML-Code ersetzen. Diese werden im folgenden Abschnitt beschrieben.
- Tabellengeneratoren erzeugen anhand von Informationen in einer Datenmenge HTML-Befehle. Weitere Informationen hierzu finden Sie unter »Datenbankinformationen in Antworten integrieren« auf Seite 29-24.

Seitengeneratoren einsetzen

Seitengeneratoren (die Komponente *TPageProducer* und ihre Nachkommen) konvertieren eine *HTML*-Vorlage , indem sie bestimmte HTML-transparente Tags durch angepaßten HTML-Code ersetzen. Sie können im Speicher eine Gruppe von Standard-Antwortvorlagen bereitstellen, die von den Seitengeneratoren ausgefüllt werden, wenn die Antwort auf eine HTTP-Anforderungsbotschaft generiert werden muß. Seitengeneratoren können verkettet werden, um ein HTML-Dokument durch sukzessives Anpassen der HTML-transparenten Tags iterativ aufzubauen.

HTML-Vorlagen

Eine HTML-Vorlage besteht aus einer Folge von HTML-Befehlen und HTML-transparenten Tags. Ein HTML-transparentes Tag hat folgende Form:

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

Die spitzen Klammern (< und >) legen den Gesamtbereich des Tag fest. Das Zeichen # folgt direkt (ohne dazwischenliegende Leerzeichen) nach der öffnenden spitzen Klammer (<). Das Zeichen # kennzeichnet den String für den Seitengenerator als HTML-transparentes Tag. Der Tag-Name folgt (ohne dazwischenliegende Leerzeichen) unmittelbar nach dem Zeichen #. Es kann sich dabei um jeden gültigen Be-

zeichner handeln. Der Tag-Name verweist auf den Konvertierungstyp, den das Tag repräsentiert.

Nach dem Tag-Namen kann das HTML-transparente Tag optionale Parameter enthalten, über die Einzelheiten der durchzuführenden Konvertierung angegeben werden. Jeder Parameter weist das Format *ParamName=Wert* auf, wobei zwischen den einzelnen Bestandteilen eines Parameters keine Leerzeichen vorhanden sein dürfen. Die verschiedenen Parameter werden dagegen durch Leerzeichen voneinander getrennt.

Durch die spitzen Klammern (< und >) ist das Tag auch für HTML-Browser transparent, die das Konstrukt *#TagName* nicht erkennen.

Vordefinierte HTML-transparente Tag-Namen verwenden

Einerseits können Sie Ihre eigenen HTML-transparenten Tags zur Darstellung jeglicher vom Seitengenerator verarbeiteten Informationen erstellen, andererseits gibt es einige vordefinierte Tag-Namen, die mit Werten vom Datentyp *TTag* verknüpft sind. Diese vordefinierten Tag-Namen entsprechen HTML-Befehlen, die von einer Antwortbotschaft zur anderen verschieden sein können. Sie sind in der folgenden Tabelle aufgelistet:

Tag-Name	TTag-Wert	Beschreibung
Link	tgLink	Das Tag wird in einen Hypertext-Link konvertiert. Das Ergebnis ist eine HTML-Sequenz, die mit einem <A>-Tag beginnt und einem </A>-Tag endet.
Image	tgImage	Das Tag wird in ein Grafikbild konvertiert. Das Ergebnis ist ein HTML-Tag <IMG>.
Table	tgTable	Das Tag wird in eine HTML-Tabelle konvertiert. Das Ergebnis ist eine HTML-Sequenz, die mit <TABLE> beginnt und mit </TABLE> endet.
ImageMap	tgImageMap	Das Tag wird in ein Bild mit zugehörigen Hot-Zones konvertiert. Das Ergebnis ist eine HTML-Sequenz, die mit einem Tag <MAP> beginnt und mit </MAP> endet.
Object	tgObject	Das Tag wird in ein eingebettetes ActiveX-Objekt konvertiert. Das Ergebnis ist eine HTML-Sequenz, die mit einem <OBJECT>-Tag beginnt und mit </OBJECT> endet.
Embed	tgEmbed	Das Tag wird in eine Netscape-kompatible Add-in-DLL konvertiert. Das Ergebnis ist eine HTML-Sequenz, die mit einem Tag <EMBED> beginnt und mit </EMBED> endet.

Alle anderen Tag-Namen sind mit *tgCustom* verbunden. Der Seitengenerator stellt keine integrierte Verarbeitung für die vordefinierten Tag-Namen bereit. Diese Tag-Namen unterstützen lediglich Anwendungen bei der Integration des Konvertierungsvorgangs in gängige Operationen.

Hinweis Bei den vordefinierten Tag-Namen wird die Groß-/Kleinschreibung nicht berücksichtigt.

Die HTML-Vorlage definieren

Seitengeneratoren bieten viele Möglichkeiten zur Festlegung der HTML-Vorlage. So können Sie beispielsweise die Eigenschaft *HTMLFile* mit dem Namen einer Datei belegen, in der die HTML-Vorlage enthalten ist. Oder Sie setzen die Eigenschaft *HTMLDoc* auf ein *TStrings* -Objekt, das die HTML-Vorlage enthält. Wenn Sie die Vorlage mit der Eigenschaft HTMLFile oder HTMLDoc angeben, können Sie die konvertierten HTML-Befehle generieren, indem Sie die Methode *Content* aufrufen.

Darüber hinaus besteht die Möglichkeit, durch einen Aufruf der Methode *ContentFromString* eine HTML-Vorlage, die als einzelner String in einem Parameter übergeben wird, direkt zu konvertieren. Sie können aber auch die Methode *ContentFromStream* aufrufen, um die HTML-Vorlage aus einem Stream einzulesen. Beispielsweise könnten Sie alle HTML-Vorlagen in einem Memofeld einer Datenbank speichern und mit der Methode ContentFromStream die konvertieren HTML-Befehle abrufen, wobei die jeweilige Vorlage direkt aus einem *TBlobStream* -Objekt gelesen wird.

HTML-transparente Tags konvertieren

Der Seitengenerator konvertiert die HTML-Vorlage, sobald Sie eine seiner *Content*-Methoden aufrufen. Wenn die Methode *Content* auf ein HTML-transparentes Tag trifft, löst sie die Behandlungsroutine für das Ereignis *OnHTMLTag* aus. Die von Ihnen geschriebene Ereignisbehandlungsroutine muß den Typ des angetroffenen Tags feststellen und es durch den angepaßten Inhalt ersetzen.

Wenn Sie keine *OnHTMLTag*-Ereignisbehandlungsroutine für den Seitengenerator erstellen, werden HTML-transparente Tags durch einen leeren String dargestellt.

Seitengeneratoren und Aktionselemente

Bei einer typischen Verwendung eines Seitengenerators wird über die Eigenschaft *HTMLFile* eine Datei angegeben, die eine HTML-Vorlage enthält. Die Ereignisbehandlungsroutine für *OnAction* ruft die Methode *Content* auf, um die Vorlage in die endgültige HTML-Sequenz zu konvertieren:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
   PageProducer1.HTMLFile := 'Greeting.html';
   Response.Content := PageProducer1.Content;
end;
```

Die Datei GREETING.HTML enthält die folgende HTML-Vorlage:

```
<HTML>
<HEAD><TITLE>Unsere neue Website</TITLE></HEAD>
<BODY>
Hallo <#UserName>! Willkommen auf unserer Website.
</BODY>
</HTML>
```

Die Ereignisbehandlungsroutine für *OnHTMLTag* ersetzt während der Ausführung das selbstdefinierte Tag (`<#UserName>`) im HTML-Code:

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
   if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

Wenn der Inhalt der Anforderungsbotschaft aus dem String *Mr. Ed* besteht, lautet der Wert von *Response.Content* wie folgt:

```
<HTML>
<HEAD><TITLE>Unsere neue Website</TITLE></HEAD>
<BODY>
Hallo Mr. Ed! Willkommen auf unserer Website.
</BODY>
</HTML>
```

Hinweis In diesem Beispiel wird in einer Ereignisbehandlungsroutine für *OnAction* der Seitengenerator aufgerufen und der Inhalt der Antwortbotschaft zugewiesen. Wenn Sie während des Entwurfs die Eigenschaft *HTMLFile* des Seitengenerators festlegen, müssen Sie keine *OnAction*-Ereignisbehandlungsroutine schreiben, sondern lediglich *PageProducer1* in der Eigenschaft *Producer* des Aktionselements angeben. Damit erzielen Sie denselben Effekt wie mit der obigen Ereignisbehandlungsroutine.

Seitengeneratoren verketten

Der Ersatztext aus einer Behandlungsroutine für das Ereignis *OnHTMLTag* muß nicht mit der endgültigen HTML-Sequenz identisch sein, die in der HTTP-Antwortbotschaft verwendet wird. Möglicherweise möchten Sie verschiedene Seitengeneratoren einsetzen und die Ausgabe des einen Generators als Eingabe für den nächsten verwenden.

Am einfachsten können Sie Seitengeneratoren miteinander verketten, indem Sie jeden Seitengenerator mit einem separaten Aktionselement verknüpfen, wobei alle Aktionselemente identischen *PathInfo*- und *MethodType*-Eigenschaften enthalten müssen. Das erste Aktionselement stellt den Inhalt für die Web-Antwortbotschaft mit Hilfe seines String-Generators bereit. Über die *OnAction*-Ereignisbehandlungsroutine muß sichergestellt werden, daß die Botschaft nicht als vollständig bearbeitet gilt. Das nächste Aktionselement ändert dann den Inhalt der Antwortbotschaft mit der Methode *ContentFromString* seines Generators. Ist die Web-Antwortbotschaft danach noch nicht fertiggestellt, übernimmt das dritte Aktionselement die Verarbeitung. Dieser Vorgang wird fortgesetzt, bis die Antworterstellung abgeschlossen ist. Alle Aktionselemente außer dem ersten verwenden eine *OnAction*-Ereignisbehandlungsroutine der folgenden Form:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
   Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

Angenommen, eine Anwendung liefert Kalenderseiten als Antwort auf Anforderungsbotschaften zurück, in denen der Monat und das Jahr der gewünschten Seite angegeben sind. Auf jeder Kalenderseite befindet sich ein Bild. Darauf folgt der Na-

me und das Jahr des Monats zwischen kleinen Bildern mit dem vorherigen Monat und den nächsten Monaten. Daran schließt sich der Kalender an. Das resultierende Bild sieht in etwa folgendermaßen aus:

Das allgemeine Format des Kalenders wird in einer Vorlagendatei gespeichert. Sie hat folgenden Inhalt:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

Die Behandlungsroutine für das Ereignis *OnHTMLTag* des ersten Seitengenerators ermittelt den Monat und das Jahr aus der Anforderungsbotschaft. Unter Verwendung dieser Informationen und der Vorlagendatei führt sie folgende Aktionen aus:

- <#MonthlyImage> wird durch <#Image Month=January Year=1997> ersetzt.
- <#TitleLine> wird durch <#Calendar Month=December Year=1996 Size=Small> January 1997 <#Calendar Month=February Year=1997 Size=Small> ersetzt.
- <#MainBody> wird durch <#Calendar Month=January Year=1997 Size=Large> ersetzt.

Die Ereignisbehandlungsroutine für *OnHTMLTag* des nächsten Seitengenerators benutzt den vom ersten Seitengenerator erzeugten Inhalt und ersetzt das Tag <#Image Month=January Year=1997> durch das entsprechende HTML-Tag <IMG>. Ein weiterer Seitengenerator löst die #Calendar-Tags mit entsprechenden HTML-Tabellen auf.

Datenbankinformationen in Antworten integrieren

In der Antwort auf eine HTTP-Anforderungsbotschaft können Informationen enthalten sein, die aus einer Datenbank stammen. Die Seite *Internet* der Komponentenpalette stellt spezielle String-Generatoren bereit, die Datensätze aus einer Datenbank in einer HTML-Tabelle darstellen können.

Die Registerkarte *Web-MIDAS* der Komponentenpalette enthält spezielle Komponenten zur Erstellung von Web-Servern, die Teil einer mehrschichtigen Datenbankanwendung sind. Informationen hierzu finden Sie unter »Web-Anwendungen mit InternetExpress erstellen« auf Seite 14-34.

Eine Sitzung zum Web-Modul hinzufügen

Sowohl CGI-Konsolenanwendungen als auch Win-CGI-Anwendungen werden als Antwort auf eine HTTP-Anforderungsbotschaft gestartet. Während Sie in diesen Anwendungsarten mit Datenbanken arbeiten, können Sie über die Standardsitzung Ihre Datenbankverbindungen verwalten, da jede Anforderungsbotschaft über eine eigene Instanz der Anwendung verfügt. Jede Instanz der Anwendung besitzt eine gesonderte Standardsitzung.

Wenn Sie dagegen eine ISAPI- oder eine NSAPI-Anwendung schreiben, wird jede Anforderungsbotschaft in einem separaten Thread einer einzelnen Anwendungsinstanz verarbeitet. Um einen Konflikt zwischen Datenbankverbindungen aus unterschiedlichen Threads zu vermeiden, müssen Sie jedem Thread eine eigene Sitzung zuweisen.

Jede Anforderungsbotschaft in einer ISAPI- oder NSAPI-Anwendung spaltet einen neuen Thread ab. Das Web-Modul für diesen Thread wird dynamisch zur Laufzeit erzeugt. Fügen Sie ein *TSession*-Objekt zum Web-Modul hinzu, das die Datenbankverbindungen für den Thread verwaltet, der das Web-Modul enthält.

Zur Laufzeit wird für jeden Thread eine separate Instanz des Web-Moduls erzeugt. Jedes dieser Module enthält das Sitzungsobjekt. Jeder Sitzung muß ein eigener Name zugewiesen sein, damit zwischen den Datenbankverbindungen der Threads, die separate Anforderungsbotschaften verarbeiten, keine Konflikte entstehen. Wenn für die Sitzungsobjekte in den einzelnen Modulen dynamisch eigene Namen generiert werden sollen, setzen Sie für jedes Sitzungsobjekt die Eigenschaft *AutoSessionName*. Die Sitzungsobjekte erzeugen dann dynamisch eindeutige Namen für sich selbst und setzen die Eigenschaft *SessionName* aller Datenmengen in dem Modul so, daß auf diese eindeutigen Namen verwiesen wird. Dadurch können sämtliche Interaktionen mit der Datenbank für jeden einzelnen Anforderungs-Thread durchgeführt werden, ohne die jeweils anderen Anforderungsbotschaften zu beeinträchtigen. Ausführliche Informationen über Sitzungen finden Sie in Kapitel 16, »Datenbanksitzungen«.

Datenbankinformationen in HTML darstellen

Spezielle String-Generatorkomponenten auf der Registerkarte *Internet* der Komponentenpalette liefern HTML-Befehle, die auf den Datensätzen einer Datenmenge basieren. Man unterscheidet zwei Arten von datensensitiven String-Generatoren:

- Der Datenmengen-Seitengenerator formatiert die Felder einer Datenmenge als Text in einem HTML-Dokument.
- Tabellengeneratoren formatieren die Datensätze einer Datenmenge als HTML-Tabelle.

Datenmengen-Seitengeneratoren verwenden

Datenmengen-Seitengeneratoren funktionieren genauso wie die übrigen Komponenten zur Seitengenerierung: Sie konvertieren eine Vorlage mit HTML-transparenten Tags in eine endgültige HTML-Darstellung. Mit ihrer Hilfe lassen sich jedoch Tags,

deren Tag-Name dem Namen eines Felds in einer Datenmenge entspricht, in den aktuellen Wert des betreffenden Felds konvertieren. Weitere Informationen zur Verwendung von Seitengeneratoren finden Sie unter »Seitengeneratoren einsetzen« auf Seite 29-20.

Zur Verwendung eines Datenmengen-Seitengenerators fügen Sie dem Web-Modul eine *TDataSetPageProducer* -Komponente hinzu und setzen ihre Eigenschaft *DataSet* auf die Datenmenge, deren Feldwerte im HTML-Inhalt angezeigt werden sollen. Erstellen Sie eine HTML-Vorlage, welche die Ausgabe des Datenmengen-Seitengenerators beschreibt. Fügen Sie in der HTML-Vorlage für jeden darzustellenden Feldwert ein Tag im Format

```
<#Feldname>
```

hinzu, wobei *Feldname* den Namen des Felds in der Datenmenge angibt, dessen Wert angezeigt werden soll.

Wenn die Anwendung eine der Methoden *Content, ContentFromString* oder *ContentFromStream* aufruft, setzt der Datenmengen-Seitengenerator die aktuellen Feldwerte für die Tags ein, die diese Felder repräsentieren.

Tabellengeneratoren verwenden

Die Registerkarte *Internet* der Komponentenpalette enthält zwei Komponenten, mit deren Hilfe eine HTML-Tabelle zur Darstellung der Datensätze einer Datenmenge erstellt wird:

- Datenmengen-Tabellengeneratoren> dienen zum Formatieren der Felder einer Datenmenge als Text in einem HTML-Dokument.
- Abfrage-Tabellengeneratoren führen eine Abfrage durch, nachdem die von der Anforderungsbotschaft zu liefernden Parameter festgelegt wurden, und formatieren die resultierende Datenmenge als HTML-Tabelle.

Mit diesen beiden Tabellengeneratoren können Sie die Darstellung der entstehenden HTML-Tabelle durch Festlegung der Eigenschaften für Tabellenfarbe, Tabellenränder, Breite der Trennlinien usw. anpassen. Zur Entwurfszeit können Sie die Eigenschaften eines Tabellengenerators im Dialogfeld *Antwort-Editor* festlegen. Um dieses Dialogfeld zu öffnen, doppelklicken Sie auf die Tabellengeneratorkomponente.

Die Tabellenattribute festlegen

Tabellengeneratoren verwenden das Objekt *THTMLTableAttributes* zur Beschreibung des Erscheinungsbildes einer HTML-Tabelle, in der Datensätze aus der Datenmenge angezeigt werden. Das Objekt *THTMLTableAttributes* enthält Eigenschaften für die Breite und den Abstand der Tabelle innerhalb des HTML-Dokuments sowie für die Hintergrundfarbe, die Rahmenbreite, die Ausrichtung des Zelleninhalts und den Zellenabstand. Diese Eigenschaften werden in Optionen des HTML-Tags <TABLE> umgesetzt, die vom Tabellengenerator erzeugt werden.

Zur Entwurfszeit legen Sie diese Eigenschaften mit Hilfe des Objektinspektors fest. Wählen Sie das Tabellengeneratorobjekt im Objektinspektor aus, und erweitern Sie

die Eigenschaft *TableAttributes*, um auf die Anzeigeeigenschaften des *THTMLTable-Attributes*-Objekts zuzugreifen.

Diese Eigenschaften können Sie auch zur Laufzeit über Quelltextanweisungen angeben.

Die Zeilenattribute festlegen

Ähnlich wie für die Tabellenattribute können Sie auch für die Tabellenzeilen, in denen Daten angezeigt werden, die Ausrichtung und die Hintergrundfarbe der Zellen festlegen. Die Eigenschaft *RowAttributes* ist ein THTMLTableRowAttributes-Objekt.

Zur Entwurfszeit legen Sie diese Eigenschaften mit Hilfe des Objektinspektors fest, indem Sie die Darstellung der Eigenschaft *RowAttributes* erweitern. Sie können diese Eigenschaften aber auch zur Laufzeit über den Quelltext ändern.

Über den Wert der Eigenschaft *MaxRows* können Sie die Anzahl der Zeilen festlegen, die in der HTML-Tabelle angezeigt werden.

Die Spalten festlegen

Wenn Sie die Datenmenge für die Tabelle zur Entwurfszeit bereits kennen, können Sie mit Hilfe des Spalteneditors die Feldbindung und die Anzeigeattribute der Spalten festlegen. Wählen Sie den Tabellengenerator, und klicken Sie mit der rechten Maustaste. Wählen Sie im lokalen Menü den Spalteneditor. In diesem Editor können Sie Spalten hinzufügen, löschen oder anders anordnen. Nach der Auswahl der gewünschten Spalten im Spalteneditor wechseln Sie zum Objektinspektor, um die Feldbindungen und die Anzeigeeigenschaften für die einzelnen Spalten festzulegen.

Wenn der Name der Datenmenge aus der HTTP-Anforderungsbotschaft ermittelt wird, ist es nicht möglich, die Felder zur Entwurfszeit im Spalteneditor zu binden. Die Spalten können dann aber zur Laufzeit vom Programm angepaßt werden. Dazu richten Sie die entsprechenden *THTMLTableColumn*-Objekte ein und fügen sie mit den Methoden der Eigenschaft *Columns* der Tabelle hinzu. Wenn Sie die Eigenschaft *Columns* nicht definieren, erzeugt der Tabellengenerator einen Standardspaltensatz, der den Feldern der Datenmenge entspricht, ohne spezielle Anzeigeeigenschaften festzulegen.

Tabellen in HTML-Dokumente einbetten

Die HTML-Tabelle, die eine Datenmenge repräsentiert, kann mit Hilfe der Eigenschaften *Header* und *Footer* des Tabellengenerators in ein größeres Dokument eingebettet werden. Mit der Eigenschaft *Header* legen Sie sämtliche Informationen vor der Tabelle fest, mit der Eigenschaft *Footer* alles, was auf die Tabelle folgt.

Sie können auch einen anderen String-Generator (zum Beispiel einen Seitengenerator) verwenden, um die Werte für die Eigenschaften *Header* und *Footer* zu generieren.

Wenn die Tabelle in ein größeres Dokument integriert wird, ist es oft sinnvoll, einen Titel hinzuzufügen. Verwenden Sie dazu die Eigenschaften *Caption* und *Caption-Alignment*.

Einen Datenmengen-Tabellengenerator einrichten

TDataSetTableProducer ist ein Tabellengenerator, der für eine Datenmenge eine HTML-Tabelle erzeugt. Die Datenmenge, deren Datensätze in der Tabelle angezeigt werden sollen, legen Sie über die Eigenschaft *DataSet* von *TDataSetTableProducer* fest. Verwenden Sie dazu nicht die Eigenschaft DataSource, die für die meisten datensensitiven Objekte in einer konventionellen Datenbankanwendung eingesetzt wird. *TDataSetTableProducer* generiert seine Datenquelle intern.

Wenn Ihre Web-Anwendung immer die gleichen Datensätze aus derselben Datenmenge anzeigt, können Sie den Wert von *DataSet* zur Entwurfszeit festlegen. Basiert die Datenmenge dagegen auf Informationen in der HTTP-Anforderungsbotschaft, muß die Eigenschaft *DataSet* zur Laufzeit gesetzt werden.

Einen Abfrage-Tabellengenerator einrichten

Sie können zur Anzeige eines Abfrageergebnisses eine HTML-Tabelle erzeugen und dabei die Parameter für die Abfrage aus der HTTP-Anforderungsbotschaft übernehmen. Legen Sie das *TQuery*-Objekt, das diese Parameter benutzt, als *Query*-Eigenschaft einer *TQueryTableProducer* -Komponente fest.

Bei einer GET-Anforderung stammen die Parameter der Abfrage aus den Abfragefeldern der URL, die als Ziel der HTTP-Anforderungsbotschaft angegeben wurde. Handelt es sich dagegen um eine POST-Anforderung, werden die Parameter für die Abfrage aus dem Inhalt der Anforderungsbotschaft übernommen.

Wenn Sie die Methode *Content* von *TQueryTableProducer* aufrufen, wird die Abfrage mit den Parametern ausgeführt, die in dem Abfrageobjekt gefunden wurden. Anschließend wird für die Anzeige der Datensätze in der resultierenden Datenmenge eine HTML-Tabelle formatiert.

Wie bei jedem Tabellengenerator können Sie auch hier die Anzeigeeigenschaften bzw. die Spaltenbindungen der HTML-Tabelle anpassen oder die Tabelle in ein größeres HTML-Dokument einbetten.

Server-Anwendungen testen

Da eine Web-Server-Anwendung als Antwort auf die Botschaften eines Web-Servers ausgeführt wird, können bei ihrem Test mit dem Debugger spezifische Probleme auftreten. Sie können die Anwendung nicht einfach von der IDE aus starten, da dann der Web-Server nicht in die Schleife einbezogen wird und die Anwendung die erwartete Anforderungsbotschaft nicht findet. In den folgenden Abschnitten wird erläutert, wie Sie eine Web-Server-Anwendung mit dem Debugger testen.

ISAPI- und NSAPI-Anwendungen testen

ISAPI- und NSAPI-Anwendungen sind effektiv DLLs, die vordefinierte Eintrittspunkte enthalten. Der Web-Server übergibt die Anforderungsbotschaften an die Anwendung, indem er diese Eintrittspunkte aufruft. Zum Start des Servers müssen Sie die Startparameter Ihrer Anwendung festlegen. Richten Sie Haltepunkte ein, damit ein normales Testen gewährleistet ist, wenn der Server eine Anforderung an die DLL übergibt.

Hinweis Stellen Sie vor dem Start des Web-Servers (mit Hilfe der Startparameter Ihrer Anwendung) sicher, daß der Server nicht bereits läuft.

Unter Windows NT testen

Unter Windows NT benötigen Sie zum Testen einer DLL die entsprechenden Benutzerrechte. Fügen Sie im Benutzer-Manager Ihren Namen in die Listen für folgende Benutzerrechte ein:

- Anmelden als Dienst
- Als Teil des Betriebsystems handeln
- Generieren von Sicherheitsüberwachung

Mit einem Microsoft IIS-Server testen

Wenn Sie zum Testen einer Web-Server-Anwendung einen Microsoft IIS-Server einsetzen (Version 3 oder früher), wählen Sie *Start / Parameter*, und legen Sie die Parameter der Anwendung wie folgt fest:

```
Host-Anwendung: c:\winnt\system32\inetsrv\inetinfo.exe
Startparameter:  -e w3svc
```

Dadurch starten Sie den IIS-Server und können Ihre ISAPI-DLL testen.

Hinweis Ihr Verzeichnispfad für inetinfo.exe muß nicht mit dem im obigen Beispiel identisch sein.

Wenn Sie einen Microsoft IIS-Server der Version 4 oder höher verwenden, müssen Sie vor dem Testen Einstellungen in der Registrierung und des IIS-Administrationsdienstes ändern:

1. Geben Sie mit Hilfe von DCOMCnfg IIS Admin Service als Benutzerkonto an.
2. Entfernen Sie mit dem Registrierungseditor (REGEDIT) oder einem anderen Dienstprogramm das Schlüsselwort *LocalService* aus allen IISADMIN-Unterschlüsseln unter HKEY_CLASSES_ROOT/ AppID und HKEY_CLASSES_ROOT/ CLSID. Dieses Schlüsselwort ist in folgenden Unterschlüsseln enthalten:

```
{61738644-F196-11D0-9953-00C04FD919C1} // IIS WAMREG admin Service
{9F0BD3A0-EC01-11D0-A6A0-00A0C922E752} // IIS Admin Crypto Extension
{A9E69610-B80D-11D0-B9B9-00A0C922E750} // IISADMIN Service
```

Entfernen Sie außerdem unter dem Knoten *AppID* das Schlüsselwort *RunAs* aus den beiden ersten dieser Unterschlüssel. Fügen Sie in den letzten Unterschlüssel *Interactive User* als Wert für das Schlüsselwort *RunAs* ein.

3 Fügen Sie mit dem Registrierungseditor *LocalService32*-Unterschlüssel in alle *IISADMIN*-Unterschlüssel unter dem Knoten *CLSID* ein. Dies betrifft alle in Schritt 2 aufgeführten Unterschlüssel sowie solche, unter denen Sie das Schlüsselwort *LocalService32* gefunden haben. Geben Sie einen *LocalService32*-Unterschlüssel unter dem Knoten *CLSID/<Unterschlüssel>* ein, und legen Sie als Vorgabewert für die neuen Schlüssel folgendes fest:

```
c:\\winnt\\system32\\inetsrv\\inetinfo.exe -e w3svc
```

(Der Verzeichnispfad für die Datei INETINFO.EXE kann von System zu System variieren.)

4 Fügen Sie für die folgenden Unterschlüssel den Wert *dword:3* für das Schlüsselwort *Start* hinzu:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IISADMIN]
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MSDTC]
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC]
```

5 Stoppen Sie die WWW-, FTP- und IISAdmin-Dienste in der Microsoft Management-Konsole oder im Dialogfeld *Dienste der Systemsteuerung*. Sie können zu diesem Zweck auch das Programm KILL.EXE im NT Resource Kit starten und KILL INETINFO ausführen.

Nun können Sie die Web-Server-Anwendung auf die gleiche Weise wie unter IIS Version 3 (oder früher) testen. Wählen Sie *Start / Parameter*, und legen Sie die Parameter der Anwendung wie folgt fest:

```
Host-Anwendung: c:\winnt\system32\inetsrv\inetinfo.exe
Startparameter:  -e w3svc
```

Hinweis Wenn der Testlauf abgeschlossen ist, müssen Sie alle Änderungen rückgängig machen, die Sie in den Schritten 2 bis 4 in der Registrierung vorgenommen haben.

Unter MTS testen

Eine andere Möglichkeit zum Testen einer Web-Server-Anwendung mit einem Microsoft IIS-Server besteht darin, das Web-Verzeichnis als MTS-Bibliotheks-Package zu konfigurieren. Sie können die ISAPI-DLL dann testen, indem Sie sie unter MTS ausführen.

Um das Web-Verzeichnis als MTS-Bibliotheks-Package zu konfigurieren, gehen Sie wie folgt vor:

1 Starten Sie Internet Service Manager. Es müssen die Verzeichnisstrukturen für Internet Information Server und für Microsoft Transaction Server angezeigt werden.

2 Erweitern Sie die Verzeichnisstruktur für Internet Information Server, damit die Einträge unter *Default Web Site (Standard-Web-Site)* angezeigt werden. Wählen Sie das Verzeichnis aus, in dem die ISAPI-DLL installiert ist. Klicken Sie mit der rechten Maustaste, und wählen Sie *Eigenschaften*.

3. Markieren Sie in der Registerkarte Virtual Directory (Virtuelles Verzeichnis) das Kontrollkästchen *Getrennter Speicherbereich* (isolierter Prozeß), und klicken Sie auf *OK*.
4. Erweitern Sie die Verzeichnisstruktur für Microsoft Transaction Server, um die Einträge unter *Packages Installed (Installierte Packages)* anzuzeigen. Klicken Sie mit der rechten Maustaste auf den *Knoten Packages Installed (Installierte Packages)*, und wählen Sie *Aktualisieren*.
5. Es wird ein Package angezeigt, das dieselbe Erweiterung wie das Web-Verzeichnis hat. Klicken Sie mit der rechten Maustaste auf dieses Package, und wählen Sie *Eigenschaften*.
6. Markieren Sie in der Registerkarte *Identität* das Optionsfeld Interaktiver Benutzer, und klicken Sie auf *OK*.

Nach Ausführung dieser Schritte ist Ihr Web-Verzeichnis konfiguriert, und Sie können mit dem Testen der ISAPI-DLL beginnen:

1. Wählen Sie in Delphi *Start / Parameter*. Geben Sie in das Feld Host-Anwendung den vollständigen Pfadnamen für die ausführbare MTS-Datei ein. Normalerweise lautet er folgendermaßen:

   ```
   c:\winnt\system32\mtx.exe
   ```

2. Geben Sie im Parameterfeld die Option */p* und den Namen des MTS-Package ein. Um diesen Wert zu erhalten, starten Sie Internet Service Manager und erweitern die Verzeichnisstruktur für Microsoft Transaction Server, damit die Einträge unter *Packages Installed (Installierte Packages)*angezeigt werden. Klicken Sie mit der rechten Maustaste auf das Package, das dieselbe Erweiterung wie das Web-Verzeichnis hat. Wählen Sie *Eigenschaften*, und kopieren Sie den Package-Namen aus der Registerkarte *Allgemein* in die Zwischenablage.

   Fügen Sie den Package-Namen in das Parameterfeld ein, und setzen Sie ihn in Anführungszeichen. Geben Sie dann vor dem String */p:* ein. Der Wert im Parameterfeld sollte dem folgendermaßen ähneln:

   ```
   /p:"IIS-{Default Web Site//ROOT/WEBPUB/DEMO}"
   ```

   Zwischen dem Doppelpunkt und dem Namen des Package darf kein Leerzeichen eingefügt werden.

Tip Wenn das Web-Verzeichnis als MTS-Package installiert ist, kann die DLL mit MTS heruntergefahren werden. Erweitern Sie im Internet Service Manager die Verzeichnisstruktur für Microsoft Transaction Server, damit die Einträge unter dem Knoten *Packages Installed (Installierte Packages*) angezeigt werden. Klicken Sie mit der rechten Maustaste auf das Package, das dieselbe Erweiterung wie das Web-Verzeichnis hat, und wählen Sie *Herunterfahren*.

Mit einem Windows 95 Personal-Web-Server testen

Wenn Sie zum Testen einer Web-Server-Anwendung einen Personal-Web-Server einsetzen, legen Sie die Startparameter der Anwendung wie folgt fest:

```
Host-Anwendung: c:\Program Files\websvc\system\inetsw95.exe
Startparameter:  -w3svc
```

Dadurch starten Sie den Personal-Web-Server und können Ihre ISAPI-DLL testen.

Hinweis Ihr Verzeichnispfad für *inetsw95.exe* muß nicht mit dem im obigen Beispiel identisch sein.

Mit Netscape Server 2.0 testen

Bevor Sie Web-Server-Anwendungen auf Netscape-Servern einsetzen können, sind einige Änderungen an der Konfiguration erforderlich.

Kopieren Sie zunächst die Datei ISAPITER.DLL (aus dem Verzeichnis Bin) in das Verzeichnis C:\Netscape\Server\Nsapi\Examples. (Unter Umständen weist Ihr Verzeichnis einen anderen Namen auf.)

Nehmen Sie anschließend die folgenden Änderungen an den Server-Konfigurationsdateien vor. Diese befinden sich im Verzeichnis C:\Netscape\Server\Httpd-<ServerName>\Config.

1 Fügen Sie in die Datei OBJ.CONF folgende Zeile ein:

```
Init funcs="handle_isapi,check_isapi,log_isapi" fn="load_modules"
   shlib="c:/netscape/server/nsapi/examples/ISAPIter.dll"
```

Diese Einfügung muß nach der folgenden Zeile erfolgen:

```
Init fn=load-types mime-types=mime.types
```

2 Fügen Sie in den Abschnitt <Object name=default> der Datei OBJ.CONF folgende Zeilen ein:

```
NameTrans from="/scripts" fn="pfx2dir" dir="C:/Netscape/Server/docs/scripts"
   name="isapi"
```

Diese Einfügung muß vor der folgenden Zeile erfolgen:

```
NameTrans fn=document-root root="C:/Netscape/Server/docs"
```

3 Fügen Sie den folgenden Abschnitt an das Ende von OBJ.CONF an:

```
<Object name="isapi">
PathCheck fn="check_isapi"
ObjectType fn="force-type" type="magnus-internal/isapi"
Service fn="handle_isapi"
</Object>
```

4 Fügen Sie den folgenden Abschnitt an das Ende der Datei MIME.TYPES an:

```
type=magnus-internal/isapi exts=dll
```

Dies muß die letzte Zeile der Datei sein.

Hinweis In den Schritten 1 und 2 wurden zur Verbesserung der Lesbarkeit Zeilenumbrüche eingefügt. Beim Eintragen dieser Zeilen in die Konfigurationsdateien dürfen aber keinesfalls Wagenrücklaufzeichen verwendet werden.

Wenn Sie zum Testen einer Web-Server-Anwendung einen Netscape Fast Track-Server einsetzen, legen Sie die Startparameter der Anwendung wie folgt fest:

```
Host-Anwendung: c:\Netscape\server\bin\httpd\httpd.exe
Startparameter:  c:\Netscape\server\httpd-<servername>\config
```

Diese Zeilen starten den Server und teilen ihm den Standort der Konfigurationsdateien mit.

CGI- und Win-CGI-Anwendungen testen

Das Testen von CGI- und Win-CGI-Anwendungen gestaltet sich etwas schwieriger, da hier die Anwendung vom Web-Server aus gestartet werden muß.

Den Server simulieren

Für Win-CGI-Anwendungen können Sie den Server simulieren, indem Sie eine Datei mit den Konfigurationseinstellungen schreiben, in der die Anforderungsinformationen enthalten sind. Anschließend starten Sie die Win-CGI-Anwendung. Dabei übergeben Sie die Datei mit den Client-Informationen und dem Standort der Datei, in der das Win-CGI-Programm seinen Inhalt ablegen soll. Anschließend können Sie den Testlauf wie gewohnt durchführen.

Als DLL testen

Ein weiterer Ansatz zum Testen von CGI- und Win-CGI-Anwendungen besteht darin, die Anwendung zunächst als ISAPI- oder NSAPI-Anwendung zu erstellen und zu testen. Wenn Ihre ISAPI- bzw. NSAPI-Anwendung zufriedenstellend läuft, konvertieren Sie diese in eine CGI- oder Win-CGI-Anwendung. Gehen Sie hierzu wie folgt vor:

1. Klicken Sie mit der rechten Maustaste auf das Web-Modul, und wählen Sie *Der Objektablage hinzufügen*.
2. Geben Sie im Dialogfeld *Der Objektablage hinzufügen* einen Titel, eine Beschreibung, eine Ablageseite (in der Regel Datenmodule), den Namen des Autors und ein Symbol für das Web-Modul an.
3. Wählen Sie *OK*, um das Web-Modul als Vorlage zu speichern.
4. Wählen Sie im Hauptmenü *Datei / Neu* und anschließend *Web Server Anwendung*. Wählen Sie dann im Dialogfeld *Neue Web-Server*-Anwendung je nach Bedarf entweder *CGI* oder *Win-CGI*.
5. Löschen Sie das automatisch generierte Web-Modul.
6. Wählen Sie im Hauptmenü *Datei / Neu*, und wählen Sie die in Schritt 3 gespeicherte Vorlage aus. Sie befindet sich auf der in Schritt 2 angegebenen Ablageseite.

CGI- und Win-CGI-Anwendungen sind einfacher aufgebaut als ISAPI- und NSAPI-Anwendungen. Jede Instanz einer CGI- oder Win-CGI-Anwendung muß nur einen einzigen Thread verwalten. Deshalb sind für diese Anwendungen alle Aspekte, die bei mehreren Threads berücksichtigt werden müssen, nicht von Belang, während sie von ISAPI- und NSAPI-Anwendungen berücksichtigt werden müssen. CGI- oder Win-CGI-Anwendungen sind außerdem immun gegen Probleme, die sich in ISAPI- und NSAPI-Anwendungen aus der Zwischenspeicherung der Web-Module ergeben können.

Kapitel

30

Arbeiten mit Sockets

Dieses Kapitel beschreibt Socket-Komponenten zur Erstellung von Anwendungen, die mit anderen Systemen über das TCP/IP-Protokoll (oder damit verwandten Protokollen) kommunizieren können. Sockets ermöglichen das Lesen und Schreiben über eine Verbindung zu anderen Maschinen, ohne daß die Gegebenheiten der verwendeten Netzwerk-Software im einzelnen berücksichtigt werden müssen. Die von den Sockets bereitgestellten Verbindungen basieren zwar auf dem TCP/IP-Protokoll, sind aber so universell gehalten, daß auch verwandte Protokolle wie Xerox Network System (XNS), DECnet von Digital oder die IPX/SPX-Familie von Novell verwendet werden können.

Die Verwendung von Sockets ermöglicht die Entwicklung von Netzwerk-Servern und Client-Anwendungen, die Daten aus anderen Systemen lesen bzw. in diese schreiben können. Ein Server oder eine Client-Anwendung arbeitet normalerweise mit einem einzelnen Dienst wie HTTP (Hypertext Transfer Protocol) oder FTP (File Transfer Protocol). Durch die Verwendung von Server-Sockets kann eine Anwendung, die einen dieser Dienste bereitstellt, eine Verbindung zu Client-Anwendungen herstellen, die diesen Dienst benutzen wollen. Client-Sockets ermöglichen es einer Anwendung, die einen dieser Dienste verwendet, eine Verbindung zu Server-Anwendungen herzustellen, die diesen Dienst bereitstellen.

Dienste implementieren

Sockets bilden einen Baustein bei der Entwicklung von Netzwerk-Servern und Client-Anwendungen. Für viele Dienste wie z.B. HTTP oder FTP sind fertige Server von Fremdherstellern verfügbar. Davon sind einige sogar in das Betriebssystem eingebunden, was Ihnen wiederum Entwicklungsarbeit abnimmt. Es gibt aber trotzdem Situationen, in denen die Erstellung eigener Server- oder Client-Anwendungen angebracht ist. Dies ist beispielsweise der Fall, wenn für den gewünschten Dienst kein Server verfügbar ist oder eine stärkere Kontrolle über die Implementierung des Dienstes benötigt wird. Oft läßt sich außerdem mit selbsterstellten Anwendungen ein höheres Maß an Integration zwischen der Anwendung und der Netzwerkkommunika-

tion erzielen. So könnte es bei Verwendung von verteilten Datenmengen sinnvoll sein, eine Schicht zu entwickeln, die eine Kommunikation mit Datenbanken in anderen Systemen ermöglicht.

Was sind Dienstprotokolle?

Für die Entwicklung eines Netzwerk-Servers oder Clients benötigen Sie Kenntnisse über den Dienst, den Ihre Anwendung bereitstellt bzw. nutzt. Viele Dienste verwenden Standardprotokolle, die Ihre Netzwerkanwendung unterstützen muß. Wenn Sie eine Anwendung für einen Standarddienst wie HTTP oder FTP schreiben, müssen Sie die Protokolle kennen, die von diesen Diensten zur Kommunikation mit anderen Systemen benutzt werden. Lesen Sie dazu in der Dokumentation des betreffenden Dienstes nach.

Falls Sie einen neuen Dienst für eine Anwendung bereitstellen wollen, die mit anderen Systemen kommuniziert, besteht der erste Arbeitsschritt in der Entwicklung eines Kommunikationsprotokolls, das die Server und Clients des Dienstes benutzen. Folgende Fragen sind dabei z.B. relevant: Welche Botschaften werden gesendet? Auf welche Weise müssen diese Botschaften koordiniert werden? Wie ist die Information codiert?

Mit Anwendungen kommunizieren

Ihr Netzwerk-Server oder Ihre Client-Anwendung fungiert häufig als Schicht zwischen der Netzwerk-Software und der Anwendung, die den Dienst verwendet. Beispielsweise befindet sich ein HTTP-Server zwischen dem Internet und einer Web-Server-Anwendung, die den Inhalt zur Verfügung stellt und auf HTTP-Anforderungsbotschaften antwortet.

Sockets stellen die Schnittstelle zwischen dem Netzwerk-Server bzw. der Client-Anwendung und der Netzwerk-Software bereit. Für die Schnittstelle zwischen Ihrer Anwendung und den Anwendungen, die sie verwenden, sind Sie selbst verantwortlich. Dazu können Sie die API des Standard-Servers eines Fremdherstellers (wie ISAPI) kopieren oder Ihre eigene API entwerfen und veröffentlichen.

Dienste und Schnittstellen

Die meisten Standarddienste sind per Konvention mit bestimmten Schnittstellennummern verbunden. Schnittstellennummern werden später genauer erläutert. Für den Augenblick reicht es, daß Sie die Schnittstellennummer als numerischen Code für den Dienst betrachten.

Bei der Implementierung eines Standarddienstes stellt das Windows-Socket-Objekt Methoden zur Verfügung, mit deren Hilfe Sie die Schnittstellennummer des Dienstes ermitteln können. Wenn Sie einen neuen Dienst bereitstellen, können Sie die verbundene Schnittstellennummer in einer Datei mit dem Namen SERVICES auf Windows 95- oder Windows NT-Maschinen angeben. In der Microsoft-Dokumentation zu Windows-Sockets finden Sie Informationen über die Einrichtung der Datei SERVICES.

Typen von Socket-Verbindungen

Socket-Verbindungen lassen sich in drei grundlegende Kategorien einteilen, die festlegen, wie die Verbindung initialisiert wird und womit der lokale Socket verbunden werden soll. Diese sind:

- Client-Verbindungen.
- Empfangende Verbindungen.
- Server-Verbindungen.

Wenn die Verbindung zu einem Client-Socket hergestellt wurde, läßt sich die Server-Verbindung von einer Client-Verbindung nicht mehr unterscheiden. Beide Endpunkte weisen die gleichen Merkmale auf und empfangen die gleichen Arten von Ereignissen. Nur die empfangende Verbindung ist grundlegend anders, da sie nur einen Endpunkt aufweist.

Client-Verbindungen

Client-Verbindungen verknüpfen einen Client-Socket auf dem lokalen System mit einem Server-Socket auf einem Remote-System. Client-Verbindungen werden durch den Client-Socket initiiert. Zuerst muß der Client-Socket den Server-Socket beschreiben, zu dem die Verbindung hergestellt werden soll. Der Client-Socket sucht dann den Server-Socket und fordert eine Verbindung an, sobald der Server gefunden ist. Möglicherweise kann der Server-Socket die Verbindung nicht sofort aufbauen. Server-Sockets verwalten eine Warteschlange mit Client-Anforderungen und stellen die Verbindungen her, sobald Zeit dazu vorhanden ist. Wenn der Server-Socket die Client-Verbindung akzeptiert, sendet er dem Client-Socket eine vollständige Beschreibung des Server-Sockets, mit dem er verbunden ist. Der Client vervollständigt daraufhin die Verbindung.

Empfangende Verbindungen

Server-Sockets suchen nicht nach Clients. Sie bauen passive »Halbverbindungen« auf, die auf Client-Anforderungen warten. Server-Sockets verwalten ihre empfangenden Verbindungen in einer Warteschlage. Diese zeichnet die eingehenden Verbindungsanforderungen von Clients auf. Wenn der Server-Socket eine Client-Verbindungsanforderung akzeptiert, baut er einen neuen Socket für die Verbindung mit dem Client auf. Dadurch kann die empfangende Verbindung geöffnet bleiben und auf weitere Client-Anforderungen reagieren.

Server-Verbindungen

Server-Verbindungen werden von Server-Sockets aufgebaut, wenn eine empfangende Socket eine Client-Anforderung akzeptiert. Eine Beschreibung des Server-Sockets, der die Verbindung zu dem Client vervollständigt, wird an den Client gesendet, so-

bald der Server die Verbindung akzeptiert. Die Verbindung wird eingerichtet, wenn der Client-Socket diese Beschreibung empfängt und die Verbindung vervollständigt.

Die Sockets beschreiben

Über Sockets kann Ihre Netzwerkanwendung mit anderen Systemen im Netz kommunizieren. Jeder Socket kann als Endpunkt in einer Netzwerkverbindung betrachtet werden. Er verfügt über eine Adresse, die folgende Informationen enthält:

- Das System, auf dem er ausgeführt wird.
- Die akzeptierten Schnittstellentypen.
- Die für die Verbindung verwendete Schnittstelle (Port).

Zur vollständigen Beschreibung einer Socket-Verbindung müssen die Adressen der Sockets auf beiden Seiten der Verbindung angegeben werden. Zur Beschreibung der Adresse jedes Socket-Endpunkts können Sie sowohl die IP-Adresse oder den Host-Namen als auch die Schnittstellennummer angeben.

Bevor Sie eine Socket-Verbindung aufbauen können, müssen Sie die Sockets für deren Endpunkte vollständig beschreiben. Einige der erforderlichen Informationen sind über das System verfügbar, auf dem die Anwendung läuft. Beispielsweise brauchen Sie die lokale IP-Adresse eines Client-Sockets nicht zu beschreiben, da diese Information über das Betriebssystem zur Verfügung steht.

Welche Informationen Sie bereitstellen müssen, hängt davon ab, mit welchem Socket-Typ Sie arbeiten. Client-Sockets müssen den Server beschreiben, mit dem eine Verbindung hergestellt werden soll. Empfangende Server-Sockets müssen die Schnittstelle beschreiben, die den von ihnen zur Verfügung gestellten Dienst darstellen.

Den Host beschreiben

Der Host ist das System, auf dem die Anwendung ausgeführt wird, die den Socket enthält. Sie können den Host für einen Socket beschreiben, indem Sie dessen IP-Adresse angeben. Hierbei handelt es sich um einen String mit vier numerischen (Byte-)Werten, der in Standard-Punktnotation für das Internet angegeben wird. Beispiel:

```
123.197.1.2
```

Ein System kann mehrere IP-Adressen unterstützen.

IP-Adressen sind oft nur schwer zu erkennen und relativ tippfehleranfällig. Eine Alternative dazu ist die Verwendung des Host-Namens. Host-Namen sind Aliasnamen für die IP-Adresse, wie sie häufig in URLs (Uniform Resource Locators) benutzt werden. Diese Strings enthalten einen Domänennamen und einen Dienst. Beispiel:

```
http://www.wSite.Com
```

Die meisten Intranetze stellen Host-Namen für die IP-Adressen der Systeme im Internet bereit. Wenn auf Windows 95- und Windows NT-Maschinen kein Host-Name verfügbar ist, können Sie einen Host-Namen für Ihre lokale IP-Adresse anlegen. Ge-

ben Sie dazu den Namen in die Datei HOSTS ein. In der Microsoft-Dokumentation zu Windows-Sockets finden Sie Informationen über die Datei HOSTS.

Server-Sockets müssen keinen Host angeben. Die lokale IP-Adresse kann aus dem System ermittelt werden. Wenn das lokale System mehrere IP-Adressen unterstützt, empfängt der Server-Socket Client-Anforderungen von allen IP-Adressen gleichzeitig. Akzeptiert ein Server-Socket eine Verbindung, stellt der Client-Socket die entfernte IP-Adresse bereit.

Client-Sockets müssen den Remote-Host entweder über den Host-Namen oder über die IP-Adresse angeben.

Zwischen einem Host-Namen und einer IP-Adresse wählen

Die meisten Anwendungen geben ein System über den Host-Namen an. Host-Namen sind leichter zu erkennen und einfacher auf Tippfehler zu überprüfen. Außerdem können Server das System oder die IP-Adresse ändern, die mit einem bestimmten Host-Namen verbunden sind. Bei Verwendung eines Host-Namens ist es dem Client-Socket möglich, den vom Host-Namen repräsentierten abstrakten Standort selbst dann zu finden, wenn er an eine neue IP-Adresse verschoben wurde.

Ist der Host-Name unbekannt, muß der Client-Socket das Server-System über seine IP-Adresse angeben. Die Angabe des Systems über die IP-Adresse stellt das schnellere Vorgehen dar. Bei Angabe des Host-Namens muß der Socket zur Lokalisierung des Server-Systems erst nach der mit dem Host-Namen verbundenen IP-Adresse suchen.

Schnittstellen verwenden

Die IP-Adresse stellt im Prinzip genügend Informationen für das Auffinden des Systems zur Verfügung, das sich am anderen Ende einer Socket-Verbindung befindet. Trotzdem benötigen Sie eine Schnittstellennummer für dieses System. Ohne Schnittstellennummern könnte ein System nur eine einzige Verbindung zur selben Zeit aufbauen. Schnittstellennummern sind eindeutige Bezeichner, die ein System in die Lage versetzen, mehrere Verbindungen gleichzeitig aufzubauen. Zu diesem Zweck wird jeder Verbindung eine eigene Schnittstellennummer zugeordnet.

Zuvor haben wir Schnittstellennummern als numerische Codes für Dienste beschrieben, die von Netzwerkanwendungen implementiert werden. Dabei handelt es sich lediglich um eine Konvention, die es empfangenden Server-Verbindungen ermöglicht, sich selbst an einer festen Schnittstellennummer verfügbar zu machen. Auf diese Weise können sie von Client-Sockets ausfindig gemacht werden. Server-Sockets erwarten Botschaften unter der Schnittstellennummer, die mit dem von ihnen bereitgestellten Dienst verbunden ist. Wenn sie eine Verbindung zu einem Client-Socket akzeptieren, erstellen sie eine separate Socket-Verbindung, die eine andere, willkürliche Schnittstellennummer verwendet. So kann die empfangende Verbindung weiterhin über die Schnittstellennummer empfangen, die mit dem Dienst verbunden ist.

Client-Sockets verwenden eine willkürliche, lokale Schnittstellennummer, weil sie von anderen Sockets nicht lokalisiert werden müssen. Sie geben die Schnittstellennummer des Server-Sockets an, mit dem sie verbunden werden wollen. Auf diese

Weise können Client-Sockets die Server-Anwendung finden. Häufig wird die Schnittstellennummer durch die Angabe des gewünschten Dienstes nur indirekt angegeben.

Socket-Komponenten

Die Seite Internet der Komponentenpalette stellt zwei Socket-Komponenten (Client Sockets und Server Sockets) zur Verfügung, mit deren Hilfe Ihre Netzwerkanwendung Verbindungen zu anderen Maschinen herstellen kann. Außerdem ermöglichen diese Komponenten das Lesen und Schreiben von Informationen über die betreffende Verbindung. Mit jeder dieser Socket-Komponenten sind Socket-Objekte von Windows verbunden, die den Endpunkt einer Socket-Verbindung darstellen. Die Socket-Komponenten verwenden Socket-Objekte von Windows, um API-Aufrufe der Windows-Sockets zu kapseln. Aus diesem Grund muß sich Ihre Anwendung nicht mit Einrichtungsdetails oder der Verwaltung von Socket-Botschaften befassen.

Wenn Sie mit API-Aufrufen der Windows-Sockets arbeiten wollen oder die Details derjenigen Verbindungen anpassen möchten, die die Socket-Komponenten für Sie durchführen, können Sie die Eigenschaften, Ereignisse und Methoden der Windows-Socket-Objekte verwenden.

Client-Sockets

Fügen Sie eine Client-Socket-Komponente (*TClientSocket*) in Ihr Formular oder Datenmodul ein, um Ihre Anwendung zu einem TCP/IP-Client zu machen. Client-Sockets erlauben Ihnen die Angabe des Server-Sockets, mit dem die Verbindung hergestellt werden soll, sowie die Angabe des Dienstes, den der betreffende Server bereitstellen soll. Sobald Sie die gewünschte Verbindung beschrieben haben, können Sie mit Hilfe des Client-Sockets die Verbindung zum Server vervollständigen.

Jede Client-Socket-Komponente verwendet ein Client-Socket-Objekt von Windows (*TClientWinSocket*) für die Repräsentation des Client-Endpunktes in einer Verbindung.

Den gewünschten Server angeben

Client-Socket-Komponenten verfügen über eine Reihe von Eigenschaften, welche die Angabe des Server-Systems und der Schnittstelle erlauben, zu der die Verbindung hergestellt werden soll. Mit der Eigenschaft *Host* können Sie das Server-System über seinen Host-Namen angeben. Wenn Sie den Host-Namen nicht kennen oder wenn die Geschwindigkeit beim Suchen des Servers eine Rolle spielt, geben Sie über die Eigenschaft *Address* die IP-Adresse des Servers an. Auf jeden Fall muß entweder der Host-Name oder die IP-Adresse angegeben werden. Bei Angabe beider Informationen verwendet der Client-Socket den Host-Namen

Zusätzlich zum Server-System muß auch die Schnittstelle zu dem Server-System angegeben werden, mit dem der Client-Socket verbunden werden soll. Die Schnittstellennummer des Servers läßt sich direkt (mit Hilfe der Eigenschaft *Port*) oder indirekt

(durch Nennung des gewünschten Dienstes in der Eigenschaft *Service*) angeben. Bei Angabe beider Informationen verwendet der Client-Socket den Namen des Dienstes.

Die Verbindung aufbauen

Legen Sie die Eigenschaften für den Client-Socket fest, um damit den Server zu beschreiben, zu dem die Verbindung hergestellt werden soll. Danach kann die Verbindung zur Laufzeit durch einen Aufruf der Methode *Open* aufgebaut werden. Wenn Ihre Anwendung die Verbindung beim Start automatisch aufbauen soll, setzen Sie die Eigenschaft *Active* zur Entwurfszeit im Objektinspektor auf *True*.

Legen Sie die Eigenschaften für den Client-Socket fest, um damit den Server zu beschreiben, zu dem die Verbindung hergestellt werden soll. Danach kann die Verbindung zur Laufzeit durch einen Aufruf der Methode *Open* aufgebaut werden. Wenn Ihre Anwendung die Verbindung beim Start automatisch aufbauen soll, setzen Sie die Eigenschaft *Active* zur Entwurfszeit im Objektinspektor auf *True*.

Informationen über die Verbindung ermitteln

Nachdem die Verbindung zu einem Socket-Server vervollständigt wurde, können Sie den Client-Windows-Socket, der mit Ihrer Client-Socket-Komponente verbunden ist, zur Ermittlung von Informationen über die Verbindung einsetzen. Mit Hilfe der Eigenschaft *Socket* erhalten Sie Zugriff auf das Windows-Socket-Objekt. Dieses Objekt besitzt Eigenschaften, mit denen Sie die Adresse und die Schnittstellennummer feststellen können, die vom Client-Server und den Server-Sockets zum Aufbau der Verbindungsendpunkte verwendet werden. Die Eigenschaft *SocketHandle* liefert ein Handle auf die Socket-Verbindung, die Sie bei API-Aufrufen des Windows-Sockets verwenden können. Über die Eigenschaft *Handle* kann man auf das Fenster zugreifen, das die Botschaften von der Socket-Verbindung empfängt. Die Eigenschaft *ASyncStyles* stellt fest, welche Botschaftstypen dieses Fenster-Handle empfängt.

Die Verbindung beenden

Wenn Sie die Kommunikation mit einer Server-Anwendung über die Socket-Verbindung beenden möchten, schließen Sie die Verbindung durch den Aufruf der Methode *Close*. Die Verbindung kann auch vom Server-Endpunkt aus geschlossen werden. In diesem Fall erhalten Sie im Ereignis *OnDisconnect* eine entsprechende Benachrichtigung.

Server-Sockets

Fügen Sie eine Server-Socket-Komponente (*TServerSocket*) in Ihr Formular oder Datenmodul ein, um aus Ihrer Anwendung einen TCP/IP-Server zu machen. Über Server-Sockets geben Sie den von Ihnen bereitgestellten Dienst oder die Schnittstelle an, über die Client-Anforderungen empfangen werden sollen. Außerdem können Sie mit der Server-Socket-Komponente Verbindungsanforderungen von Clients empfangen und akzeptieren.

Jede Server-Socket-Komponente verwendet ein Server-Socket-Objekt von Windows (*TServerWinSocket*), mit dem der Server-Endpunkt in einer empfangenden Verbindung dargestellt wird. Die Komponente setzt außerdem ein Server-Client-Socket-Objekt von Windows (*TServerClientWinSocket*) für den Server-Endpunkt jeder aktiven Verbindung zu einem Client-Socket ein, den der Server akzeptiert.

Die Schnittstelle angeben

Bevor ein Server-Socket Client-Anforderungen empfangen kann, müssen Sie die Schnittstelle angeben, an welcher der Empfang stattfinden soll. Dazu steht Ihnen die Eigenschaft *Port* zur Verfügung. Wenn Ihre Server-Anwendung einen Standarddienst bereitstellt, der per Konvention mit einer bestimmten Schnittstellennummer verbunden ist, läßt sich diese Nummer indirekt über die Eigenschaft *Service* angeben. Die Verwendung dieser Eigenschaft ist empfehlenswert, weil die Eingabe des Dienstes nicht so tippfehleranfällig ist wie die Eingabe der Schnittstellennummer. Wenn Sie beide Informationen eingeben, verwendet der Server-Socket den Namen des Dienstes.

Client-Anforderungen empfangen

Sobald Sie die Schnittstellennummer für den Server-Socket eingegeben haben, können Sie zur Laufzeit durch einen Aufruf der Methode *Open* eine empfangende Verbindung aufbauen. Wenn die Anwendung diese Verbindung beim Start automatisch herstellen soll, setzen Sie die Eigenschaft *Active* im Objektinspektor zur Entwurfszeit auf *True*.

Verbindungen zu Clients aufbauen

Eine empfangende Server-Socket-Komponente akzeptiert automatisch Client-Verbindungsanforderungen, sobald diese empfangen werden. Sie erhalten darüber jedesmal eine Benachrichtigung in einem Ereignis *OnClientConnect*.

Informationen über Verbindungen ermitteln

Sobald Sie mit Ihrem Server-Socket eine empfangende Verbindung geöffnet haben, können Sie das mit dem Server-Socket verbundene Server-Socket-Objekt von Windows zur Ermittlung von Informationen über die Verbindung einsetzen. Mit Hilfe der Eigenschaft *Socket* erhalten Sie Zugriff auf das Server-Socket-Objekt von Windows. Dieses Objekt besitzt Eigenschaften, mit denen Sie alle aktiven Verbindungen zu Client-Sockets ermitteln können, die von Ihrer Server-Socket-Komponente akzeptiert wurden. Die Eigenschaft *SocketHandle* liefert ein Handle auf die Socket-Verbindung, die Sie bei API-Aufrufen des Windows-Sockets verwenden können. Über die Eigenschaft *Handle* läßt sich auf das Fenster zugreifen, das die Botschaften von der Socket-Verbindung empfängt.

Jede aktive Verbindung zu einer Client-Anwendung wird von einem Server-Client-Socket-Objekt von Windows (*TServerClientWinSocket*) gekapselt. Sie können über die Eigenschaft *Connections* des Server-Client-Socket-Objekts auf diese Verbindungen zugreifen. Diese Objekte besitzen Eigenschaften, mit deren Hilfe Sie die Adresse und

Schnittstellennummer feststellen können, die vom Client und den Server-Sockets für den Aufbau der Verbindungsendpunkte verwendet werden. Die Eigenschaft *SocketHandle* liefert ein Handle auf die Socket-Verbindung, die Sie bei API-Aufrufen des Windows-Sockets verwenden können. Über die Eigenschaft *Handle* läßt sich auf das Fenster zugreifen, das die Botschaften von der Socket-Verbindung empfängt. Die Eigenschaft *ASyncStyles* stellt fest, welche Botschaftstypen dieses Fenster-Handle empfängt.

Die Server-Verbindung beenden

Wenn Sie die empfangende Verbindung schließen wollen, rufen Sie die Methode *Close* auf. Dadurch werden alle Verbindungen zu Client-Anwendungen geschlossen, und anstehende Verbindungen, die noch nicht akzeptiert wurden, werden verworfen. Danach wird die empfangende Verbindung geschlossen, so daß die Server-Socket-Komponente keine neuen Verbindungen mehr akzeptieren kann.

Wenn Clients ihre einzelnen Verbindungen zu Ihrem Server-Socket schließen, werden Sie durch das Ereignis *OnClientDisconnect* davon benachrichtigt.

Auf Socket-Ereignisse antworten

Beim Schreiben von Anwendungen, die Sockets verwenden, ist in der Regel das Erstellen der Ereignisbehandlungsroutinen für die Socket-Komponenten am aufwendigsten. Einige Sockets generieren zum Zeitpunkt, an dem das Lesen aus der bzw. das Schreiben in die Socket-Verbindung beginnen soll, Diese sind im Abschnitt »Lese- und Schreib-Ereignisse« auf Seite 30-12 beschrieben.

Client-Sockets empfangen ein *OnDisconnect*-Ereignis, wenn der Server eine Verbindung beendet, und Server-Socket empfangen ein *OnClientDisconnect*-Ereignis, wenn der Client eine Verbindung beendet.

Sowohl Client-Sockets als auch Server-Sockets generieren Fehler-Events, wenn Sie von der Verbindung Fehlerbotschaften empfangen.

Socket-Komponenten empfangen auch im Verlauf der Eröffnung und des vollständigen Aufbaus einer Verbindung eine Reihe von Ereignissen. Wenn Ihre Anwendung die Art der Socket-Eröffnung beeinflussen oder sofort nach dem Aufbau der Verbindung mit dem Lesen oder dem Schreiben beginnen soll, können Sie Ereignisbehandlungsroutinen schreiben, die auf diese Client-Ereignisse oder Server-Ereignisse reagieren.

Fehlerereignisse

Client-Sockets generieren ein *OnError*-Ereignis, wenn sie Fehlerbotschaften von der Verbindung empfangen. Server-Sockets hingegen generieren in diesem Fall ein *OnClientError*-Ereignis. Sie können zum Reagieren auf diese Fehlerbotschaften eine *OnError*- oder *OnClientError*-Ereignisbehandlungsroutine schreiben. An die Ereignisbehandlungsroutine werden folgende Informationen übergeben:

- Das Windows-Socket-Objekt, welches die Benachrichtigung über den Fehler empfangen hat
- Wie das Socket versucht hat, auf das Auftreten des Fehlers zu reagieren
- Der Fehlercode, den die Fehlerbotschaft geliefert hat

Sie können in der Ereignisbehandlungsroutine auf den Fehler reagieren und den Fehlercode auf 0 setzen, um zu verhindern, daß das Socket eine Exception auslöst.

Client-Ereignisse

Wenn ein Client-Socket eine Verbindung eröffnet, treten folgende Ereignisse ein:

1. Bevor versucht wird, das Server-Socket zu suchen, tritt ein *OnLookup*-Ereignis auf. Zu diesem Zeitpunkt können Sie keine Änderungen an den Eigenschaften *Host*, *Address*, *Port* oder *Service* des gefundenen Server-Sockets vornehmen. Sie können mit Hilfe der Eigenschaft *Socket* auf das Windows-Socket des Clients zugreifen und dessen Eigenschaft *SocketHandle* für Windows-API-Aufrufe verwenden, welche die Client-Eigenschaften des Sockets beeinflussen. Beispiel: Eine Festlegung der Schnittstellennummer in der Client-Anwendung läßt sich zu diesem Zeitpunkt, vor der Kontaktaufnahme mit dem Server, vornehmen.
2. Das Windows-Socket wird für Ereignisbenachrichtigung eingerichtet und initialisiert.
3. Nachdem das Server-Socket gefunden wurde, tritt ein *OnConnecting*-Ereignis ein. Zu diesem Zeitpunkt kann das Windows-Socket-Objekt, das über die Eigenschaft *Socket* zur Verfügung steht, Informationen zu dem Server-Socket zur Verfügung stellen, welches das andere Ende der Verbindung bildet. Hier besteht die erste Möglichkeit, die tatsächlich für die Verbindung verwendete Schnittstelle und die IP-Adresse zu ermitteln. Diese Informationen können von denen für das empfangende Socket abweichen, das die Verbindung akzeptiert hat.
4. Die Verbindungsanforderung wird vom Server akzeptiert und vom Client-Socket vollständig durchgeführt.
5. Nachdem die Verbindung aufgebaut wurde, tritt ein *OnConnect*-Ereignis ein. Wenn das Socket unmittelbar Lese- oder Schreibvorgänge über die Verbindung starten soll, müssen Sie hierzu eine entsprechende *OnConnect*-Ereignisbehandlungsroutine schreiben.

Server-Ereignisse

Mit Server-Socket-Komponenten können zwei Typen von Verbindungen hergestellt werden: empfangende Verbindungen und Verbindungen zu Client-Anwendungen. Das Server-Socket empfängt bei Herstellung einer dieser Verbindungen jeweils Ereignisse.

Ereignisse bei empfangenden Verbindungen

Direkt vor dem Aufbau der empfangenden Verbindung tritt das Ereignis *OnListen* auf. Zu diesem Zeitpunkt können Sie das Windows-Socket-Objekt des Servers über die Eigenschaft *Socket* ermitteln. Über dessen Eigenschaft *SocketHandle* lassen sich Änderungen am Socket vornehmen, bevor dieses für die empfangende Verbindung geöffnet wird. Beispiel: Wenn Sie festlegen möchten, daß der Server nur für bestimmte IP-Adressen empfangsbereit sein soll, können Sie dies über eine *OnListen*-Ereignisbehandlungsroutine festlegen.

Ereignisse bei Client-Verbindungen

Wenn ein Server-Socket eine Anforderung für eine Client-Verbindung akzeptiert, treten die folgenden Ereignisse auf:

1. Das Server-Socket generiert ein *OnGetSocket*-Ereignis, wobei der Windows-Socket-Handle für das Socket übergeben wird, welches den Server-Endpunkt der Verbindung bildet. Wenn Sie einen eigenen, individuell angepaßten Nachkommen des Objekts *TServerClientWinSocket* zur Verfügung stellen möchten, können Sie diesen in einer *OnGetSocket*-Ereignisbehandlungsroutine erstellen; dieses Objekt wird dann an Stelle von *TServerClientWinSocket* verwendet.
2. Es tritt ein *OnAccept*-Ereignis ein, welches das neue *TServerClientWinSocket*-Objekt an die Ereignisbehandlungsroutine übergibt. Hier können Sie zum ersten Mal die Eigenschaften von *TServerClientWinSocket* zum Ermitteln von Informationen über den Server-Endpunkt der Verbindung zu einem Client verwenden.
3. Wenn für die Eigenschaft *ServerType* der Wert *stThreadBlocking* gesetzt ist, tritt ein *OnGetThread*-Ereignis ein. Wenn Sie einen eigenen, individuell angepaßten Nachkommen des Objekts *TServerClientThread* zur Verfügung stellen möchten, können Sie dieses in einer *OnGetThread*-Ereignisbehandlungsroutine erstellen. Dieses Objekt wird dann an Stelle von *TServerClientThread* verwendet. Weitere Informationen zum Erstellen von benutzerdefinierten Server-Client-Threads finden Sie im Abschnitt »Server-Threads schreiben« auf Seite 30-15.
4. Wenn für die Eigenschaft *ServerType* der Wert *stThreadBlocking* gesetzt ist, tritt ein *OnThreadStart*-Ereignis ein, sobald die Ausführung des Threads beginnt. Wenn Sie eine Initialisierung des Threads durchführen oder API-Aufrufe des Windows-Sockets durchführen möchten, bevor der Thread mit dem Lesen oder Schreiben über die Verbindung beginnt, müssen Sie dies in der *OnThreadStart*-Ereignisbehandlungsroutine angeben.
5. Der Client baut die Verbindung vollständig auf, und ein *OnClientConnect*-Ereignis tritt ein. Bei einem nicht-blockierenden Server können zu diesem Zeitpunkt die Lese- und Schreibvorgänge über die Socket-Verbindung beginnen.

Informationen über Socket-Verbindungen lesen und schreiben

Sie bauen Socket-Verbindungen zu anderen Maschinen auf, damit Sie Informationen über diese Verbindungen lesen und speichern können. Um welche Informationen es sich dabei handelt, und zu welchem Zeitpunkt Sie diese ermitteln oder speichern, hängt von dem mit der Socket-Verbindung verbundenen Dienst ab.

Das Lesen und Schreiben von Informationen über Sockets kann asynchron geschehen. Dadurch wird die Ausführung von anderem Code in der Netzwerkanwendung nicht blockiert. Dies wird als nicht-blockierende Verbindung bezeichnet. Sie können aber auch blockierende Verbindungen aufbauen, in denen Ihre Anwendungen mit der Ausführung der nächsten Codezeile so lange warten, bis der Lese- oder Schreibvorgang beendet ist.

Nicht-blockierende Verbindungen

Bei nicht-blockierenden Verbindungen werden die Lese- und Schreibvorgänge asynchron durchgeführt, so daß die Übertragung der Daten die Ausführung von anderem Code in der Netzwerkanwendung nicht blockiert. So erstellen Sie eine nicht-blockierende Verbindung:

- Setzen Sie für Client-Sockets die Eigenschaft *ClientType* auf *ctNonBlocking*.
- Setzen Sie für Server-Sockets die Eigenschaft *ServerType* auf *stNonBlocking*.

Wenn es sich um eine nicht-blockierende Verbindung handelt, wird das Socket über Lese- und Schreibereignisse darüber informiert, daß das Socket am anderen Ende der Verbindung versucht, Informationen zu lesen oder zu schreiben.

Lese- und Schreib-Ereignisse

Nicht-blockierende Sockets generieren Lese- und Schreibereignisse, die Ihr Socket darüber informieren, wann es Lese- und Schreibvorgänge über die Verbindung durchführen muß. Bei Client-Sockets können Sie auf diese Benachrichtigungen in einer *OnRead*- oder *OnWrite*-Ereignisbehandlungsroutine reagieren, bei Server-Sockets in einer *OnClientRead*- oder *OnClientWrite*-Ereignisbehandlungsroutine.

Das mit der Socket-Verbindung verbundene Socket-Objekt von Windows wird in den Behandlungsroutinen für die Lese- und Schreibereignisse als Parameter übergeben. Dieses Windows-Socket-Objekt besitzt eine Reihe von Methoden, mit deren Hilfe Sie Informationen über die Verbindung lesen und schreiben können.

Zum Lesen aus einer Socket-Verbindung stehen Ihnen die Methoden *ReceiveBuf* oder *ReceiveText* zur Verfügung. Vor der Verwendung der Methode *ReceiveBuf* ermitteln Sie mit Hilfe der Methode *ReceiveLength* eine Schätzung der Bytes-Anzahl, die der Socket auf der anderen Seite der Verbindung versenden will.

Mit den Methoden *SendBuf*, *SendStream* oder *SendText* können Sie in die Socket-Verbindung schreiben. Wenn Sie nach dem Schreiben der Informationen über den Socket die Socket-Verbindung nicht mehr benötigen, verwenden Sie die Methode *SendStreamThenDrop*.*SendStreamThenDrop* schließt die Socket-Verbindung, sobald alle In-

formationen, die aus dem Stream gelesen werden können, gespeichert sind. Bei Verwendung der Methode *SendStream* oder *SendStreamThenDrop* dürfen Sie das Stream-Objekt nicht freigeben. Der Socket gibt nämlich den Stream automatisch frei, wenn die Verbindung geschlossen wird.

Hinweis *SendStreamThenDrop* schließt nur Server-Verbindungen zu einem einzelnen Client, nicht aber empfangende Verbindungen.

Blockierende Verbindungen

Wenn eine Verbindung blockiert ist, muß der Socket den Lese- und Schreibvorgang für Informationen über die Verbindung initiieren, anstatt passiv auf eine Benachrichtigung von der Socket-Verbindung zu warten. Arbeiten Sie mit einem blockierenden Socket, wenn Ihr Ende der Verbindung für das Lesen und Schreiben verantwortlich ist.

Setzen Sie für Client-Sockets die Eigenschaft *ClientType* auf *ctBlocking*, um eine blokkierende Verbindung aufzubauen. In Abhängigkeit von anderen Aktivitäten Ihrer Client-Anwendung könnten Sie einen neuen Thread für das Lesen und Schreiben erstellen. Sie geben damit Ihrer Anwendung die Möglichkeit, einen Code in anderen Threads auszuführen, während Sie auf das Ende des Lese- und Schreibvorgangs wartet.

Setzen Sie für Server-Sockets die Eigenschaft *ServerType* auf *stThreadBlocking*, um eine blockierende Verbindung aufzubauen. Blockierende Verbindungen unterbrechen die Ausführung vom Code, während der Socket auf die zu schreibenden oder zu lesenden Informationen über die Verbindung wartet. Aus diesem Grund spalten Server-Socket-Komponenten immer einen neuen Thread für jede Client-Verbindung ab, wenn die Eigenschaft *ServerType* auf *stThreadBlocking* gesetzt ist.

Auch wenn keine Threads verwendet werden, können Lese- und Schreibvorgänge mit Hilfe von *TWinSocketStream* durchgeführt werden.

Verwenden von Threads bei blockierenden Verbindungen

Client-Sockets spalten beim Lesen oder Schreiben über eine blockierende Verbindung nicht automatisch neue Threads ab. Wenn die Client-Anwendung keine anderen Aktionen ausführen muß, bis der Lese- oder Schreibvorgang beendet ist, ist dies genau der Zustand, den Sie wünschen. Falls die Anwendung aber eine Benutzerschnittstelle enthält, die noch auf Benutzeranforderungen antworten muß, könnten Sie einen separaten Thread für das Lesen erstellen.

Wenn Server-Sockets blockierende Verbindungen aufbauen, spalten sie immer separate Threads für jede Client-Verbindung ab. So muß kein Client warten, bis ein anderer Client mit dem Lesen und Schreiben von Informationen über die Verbindung fertig ist. Standardmäßig verwenden Server-Sockets *TServerClientThread*-Objekte zur Implementierung des Ausführungs-Threads für die jeweilige Verbindung.

TServerClientThread-Objekte simulieren die Ereignisse *OnClientRead* und *OnClientWrite*, die bei nicht-blockierenden Verbindungen auftreten. Diese Ereignisse treten jedoch am empfangenden Socket auf, an dem der Thread nicht lokal ausgeführt wird.

Wenn häufig Client-Anforderungen auftreten, ist es sinnvoll, einen eigenen Nachkommen des Objekts *TServerClientThread* zu erstellen, um thread-sichere Lese- und Schreibvorgänge sicherzustellen.

Verwenden von TWinSocketStream

Beim Implementieren des Threads für eine blockierende Verbindung müssen Sie ermitteln, wann das Socket am anderen Ende der Verbindung zum Lesen oder Schreiben bereit ist. Blockierende Verbindungen benachrichtigen das Socket nicht, wann Daten gelesen oder geschrieben werden sollen. Zum Erkennen, wann die Verbindung bereit ist, dient ein *TWinSocketStream*-Objekt. *TWinSocketStream* stellt Methoden zur Verfügung, mit denen die zeitliche Festlegung von Lese- und Schreibvorgängen koordiniert werden kann. Rufen Sie die Methode *WaitForData* auf, damit so lange gewartet wird, bis das Socket am anderen Ende für den Schreibvorgang bereit ist.

Beim Lesen und Schreiben mit Hilfe von *TWinSocketStream* tritt im Stream ein Zeitüberlauf auf, wenn das Lesen und Schreiben nicht nach einer festgelegten Zeit beendet wird. Dieser Zeitüberlauf führt dazu, daß die Socket-Anwendung nicht endlos versucht, über eine nicht mehr bestehende Verbindung zu lesen oder zu schreiben.

Hinweis Der Einsatz von *TWinSocketStream* für eine nicht-blockierende Verbindung ist nicht möglich.

Client-Threads schreiben

Wenn Sie einen Thread für Client-Verbindungen schreiben, müssen Sie mit Hilfe des Dialogfensters *Neues Thread-Objekt* einen neuen Thread definieren. Weitere Informationen hierzu finden Sie im Abschnitt »Thread-Objekte definieren« auf Seite 8-2.

Die Methode *Execute* Ihres neuen Thread-Objekts verarbeitet die Details der Lese- und Schreibvorgänge über die Thread-Verbindung. Sie erstellt ein *TWinSocketStream*-Objekt und verwendet dieses zum Lesen oder Schreiben. Beispiel:

```
procedure TMyClientThread.Execute;
var
  TheStream: TWinSocketStream;
  buffer: string;
begin
  { ein TWinSocketStream-Objekt für Lese- und Schreibvorgänge erstellen }
  TheStream := TWinSocketStream.Create(ClientSocket1.Socket, 60000);
  try
    { Befehle abrufen und verarbeiten, bis die Verbindung oder der Thread beendet ist }
    while (not Terminated) and (ClientSocket1.Active) do
    begin
      try
        GetNextRequest(buffer); { GetNextRequest muß eine thread-sichere Methode sein }
        { die Anforderung auf den Server schreiben }
        TheStream.Write(buffer, Length(buffer) + 1);
        { die Kommunikation fortsetzen (z. B. eine Antwort vom Server lesen) }
        ...
      except
        if not(ExceptObject is EAbort) then
          Synchronize(HandleThreadException); { Sie müssen die HandleThreadException
                                                schreiben }
```

```
      end;
    end;
  finally
   TheStream.free;
  end;
end;
```

Damit Sie Ihren Thread verwenden können, müssen Sie ihn in einer *OnConnect*-Ereignisbehandlungsroutine erstellen. Nähere Informationen zum Erstellen und Ausführen von Threads finden Sie im Abschnitt »Thread-Objekte ausführen« auf Seite 8-11.

Server-Threads schreiben

Threads für Server-Verbindungen sind Nachkommen des Objekts *TServerClientThread*. Aus diesem Grund können Sie das Dialogfenster *Neues Thread-Objekt* zum Schreiben von Server-Threads nicht verwenden. Deklarieren Sie statt dessen Ihren Thread auf folgende Art manuell:

```
TMyServerThread = class(TServerClientThread);
```

Zum Implementieren dieses Threads verwenden Sie die Methode *ClientExecute* an Stelle der Methode *Execute*.

Beim Implementieren der Methode *ClientExecute* gehen Sie größtenteils so vor wie beim Schreiben der Methode *Execute* für den Thread einer Client-Verbindung. Jedoch müssen Sie an Stelle einer Client-Socket-Komponente, die Sie aus der Komponentenpalette in Ihre Anwendung einfügen, für den Server-Client-Thread das *TServerClientWinSocket*-Objekt verwenden, das beim Akzeptieren einer Client-Verbindung durch ein empfangendes Server-Socket erzeugt wird. Dies steht über die öffentliche Eigenschaft *ClientSocket* zur Verfügung. Zusätzlich können Sie die geschützte Methode *HandleException* verwenden, anstatt eigene thread-sichere Exception-Behandlungsroutinen zu schreiben. Beispiel:

```
procedure TMyServerThread.ClientExecute;
var
  Stream : TWinSocketStream;
  Buffer : array[0 .. 9] of Char;
begin
  { Sicherstellen, daß Verbindung aktiv ist }
  while (not Terminated) and ClientSocket.Connected do
  begin
    try
      Stream := TWinSocketStream.Create(ClientSocket, 60000);
      try
        FillChar(Buffer, 10, 0); { Puffer initialisieren }
        { Dem Client 60 Sekunden bis zum Starten des Schreibvorgangs Zeit lassen }
        if Stream.WaitForData(60000) then
        begin
          if Stream.Read(Buffer, 10) = 0 then { wenn der Lesevorgang nicht innerhalb von 60
                                                stattfindet }
            ClientSocket.Close; { Verbindung beenden }
          { Die Anforderung jetzt verarbeiten }
          ...
```

```
          end
          else
            ClientSocket.Close; { Verbindung beenden, wenn der Client nicht startet }
        finally
          Stream.Free;
        end;
      except
        HandleException;
      end;
    end;
  end;
```

Warnung Server-Sockets führen eine Zwischenspeicherung der von ihnen verwendeten Threads durch. Stellen Sie sicher, daß die Methode *ClientExecute* die erforderlichen Initialisierungsvorgänge durchführt, so daß durch bei der letzten Ausführung des Threads aufgetretene Änderungen keine unerwünschten Ergebnisse auftreten.

Damit Sie Ihren Thread verwenden können, müssen Sie ihn in einer *OnGetThread*-Ereignisbehandlungsroutine erstellen. Setzen Sie beim Erstellen des Threads den Parameter *CreateSuspended* auf *False*.

Teil

IV

Benutzerdefinierte Komponenten erzeugen

Die Kapitel in Teil IV dieses Handbuchs zeigen Konzepte und Techniken auf, die für den Entwurf und die Implementierung von Komponenten in Delphi erforderlich sind.

Kapitel

31

Die Komponentenentwicklung im Überblick

Dieses Kapitel gibt einen Überblick über die Entwicklung von Komponenten für Delphi-Anwendungen. Mit Delphi und seinen Standardkomponenten sollten Sie an dieser Stelle bereits vertraut sein.

Folgende Themen werden behandelt:

- Die Bibliothek visueller Komponenten
- Komponenten und Klassen
- Wie werden Komponenten erzeugt?
- Was zeichnet Komponenten aus?
- Neue Komponenten erzeugen
- Komponenten vor der Installation testen

Weitere Informationen zur Installation neuer Komponenten finden Sie unter »Komponenten-Packages installieren« auf Seite 9-6.

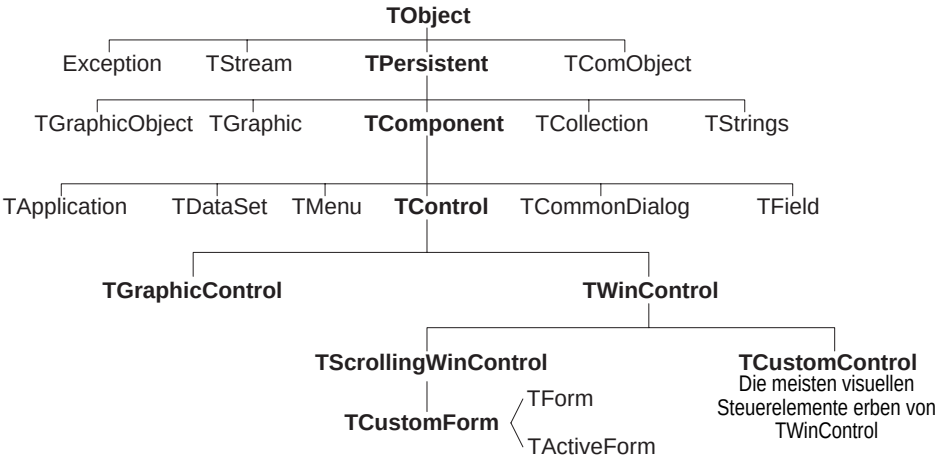
Die Bibliothek visueller Komponenten

Delphi-Komponenten sind Teil einer Klassenhierarchie, die als Bibliothek visueller Komponenten (Visual Component Library = VCL) bezeichnet wird. Abbildung 31.1 zeigt die Beziehung zwischen den einzelnen Klassen, aus denen sich die VCL zusammensetzt. Eine detailierte Behandlung der Klassenhierarchien und der Vererbung zwischen den Klassen finden Sie in Kapitel 32, »Objektorientierte Programmierung für Komponentenentwickler«.

Die Klasse *TComponent* ist der gemeinsame Vorfahr aller Komponenten der VCL. *TComponent* stellt die Grundausstattung an Eigenschaften und Ereignissen bereit, die

eine Komponente in der Delphi-Umgebung benötigt. Auf jeder Stufe der Hierarchie kommen dann weitere, individuelle Fähigkeiten hinzu.

Abbildung 31.1 Die Klassenhierarchie der VCL



Wenn Sie eine Komponente entwickeln, fügen Sie sie der VCL hinzu, indem Sie von einer bestehenden Klasse eine neue Klasse ableiten.

Komponenten und Klassen

Da Komponenten Klassen sind, gehen Komponentenentwickler anders vor als Anwendungsprogrammierer. Neue Komponenten erfordern stets die Ableitung neuer Klassen.

Es gibt zwei grundlegende Unterschiede zwischen der Entwicklung von Komponenten und ihrer Verwendung in einer Anwendung. Wenn Sie eine Komponente entwikkeln,

- haben Sie Zugriff auf Bereiche der Klasse, die einem Anwendungsprogrammierer nicht zugänglich sind;
- fügen Sie der Komponente neue Bestandteile hinzu (z. B. Eigenschaften).

Diese Unterschiede bringen jedoch auch zusätzliche Konventionen mit sich. Als Komponentenentwickler müssen Sie sich in erster Linie darüber Gedanken machen, wie die Anwendungsprogrammierer Ihre Komponenten einsetzen werden.

Wie werden Komponenten erzeugt?

Eine Komponente kann nahezu jedes beliebige Programmelement sein, das zur Entwurfszeit manipuliert werden kann. Um eine neue Komponente zu erzeugen, leiten Sie von einer bestehenden Klasse eine neue Klasse ab. Sie können neue Komponenten einfach von vorhandenen Komponenten ableiten. Meist werden Sie jedoch Komponenten erzeugen, indem Sie

- bestehende Steuerelemente verändern;
- fensterorientierte Steuerelemente erzeugen;
- grafische Steuerelemente erzeugen;
- neue Klassen von Windows-Steuerelementen ableiten;
- nichtvisuelle Komponenten erzeugen.

Tabelle 31.1 faßt die verschiedenen Komponentenarten und ihre Ausgangsklassen zusammen.

Tabelle 31.1 Ausgangsklassen für neue Komponenten

Verfahren	Ausgangsklasse
Ändern einer bestehenden Komponente	Jede bestehende Komponente (z. B. *TButton* oder *TListBox*) oder ein abstrakter Komponententyp (z. B. *TCustomListBox)*
Erzeugen einer fensterorientierten Komponente	*TWinControl*
Erzeugen eines grafischen Steuerelements	*TGraphicControl*
Ableiten einer Klasse von einem Windows-Steuerelement	Jedes beliebige Windows-Steuerelement
Erzeugen einer nichtvisuellen Komponente	*TComponent*

Sie können auch Klassen ableiten, die keine Komponenten sind und nicht in einem Formular manipuliert werden können. In Delphi gibt es viele solcher Klassen, beispielsweise *TRegIniFile* und *TFont*.

Vorhandene Steuerelemente modifizieren

Die einfachste Möglichkeit, eine Komponente zu erzeugen, besteht darin, eine vorhandene Komponente zu modifizieren. Von allen Komponenten, die Delphi bereitstellt, können neue Komponenten abgeleitet werden.

Manche Steuerelemente, beispielsweise Listenfelder und Gitter, existieren in verschiedenen Ausprägungen, die ein und dasselbe Grundthema variieren. In diesen Fällen enthält die VCL eine abstrakte Klasse, deren Name das Wort »Custom« enthält (z. B. *TCustomGrid*) und die als Grundlage für abgeleitete Versionen verwendet werden kann.

Nehmen wir an, Sie wollen ein spezielles Listenfeld erzeugen, für das einige Eigenschaften der Standardklasse *TListBox* überflüssig sind. Da Sie keine Eigenschaften einer Vorfahrklasse entfernen oder verbergen können, müssen Sie als Ausgangspunkt eine Klasse wählen, die in der Hierarchie irgendwo über *TListBox* steht. Damit Sie nicht bei der abstrakten Klasse *TWinControl* beginnen und alle Funktionen eines Listenfeldes neu erfinden müssen, gibt es in der VCL die Klasse *TCustomListBox*, die die Eigenschaften eines Listenfeldes implementiert, sie aber nicht alle als **published** deklariert. Wenn Sie von einer abstrakten Klasse wie *TCustomListBox* eine Komponente ableiten, deklarieren Sie nur diejenigen Eigenschaften als **published**, die in Ihrer Komponente verfügbar sein sollen. Der Rest bleibt weiterhin als **protected** deklariert.

In Kapitel 33, »Eigenschaften erstellen«, wird erläutert, wie geerbte Eigenschaften als **published** deklariert werden. Kapitel 39, »Vorhandene Komponenten modifizieren« und Kapitel 41, »Gitter anpassen«, zeigen, wie vorhandene Steuerelemente verändert werden.

Fensterorientierte Steuerelemente erzeugen

Fensterorientierte Steuerelemente sind Objekte, die zur Laufzeit sichtbar sind und mit denen der Benutzer interagieren kann. Jedes fensterorientierte Steuerelement hat ein Fenster-Handle, das über die Eigenschaft *Handle* abgefragt wird und das Steuerelement gegenüber Windows identifiziert. Steuerelemente, die ein Handle haben, können den Eingabefokus erhalten. Außerdem kann das Handle an Funktionen der Windows-API übergeben werden.

Alle fensterorientierten Steuerelemente stammen von der Klasse *TWinControl* ab. Dazu gehören die meisten der unter Windows üblichen Standard-Steuerelemente wie z.B. Schaltflächen, Listenfelder und Eingabefelder. Obgleich Sie die Möglichkeit haben, ein völlig neues Steuerelement (also eines, das nicht von einem vorhandenen Steuerelement abstammt) direkt von *TWinControl* abzuleiten, bietet Ihnen Delphi zu diesem Zweck die Klasse *TCustomControl* an. *TCustomControl* ist ein spezielles fensterorientiertes Steuerelement, das über komplexe grafische Funktionen verfügt.

Kapitel 41, »Gitter anpassen«, enthält ein Beispiel dafür, wie ein fensterorientiertes Steuerelement erzeugt wird.

Grafische Steuerelemente erzeugen

Wenn Sie eine Komponente entwickeln, die keinen Eingabefokus zu erhalten braucht, erzeugen Sie am besten ein grafisches Steuerelement. Grafische Steuerelemente ähneln fensterorientierten Steuerelementen, haben aber kein Fenster-Handle und verbrauchen daher weniger Systemressourcen. Ein Beispiel für ein grafisches Steuerelement ist die Komponente *TLabel*. Obwohl solche Steuerelemente nie den Eingabefokus erhalten können, ist es möglich, sie auf Maus-Ereignisse reagieren zu lassen.

Delphi unterstützt Sie beim Entwickeln von benutzerdefinierten Steuerelementen durch die Komponente *TGraphicControl*. *TGraphicControl* ist eine abstrakte Klasse, die von *TControl* abgeleitet ist. Obwohl Sie Steuerelemente auch direkt von *TControl* ab-

leiten können, ist *TGraphicControl* die bessere Wahl, da diese Klasse bereits über eine Zeichenfläche verfügt und *WM_PAINT*-Botschaften bearbeitet. Sie brauchen nur noch die Methode *Paint* zu überschreiben.

Kapitel 40, »Grafische Komponenten erzeugen« enthält ein Beispiel dafür, wie ein grafisches Steuerelement erzeugt wird.

Unterklassen von Windows-Steuerelementen erzeugen

In der traditionellen Windows-Programmierung werden benutzerdefinierte Steuerelemente erzeugt, indem eine neue Fensterklasse definiert und bei Windows registriert wird. Diese Fensterklasse (sie ähnelt den Objekten oder Klassen in der objektorientierten Programmierung) enthält Informationen, die alle Instanzen einer bestimmten Art von Steuerelement gemeinsam verwenden. Sie können auf der Grundlage einer bestehenden Klasse eine neue Fensterklasse erzeugen (die man als Unterklasse bezeichnet). Anschließend speichern Sie das neue Steuerelement, wie unter Windows üblich, in einer DLL und stellen eine Schnittstelle dafür bereit.

Mit Delphi können Sie jede bestehende Fensterklasse in einer Komponente kapseln. Wenn Sie also bereits über eine Bibliothek mit benutzerdefinierten Steuerelementen verfügen, die Sie in Delphi-Anwendungen einsetzen wollen, erzeugen Sie daraus einfach Delphi-Komponenten, die sich genau wie diese Steuerelemente verhalten. Wie gewohnt können Sie auch von solchen Komponenten wieder neue Steuerelemente ableiten.

Beispiele für die Techniken, mit denen Unterklassen von Windows-Steuerelementen erzeugt werden, finden Sie in der Unit STDCTLS (diese Datei enthält Standard-Windows-Steuerelemente, wie z. B. *TEdit*).

Nichtvisuelle Komponenten erzeugen

Nichtvisuelle Komponenten dienen als Schnittstellen zu Elementen wie etwa Datenbanken (*TDataSet*) oder Zeitgebern (*TTimer*) und als Platzhalter für Dialogfelder (*TCommonDialog* und seine Nachkommen). Als Komponentenentwickler werden Sie in erster Linie visuelle Steuerelemente erzeugen. Nichtvisuelle Komponenten können direkt von *TComponent* abgeleitet werden, der abstrakten Basisklasse für alle Komponenten.

Was zeichnet Komponenten aus?

Damit Ihre Komponenten in der Delphi-Umgebung zuverlässig funktionieren, müssen Sie als Entwickler gewisse Konventionen beachten. Dieser Abschnitt erläutert folgende Themen:

- Abhängigkeiten vermeiden
- Eigenschaften, Ereignisse und Methoden
- Grafik kapseln

- Registrierung

Abhängigkeiten vermeiden

Komponenten sind dann besonders gut verwendbar, wenn ihre Ausführung und Funktionalität innerhalb des Quelltextes nicht durch Beschränkungen eingegrenzt wird. Es liegt in der Natur von Komponenten, daß sie in einer Anwendung in den verschiedensten Kombinationen und in wechselndem Kontext vorkommen. Sie als Entwickler sind verantwortlich dafür, daß sich Ihre Komponenten ohne Vorbedingung in jeder Situation korrekt verhalten.

Ein schönes Beispiel für die Vermeidung von Abhängigkeiten stellt die Eigenschaft *Handle* von *TWinControl* dar. Wenn Sie schon einmal eine Windows-Anwendung geschrieben haben, wissen Sie, daß eines der fehleranfälligsten Probleme darin besteht, daß Sie auf ein Fenster nur dann zugreifen dürfen, wenn Sie es vorher durch einen Aufruf der API-Funktion *CreateWindow* erzeugt haben. Bei fensterorientierten Steuerelementen in Delphi taucht diese Schwierigkeit erst gar nicht auf. Delphi sorgt nämlich automatisch dafür, daß ein gültiges Fenster-Handle verfügbar ist, wenn es gebraucht wird. Indem das Handle durch eine Eigenschaft repräsentiert wird, kann das Steuerelement überprüfen, ob das Fenster bereits erzeugt wurde. Wenn das Handle ungültig ist, erzeugt das Steuerelement das Fenster und gibt das Handle zurück. Dadurch ist sichergestellt, daß eine Anwendung immer ein gültiges Handle abrufen kann, indem sie auf die Eigenschaft *Handle* zugreift.

Delphi-Komponenten nehmen dem Entwickler Aufgaben wie das Erzeugen von Fenstern ab und erlauben ihm dadurch, sich auf wichtigere Dinge zu konzentrieren. Der Entwickler braucht weder ein Fenster zu erzeugen noch zu überprüfen, ob ein Handle existiert, bevor er es einer API-Funktion übergibt. Er kann davon ausgehen, daß sich die Anwendung korrekt verhält und muß sich nicht ständig den Kopf über mögliche Fehlerquellen zerbrechen.

Natürlich dauert es länger, Komponenten zu entwickeln, die frei von Abhängigkeiten sind. Die zusätzliche Zeit ist aber gut angelegt. Ausgereifte Komponenten ersparen nicht nur den Anwendungsprogrammierern alle möglichen Unannehmlichkeiten, sondern verringern auch Ihren eigenen Aufwand für die Dokumentation und die technische Unterstützung.

Eigenschaften, Ereignisse und Methoden

Abgesehen vom sichtbaren Abbild, das im Formulardesigner manipuliert wird, sind die augenfälligsten Attribute einer Komponente ihre Eigenschaften, Ereignisse und Methoden. Jedem der drei Themen ist in diesem Buch ein eigenes Kapitel gewidmet. Die folgenden Abschnitte enthalten dazu einige Anregungen.

Eigenschaften

Eigenschaften stellen sich dem Anwendungsprogrammierer so dar, als würde er auf eine Variable zugreifen. Der Komponentenentwickler hat die Möglichkeit, die zu-

grundeliegende Datenstruktur zu verbergen oder spezielle Verarbeitungsschritte in die Wege zu leiten, wenn auf den Wert der Eigenschaft zugegriffen wird.

Eigenschaften bieten verschiedene Vorteile:

- Sie sind zur Entwurfszeit verfügbar. Der Anwendungsprogrammierer kann den ursprünglichen Wert einer Eigenschaft ändern, ohne Quelltext eingeben zu müssen.
- Sie können bereits bei der Zuweisung Werte oder Formate überprüfen. Diese Eingabeprüfung zur Entwurfszeit hilft Fehler vermeiden.
- Die Komponente selbst kann bei Bedarf geeignete Werte erzeugen. Einer der häufigsten Programmierfehler ist der Zugriff auf eine Variable, die noch nicht initialisiert wurde. Indem Sie Daten als Eigenschaften darstellen, können Sie dafür sorgen, daß sie bei Bedarf immer einen Wert enthalten.
- Eigenschaften verbergen Daten hinter einer einfachen, konsistenten Schnittstelle. Die zugrundeliegende Struktur der Daten kann geändert werden, ohne daß der Anwendungsprogrammierer dies bemerkt.

Kapitel 33, »Eigenschaften erstellen«, erläutert, wie Sie Ihren Komponenten Eigenschaften hinzufügen.

Ereignisse

Ein Ereignis ist eine spezielle Eigenschaft, deren Wert sich zur Laufzeit als Folge einer Benutzereingabe ändert. Ereignisse geben dem Anwendungsprogrammierer die Möglichkeit, die Ausführung bestimmter Quelltextblöcke beispielsweise von Mausklicks oder Tastendrücken abhängig zu machen. Den Quelltext, der ausgeführt wird, wenn ein Ereignis eintritt, nennt man eine *Ereignisbehandlungsroutine*.

Mit Hilfe von Ereignissen kann der Anwendungsprogrammierer auf verschiedene Arten von Eingaben reagieren, ohne neue Komponenten zu definieren.

Kapitel 34, »Ereignisse erzeugen«, erläutert, wie Standardereignisse implementiert und neue Ereignisse definiert werden.

Methoden

Klassenmethoden sind Prozeduren und Funktionen, die sich auf die gesamte Klasse und nicht nur auf bestimmte Klasseninstanzen beziehen. Die Konstruktor-Methode (*Create*) einer jeden Komponente ist beispielsweise eine Klassenmethode. Komponentenmethoden sind Prozeduren und Funktionen, die sich auf die Komponenten selbst beziehen. Anwendungsprogrammierer können Komponenten mit Hilfe von Methoden dazu veranlassen, bestimmte Aktionen durchzuführen oder einen Wert zurückzugeben, der nicht in Form einer Eigenschaft verfügbar ist.

Da sie die Ausführung von Quelltext erfordern, können Methoden nur zur Laufzeit aufgerufen werden. Methoden sind aus verschiedenen Gründen nützlich:

- Sie kapseln die Funktionalität einer Komponente in dem Objekt, das auch die Daten enthält.

- Sie verbergen komplizierte Arbeitsschritte hinter einer einfachen, konsistenten Schnittstelle. Ein Anwendungsprogrammierer kann beispielsweise die Methode *AlignControls* einer Komponente aufrufen, ohne ihre Arbeitsweise oder die Unterschiede zu einer gleichnamigen Methode anderer Komponenten zu kennen.
- Sie erlauben die Aktualisierung mehrerer Eigenschaften in einem einzigen Aufruf.

In Kapitel 35, »Methoden erzeugen«, erfahren Sie, wie Sie Ihren Komponenten Methoden hinzufügen.

Grafik kapseln

Delphi vereinfacht den Umgang mit den grafischen Funktionen von Windows, indem die verschiedenen Funktionen in einer Zeichenfläche gekapselt werden. Die Zeichenfläche stellt die Zeichenoberfläche eines Fensters oder Steuerelements dar und enthält ihrerseits Klassen, wie z. B. *TPen*, *TBrush* und *TFont*. Eine Zeichenfläche entspricht etwa einem Windows-Gerätekontext, nimmt Ihnen aber alle Verwaltungsaufgaben ab.

Wenn Sie schon einmal eine grafische Windows-Anwendung geschrieben haben, kennen Sie die Anforderungen, die die grafische Geräteschnittstelle von Windows (GDI) an den Programmierer stellt. So haben Sie bei der GDI beispielsweise nur eine begrenzte Anzahl von Gerätekontexten zur Verfügung und müssen grafische Objekte in den Ausgangszustand zurückversetzen, bevor Sie sie freigeben.

In Delphi brauchen Sie sich um solche Dinge nicht zu kümmern. Um auf einem Formular oder einer anderen Komponente zu zeichnen, greifen Sie einfach auf die Eigenschaft *Canvas* zu. Wenn Sie einen bestimmten Stift oder Pinsel benötigen, setzen Sie die Eigenschaften *Color* oder *Style*. Am Ende gibt Delphi die Ressourcen für Sie frei. Außerdem hält Delphi häufig benötigte Ressourcen in einem Zwischenspeicher, damit sie von der Anwendung nicht ständig neu erzeugt werden müssen.

Sie haben nach wie vor vollen Zugriff auf die Windows-GDI, werden aber feststellen, daß Ihr Code kompakter erscheint und schneller läuft, wenn Sie sich der integrierten Zeichenfläche der Delphi-Komponenten bedienen. Details über grafische Funktionen finden Sie in Kapitel 36, »Grafiken in Komponenten«.

Registrierung

Bevor Sie Ihre Komponenten in der Entwicklungsumgebung von Delphi installieren können, müssen Sie sie registrieren. Durch die Registrierung teilen Sie Delphi mit, an welcher Position der Palette eine Komponente angezeigt werden soll. Außerdem können Sie festlegen, auf welche Weise Delphi Ihre Komponenten in der Formulardatei speichert. Details über die Registrierung finden Sie in Kapitel 38, »Komponenten zur Entwurfszeit verfügbar machen«.

Eine neue Komponente erzeugen

Es gibt zwei verschiedene Möglichkeiten, eine neue Komponente zu erzeugen:

- Der Komponentenexperte
- Eine Komponente manuell erzeugen

Mit jedem dieser Verfahren können Sie Komponenten erzeugen, die mit einem minimalen Funktionsumfang ausgestattet sind und in der Komponentenpalette installiert werden können. Danach können Sie die neue Komponente in ein Formular einfügen und ihr Verhalten zur Entwurfszeit und zur Laufzeit testen. Anschließend fügen Sie der Komponente weitere Funktionen hinzu, aktualisieren die Komponentenpalette und führen einen erneuten Test durch.

Einige grundlegende Schritte müssen immer durchgeführt werden, wenn eine neue Komponente erzeugt wird. Diese Schritte werden im folgenden beschrieben. Andere Beispiele in diesem Handbuch gehen davon aus, daß Ihnen dieses Vorgehen bekannt ist. Das Erzeugen einer Komponente erfolgt in folgenden Schritten:

1. Sie erzeugen eine Unit für die neue Komponente.
2. Sie leiten die Komponente von einem vorhandenen Komponententyp ab.
3. Sie fügen der Komponente Eigenschaften, Methoden und Ereignisse hinzu.
4. Sie registrieren die Komponente in Delphi.
5. Sie erzeugen eine Hilfedatei für die Komponente und ihre Eigenschaften, Methoden und Ereignisse.
6. Sie erzeugen ein Package (eine spezielle Art von DLL), mit dem Sie die Komponente in der Entwicklungsumgebung von Delphi installieren können.

Zu einer vollständigen Komponente gehören am Ende folgende Dateien:

- Ein Package (BPL) oder eine Package Collection (DPC)
- Eine compilierte Package-Datei (DCP)
- Eine compilierte Unit (DCU)
- Eine Paletten-Bitmap (DCR)
- Eine Hilfedatei (HLP)

Die Erzeugung einer Hilfedatei (die dem Anwender die Verwendung der Komponente erklärt) ist optional. Das Beifügen einer Hilfedatei ist nur dann obligatorisch, wenn eine Hilfedatei erzeugt wurde.

Die restlichen Kapitel von Teil IV erläutern alle Aspekte der Komponentenentwicklung und enthalten mehrere vollständige Beispiele für verschiedene Komponentenarten.

Der Komponentenexperte

Der Komponentenexperte vereinfacht die einleitenden Schritte der Komponentenentwicklung. Sie brauchen nur folgende Informationen bereitzustellen:

- Die Klasse, von der die neue Komponente abgeleitet wird.
- Der Klassenname der neuen Komponente
- Die Seite der Komponentenpalette, in der die neue Komponente angezeigt werden soll.
- Der Name der Unit, in der die Komponente erzeugt wird.
- Der Verzeichnispfad, in welchem die Unit abgelegt ist.
- Der Name des Package, das die neue Komponente enthalten soll.

Der Komponentenexperte führt dieselben Aufgaben durch wie Sie selbst, wenn Sie eine Komponente manuell erzeugen:

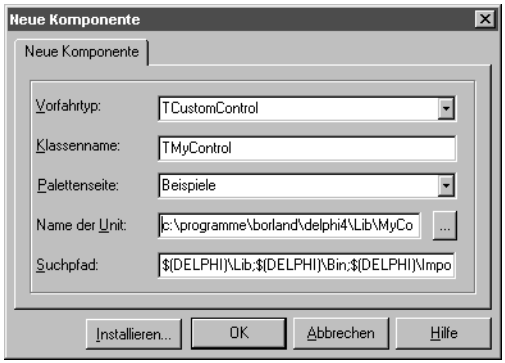
- Er erzeugt eine Unit.
- Er leitet die Komponente ab.
- Er registriert die Komponente.

Der Komponentenexperte kann einer vorhandenen Unit keine Komponenten hinzufügen. Diese Aufgabe müssen Sie manuell durchführen.

Sie können den Komponentenexperten auf zwei Arten aufrufen:

- Wählen Sie *Komponente / Neue Komponente.*
- Wählen Sie *Datei / Neu*, und doppelklicken Sie auf *Komponente.*

Abbildung 31.2 Der Komponentenexperte



Der Komponentenexperte benötigt von Ihnen folgende Informationen:

1 Im Feld *Vorfahrtyp* geben Sie die Klasse an, von der Sie die neue Komponente ableiten.

2 Im Feld *Klassenname* geben Sie den Klassennamen der neuen Komponente an.

3. Im Feld *Palettenseite* geben Sie an, auf welcher Registerkarte der Komponentenpalette die neue Komponente installiert werden soll.
4. Im Feld *Unit-Dateiname* geben Sie den Namen der Unit an, in der die neue Komponentenklasse deklariert werden soll.
5. Wenn sich die Unit nicht im Suchpfad befindet, bearbeiten Sie den Eintrag im Feld *Suchpfad*.

Um die Komponente einem neuen oder bereits vorhandenen Package hinzuzufügen, klicken Sie auf *Komponente / Installieren*, geben im gleichnamigen Dialogfeld ein Package an und bestätigen mit *OK*.

Warnung Wenn Sie eine Komponente von einer Klasse der VCL ableiten, deren Name mit »TCustom« beginnt (z. B. *TCustomControl*), dürfen Sie die neue Komponente nicht in einem Formular plazieren, bevor Sie nicht alle abstrakten Methoden der Originalkomponente überschrieben haben. Delphi kann keine Instanzen einer Klasse erzeugen, die abstrakte Methoden oder Eigenschaften enthält.

Um sich den Quelltext der Unit anzeigen zu lassen, klicken Sie auf *Unit anzeigen* (wenn der Komponentenexperte bereits geschlossen ist, öffnen Sie die Datei im Quelltext-Editor mit *Datei / Öffnen*). Delphi erzeugt eine neue Unit mit der Klassendeklaration und der Prozedur *Register* und fügt ihrem **uses**-Abschnitt alle Standard-Units von Delphi hinzu. Die Unit sieht etwa folgendermaßen aus:

```
unit MyControl;
interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
   TMyControl = class(TCustomControl)
   private
   { Private-Deklarationen }
   protected
   { Protected-Deklarationen }
   public
   { Public-Deklarationen }
   published
   { Published-Deklarationen }
end;

procedure Register;

implementation

procedure Register;
begin
   RegisterComponents('Samples', [TMyControl]);
end;
end.
```

Eine Komponente manuell erzeugen

Die einfachste Art, eine neue Komponente zu erzeugen, ist der Komponentenexperte. Sie können dieselben Schritte aber auch manuell durchführen.

Gehen Sie dazu folgendermaßen vor:

1. Erzeugen Sie eine Unit.
2. Leiten Sie die Komponente ab.
3. Registrieren Sie die Komponente.

Eine Unit erzeugen

Eine Unit ist ein separat compiliertes Modul mit Object Pascal-Code. Delphi verwendet Units für verschiedene Zwecke. Jedes Formular und die meisten Komponenten (oder Gruppen verwandter Komponenten) haben ihre eigenen Units.

Für eine neue Komponente müssen Sie entweder eine neue Unit erzeugen oder die Komponente einer vorhandenen Unit hinzufügen.

Um eine Unit zu erzeugen, wählen Sie *Datei* / *Neu* und klicken doppelt auf *Unit*. Delphi erzeugt eine neue Unit und öffnet sie im Quelltext-Editor.

Um eine vorhandene Unit zu öffnen, wählen Sie *Datei* / *Öffnen* und geben die Quelltext-Unit an, der Sie Ihre Komponente hinzufügen wollen.

Hinweis Eine Unit, der Sie eine Komponente hinzufügen, darf nur Komponenten-Code enthalten. Wenn Sie einer Unit, die beispielsweise ein Formular enthält, eine Komponente hinzufügen, verursachen Sie einen Fehler in der Komponentenpalette.

Sobald Sie eine neue oder bestehende Unit für die Komponente festgelegt haben, können Sie die Komponentenklasse ableiten.

Die Komponente ableiten

Jede Komponente ist eine Klasse, die von *TComponent* oder einem davon abgeleiteten, mehr spezialisierten Nachkommen (wie z. B. *TControl* oder *TGraphicControl*), abgeleitet wurde. Eine Komponente kann auch von einer bestehenden Komponentenklasse abgeleitet worden sein. »Wie werden Komponenten erzeugt?« auf Seite 31-3 beschreibt, von welchen Klassen die verschiedenen Komponentenarten abgeleitet werden sollten.

Das Ableiten von Klassen wird im Abschnitt »Neue Klassen definieren« auf Seite 32-2 beschrieben.

Um eine Komponente abzuleiten, fügen Sie dem **interface**-Abschnitt der Unit, die diese Komponente enthalten soll, eine entsprechende Objekttyp-Deklaration hinzu.

Eine einfache Komponentenklasse ist eine nichtvisuelle Komponente, die direkt von *TComponent* abgeleitet wurde.

Beispiel für die Ableitung einer Komponente

Um eine einfache Komponentenklasse zu erzeugen, fügen Sie die nachfolgende Klassendeklaration zum **interface**-Abschnitt Ihrer Komponenten-Unit hinzu:

```
type
   TNewComponent = class(TComponent)
   end;
```

Die neu erzeugte Komponente verfügt bis jetzt über exakt die gleiche Funktionalität wie *TComponent*. Sie haben hier schlicht das Grundgerüst erzeugt, auf dessen Basis Sie Ihre neue Komponente erweitern können.

Die Komponente registrieren

Das Registrieren ist ein einfacher Vorgang, bei dem Delphi mitgeteilt wird, welche Komponenten der Komponentenbibliothek hinzugefügt werden sollen und auf welcher Seite der Komponentenpalette sie angezeigt werden sollen. Details zu diesem Thema finden Sie in Kapitel 38, »Komponenten zur Entwurfszeit verfügbar machen«.

So registrieren Sie eine Komponente:

1. Fügen Sie dem **interface**-Abschnitt der Komponenten-Unit eine Prozedur namens *Register* hinzu. Da dieser Prozedur keine Parameter übergeben werden, ist die Deklaration ganz einfach:

   ```
   procedure Register;
   ```

2. Wenn die Unit bereits Komponenten enthält, sollte auch die Prozedur *Register* bereits deklariert sein. Sie brauchen an der Deklaration nichts zu ändern.

3. Geben Sie den Quelltext für die Prozedur *Register* im **implementation**-Abschnitt der Unit ein, indem Sie für jede Komponente, die registriert werden soll, die Prozedur *RegisterComponents* aufrufen. *RegisterComponents* erwartet zwei Parameter: den Namen einer Seite der Komponentenpalette und ein Set mit Komponententypen. Wenn Sie in eine bestehende Registrierung eine neue Komponente aufnehmen wollen, können Sie entweder die Komponente dem Set hinzufügen oder in einer eigenen Anweisung erneut *RegisterComponents* aufrufen.

Beispiel für die Registrierung einer Komponente

Um beispielsweise eine Komponente namens *TMyControl* zu registrieren und sie in die Seite *Beispiele* der Komponentenpalette aufzunehmen, fügen Sie der Unit, die die Deklaration von *TMyControl* enthält, die folgende Prozedur *Register* hinzu:

```
procedure Register;
begin
   RegisterComponents('Beispiele', [TNewControl]);
end;
```

Diese *Register*-Prozedur fügt *TMyControl* der Seite *Beispiele* der Komponentenpalette hinzu.

Sobald eine Komponente registriert ist, kann Sie in ein Package (siehe Kapitel 20, »Tabellen«) compiliert und in der Komponentenpalette installiert werden.

Komponenten vor der Installation testen

Sie können das Laufzeitverhalten einer Komponente testen, bevor Sie sie in der Komponentenpalette installieren. Dies empfiehlt sich vor allem, wenn Sie neue Komponenten mit dem Debugger untersuchen. Das hier beschriebene Vorgehen läßt sich auch auf Komponenten anwenden, die nicht in der Komponentenpalette angezeigt werden.

Um nicht installierte Komponenten zu testen, müssen Sie die Aktionen emulieren, die Delphi durchführt, wenn die Komponente in der Palette markiert und in einem Formular plaziert wird.

So testen Sie eine nicht installierte Komponente:

1. Fügen Sie die Unit der Komponente dem **uses**-Abschnitt der Formular-Unit hinzu.
2. Fügen Sie dem Formular ein Objektfeld für die Komponente hinzu.

   An diesem Punkt ergibt sich ein wichtiger Unterschied zwischen Ihrem Vorgehen und dem Vorgehen von Delphi. Sie fügen das Objektfeld dem **public**-Abschnitt im unteren Bereich der Typdeklaration des Formulars hinzu. Delphi würde es weiter oben in dem von ihm verwalteten Bereich einfügen.

   Sie sollten dem Teil der Typdeklaration, der von Delphi verwaltet wird, niemals Felder hinzufügen. Für alle Objekte in diesem Teil der Deklaration gibt es eine Entsprechung in der Formulardatei. Wenn Sie hier Komponenten einfügen, die im Formular nicht existieren, erzeugen Sie möglicherweise eine ungültige Formulardatei.
3. Weisen Sie dem Ereignis *OnCreate* des Formulars eine Behandlungsroutine zu.
4. Erzeugen Sie in dieser Ereignisbehandlungsroutine die Komponente.

   Wenn Sie den Konstruktor der Komponente aufrufen, müssen Sie einen Parameter übergeben, der den Eigentümer der Komponente angibt (also das Objekt, das die Komponente zum geeigneten Zeitpunkt freigibt). Sie werden fast immer *Self* als Eigentümer übergeben. Innerhalb einer Methode stellt *Self* immer eine Referenz auf das Objekt dar, das die Methode enthält. In der Behandlungsroutine für Ereignis *OnCreate* des Formulars bezieht sich *Self* also auf das Formular.
5. Weisen Sie der Eigenschaft *Parent* einen Wert zu.

   Nachdem Sie ein Steuerelement erzeugt haben, besteht der erste Schritt immer im Zuweisen eines Wertes an die Eigenschaft *Parent*. Diese Eigenschaft zeigt auf die Komponente, in der das Steuerelement optisch enthalten ist. Normalerweise ist dies das Formular; es kann sich aber auch um ein Gruppenfeld oder um eine Tafel handeln. Gewöhnlich weisen Sie *Parent* den Wert *Self* zu, also das Formular. *Parent* sollte immer die erste Eigenschaft eines neuen Steuerelements sein, der Sie einen Wert zuweisen.

Warnung Wenn Ihre Komponente kein Steuerelement ist (d.h. wenn sie nicht direkt oder indirekt von *TControl* abstammt), überspringen Sie diesen Schritt. Sollten Sie versehentlich die Eigenschaft *Parent* des Formulars (nicht der Komponente) auf *Self* setzen, verursachen Sie möglicherweise ein Problem auf Betriebssystemebene.

6 Weisen Sie den anderen Eigenschaften der Komponente die gewünschten Werte zu.

Beispiel für das Testen von noch nicht installierten Komponenten

Nehmen wir an, Sie wollen eine Komponente vom Typ *TMyControl* testen, die in einer Unit namens *MyControl* deklariert ist. Erzeugen Sie ein neues Projekt, und führen Sie die beschriebenen Schritte durch. Am Ende sollten Sie eine Formular-Unit erhalten, die etwa so aussieht:

```
unit Unit1;
interface

uses
   SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
   Forms, Dialogs, MyControl; { 1. NewTest der uses-Klausel hinzufügen }

type
   TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject); { 3. Behandlungsroutine für OnCreate zuweisen }

   private
    { Private declarations }
   public
    { Public Declarations }
    MyControl1: TMyControl1; { 2. Objektfeld einfügen }
   end;

var
   Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
   MyControl1 := TMyControl.Create(Self); { 4. Komponente erzeugen }
   MyControl1.Parent := Self; { 5. Eigenschaft Parent setzen, falls Komponente ein
                                Steuerelement ist}
   MyControl1.Left := 12; { 6. Weitere Eigenschaftswerte zuweisen ... )
   % { ..wird nach Bedarf fortgesetzt }
end;
end.
```

Kapitel

32

Objektorientierte Programmierung für Komponentenentwickler

Falls Sie bereits Anwendungen in Delphi geschrieben haben, wissen Sie, daß eine Klasse sowohl Daten als auch Quelltext enthält und daß Sie Klassen zur Entwurfs- und zur Laufzeit bearbeiten können. In diesem Zusammenhang sind Sie ein Komponentenbenutzer.

Wenn Sie dagegen neue Komponenten entwickeln, gehen Sie mit Klassen anders um als Anwendungsentwickler. Sie versuchen, die internen Vorgänge der Komponente vor dem Entwickler, der sie benutzt, zu verbergen. Durch die Auswahl geeigneter Vorfahren für die Komponenten und mit Hilfe von Schnittstellen, die nur die vom Entwickler benötigten Eigenschaften und Methoden zur Verfügung stellen, können Sie unter Berücksichtigung der in diesem Kapitel angesprochenen Richtlinien vielseitige und wiederverwendbare Komponenten entwickeln.

Wenn Sie Komponenten erzeugen wollen, sollten Ihnen die folgenden Themen vertraut sein, die mit der objektorientierten Programmierung (OOP) zusammenhängen:

- Neue Klassen definieren
- Vorfahren, Nachkommen und Klassenhierarchien
- Zugriffssteuerung
- Verteilen von Methoden
- Abstrakte Klassenelemente
- Klassen und Zeiger

Neue Klassen definieren

Der Unterschied zwischen Komponenten- und Anwendungsentwicklern besteht darin, daß die ersteren neue Klassen erzeugen, während letztere Instanzen von Klassen bearbeiten.

Eine Klasse ist im Grunde genommen ein Typ. Als Programmierer arbeiten Sie immer mit Typen und Instanzen (auch dann, wenn Sie diese Begriffe nicht benutzen). Sie deklarieren beispielsweise Variablen eines bestimmten Typs, z. B. vom Typ *Integer*. Klassen sind normalerweise komplexer als Datentypen, funktionieren aber auf dieselbe Weise. Durch Zuweisen unterschiedlicher Werte an Instanzen desselben Typs können Sie unterschiedliche Aufgaben ausführen.

So kommt es beispielsweise relativ häufig vor, daß Sie Formulare mit zwei Schaltflächen erstellen, die die Beschriftung *OK* und *Abbrechen* tragen. Beide Schaltflächen sind eine Instanz der Klasse *TButton*. Durch die Zuweisung unterschiedlicher Werte an ihre *Caption*-Eigenschaften und unterschiedlicher Behandlungsroutinen an ihre *OnClick*-Ereignisse erhalten Sie zwei Instanzen, die ein unterschiedliches Verhalten zeigen.

Neue Klassen ableiten

Für die Ableitung einer neuen Klasse gibt es zwei Gründe:

- Ändern der Voreinstellungen, um Wiederholungen zu vermeiden.
- Hinzufügen neuer Möglichkeiten zu der Klasse.

In beiden Fällen ist das Ergebnis ein wiederverwendbares Objekt. Wenn Sie bei neuen Komponenten auf Wiederverwendbarkeit achten, können Sie sich später viel Arbeit sparen. Versehen Sie Ihre Klassen mit sinnvollen Standardwerten, aber lassen Sie Änderungen zu.

Ändern der Voreinstellungen einer Klasse zur Vermeidung von Wiederholung

Die meisten Programmierer versuchen, Wiederholungen zu vermeiden. Wenn Sie also bemerken, daß Sie dieselben Quelltextzeilen immer wieder schreiben, fügen Sie den Quelltext in eine Subroutine oder Funktion ein oder erstellen eine Bibliothek mit Routinen, die sich in vielen Programmen verwenden lassen. Nichts anderes gilt auch für Komponenten. Wenn Sie immer wieder dieselben Eigenschaften ändern oder dieselben Methodenaufrufe durchführen, können Sie eine neue Komponente erstellen, die diese Aufgaben per Voreinstellung durchführt.

Angenommen, Sie fügen bei der Erstellung einer Anwendung jedesmal ein Dialogfeld ein, mit dessen Hilfe eine bestimmte Operation ausgeführt wird. Das wiederholte Anlegen des Dialogfeldes ist zwar nicht schwierig, aber auch nicht notwendig. Sie können das Dialogfeld einmal entwerfen, seine Eigenschaften festlegen und eine Komponente, die es kapselt, in der Komponentenpalette installieren. Dadurch, daß Sie das Dialogfeld als wiederverwendbare Komponente definieren, vermeiden Sie nicht nur die unnötige Wiederholung von Arbeitsschritten, sondern erhöhen auch die Standardisierung und schalten Fehlerquellen aus.

Kapitel 39, »Vorhandene Komponenten modifizieren«, enthält ein Beispiel für die Änderung der Standardeigenschaften einer Komponente.

Hinweis Wenn Sie nur die als **published** deklarierten Eigenschaften einer vorhandenen Komponente ändern oder nur bestimmte Ereignisbehandlungsroutinen für eine Komponente oder eine Gruppe von Komponenten speichern wollen, läßt sich dies einfacher mit Hilfe einer Komponentenschablone erledigen.

Einer Klasse neue Fähigkeiten hinzufügen

Häufig werden Komponenten erzeugt, weil neue, bisher in keiner Komponente vorhandene Fähigkeiten benötigt werden. Leiten Sie in diesem Fall die neue Komponente entweder von einer bestehenden Komponente oder von einer abstrakten Basisklasse ab, z. B. von *TComponent* oder *TControl*.

Leiten Sie die neue Komponente von der Klasse ab, die Ihren Vorstellungen am nächsten kommt. Sie können einer Klasse zwar Fähigkeiten hinzufügen, aber keine aus ihr entfernen. Enthält eine vorhandene Klasse unerwünschte Eigenschaften, müssen Sie für die Ableitung deshalb den Vorfahr der Klasse verwenden.

Wenn Sie beispielsweise einem Listenfeld neue Funktionen zuordnen wollen, können Sie eine Komponente von *TListBox* ableiten. Wenn Sie aber einige Leistungsmerkmale von Listenfeldern ausschließen möchten, müssen Sie die neue Komponente von *TCustomListBox*, dem Vorfahr von *TListBox* ableiten. Anschließend erzeugen Sie nur die gewünschten Listenfeldmerkmale (bzw. machen sie sichtbar) und führen neue Merkmale ein.

In Kapitel 41, »Gitter anpassen«, finden Sie ein Beispiel für die Anpassung einer abstrakten Komponentenklasse.

Eine neue Komponentenklasse deklarieren

Zusätzlich zu den Standardkomponenten stellt Delphi viele abstrakte Klassen bereit, die als Basis für die Ableitung neuer Komponenten dienen. In Tabelle 32.1, »Sichtbarkeitsstufen in einem Objekt« auf Seite 32-5 sind die Klassen aufgeführt, die Sie als Grundlage für neue Komponenten verwenden können.

Bei der Deklaration einer neuen Komponentenklasse fügen Sie eine Klassendeklaration in die Unit-Datei der Komponente ein.

Beispiel für die Deklaration einer neuen Klasse

So sieht die Deklaration einer einfachen grafischen Komponente aus:

```
type
   TSampleShape = class(TGraphicControl)
   end;
```

Eine vollständige Komponentendeklaration enthält normalerweise Eigenschafts-, Ereignis- und Methodendeklarationen vor dem reservierten Wort **end**. Auch die obige Deklaration ist aber zulässig und stellt einen Ausgangspunkt für eine neue Komponente dar.

Vorfahren, Nachkommen und Klassenhierarchien

Anwendungsentwickler setzen voraus, daß jedes Steuerelement über die Eigenschaften *Top* und *Left* verfügt. Diese Eigenschaften bestimmen die Position des Elements im Formular. Für den Anwendungsentwickler spielt es keine Rolle, ob alle Steuerelemente diese Eigenschaften von ihrem gemeinsamen Vorfahr *TControl* erben. Wenn Sie jedoch eine Komponente erzeugen, müssen Sie wissen, von welcher Klasse das Element abgeleitet werden muß, damit es die gewünschten Merkmale erbt. Außerdem müssen Sie genau wissen, was das Steuerelement erbt, damit Sie geerbte Fähigkeiten nicht neu einführen.

Die Klasse, von der Sie eine Komponente ableiten, wird als unmittelbarer Vorfahr bezeichnet. Jede Komponente erbt von ihrem unmittelbaren Vorfahr und vom unmittelbaren Vorfahr ihres unmittelbaren Vorfahren usw. Alle Klassen, von der eine Komponente erbt, sind die Vorfahren dieser Komponente, und die Komponente selbst ist ein Nachkomme ihrer Vorfahren.

Alle Beziehungen zwischen Vorfahren und Nachkommen in einer Anwendung bilden eine Klassenhierarchie. Jede Generation in der Hierarchie enthält mehr Merkmale als ihre Vorfahren, weil sie alle Merkmale von ihren Vorfahren erbt und dazu neue Eigenschaften und Methoden einführt oder vorhandene neu definiert.

Wenn Sie keinen unmittelbaren Vorfahr angeben, leitet Delphi Ihre Komponente vom Standardvorfahr *TObject* ab. *TObject* ist der Vorfahr aller Klassen der VCL (Visual Component Library).

Die Grundregel bei der Auswahl des Objekts, von dem abgeleitet werden soll, ist einfach: Wählen Sie jenes Objekt, das soviel wie möglich von dem enthält, was Sie in Ihr neues Objekt einbinden möchten, und das nichts davon enthält, was Sie in Ihrem neuen Objekt nicht haben möchten. Sie können Ihrem Objekt immer neue Merkmale hinzufügen. Das Entfernen von Merkmalen ist jedoch nicht möglich.

Zugriffssteuerung

Es gibt fünf Stufen der Zugriffssteuerung – die sogenannte Sichtbarkeit – für Eigenschaften, Methoden und Felder. Die Sichtbarkeit bestimmt, welcher Quelltext auf welche Abschnitte der Klasse zugreifen kann. Über die Sichtbarkeit definieren Sie die Schnittstelle zu Ihren Komponenten.

Tabelle 32.1, »Sichtbarkeitsstufen in einem Objekt« enthält die Sichtbarkeitsstufen, geordnet nach dem Grad der Restriktion, die sie bewirken:

Tabelle 32.1 Sichtbarkeitsstufen in einem Objekt

Sichtbarkeit	Bedeutung	Verwendung
private	Zugriff nur für den Quelltext, der sich in derselben Unit befindet, in der auch die Klasse definiert ist.	Verbirgt die Details der Implementierung.
protected	Zugriff nur für den Quelltext, der sich in der Unit (bzw. in den Units) befindet, in der auch die Klasse und ihre Nachkommen definiert sind.	Definiert die Schnittstelle des Komponentenentwicklers.
public	Zugriff für jeden Quelltext.	Definiert die Laufzeit-Schnittstelle.
automated	Zugriffsmöglichkeit für jeden Quelltext. Auf dieser Stufe werden Automatisierungs-Typinformationen erzeugt.	Nur für OLE-Automatisierung.
published	Zugriff für jeden Quelltext. Außerdem ist der Zugriff vom Objektinspektor aus möglich.	Definiert die Entwurfszeit-Schnittstelle.

Deklarieren Sie Elemente als **private**, wenn sie nur in der Klasse verfügbar sein sollen, in der sie auch definiert werden. Deklarieren Sie Elemente als **protected**, wenn sie nur in dieser Klasse und in ihren Nachkommen verfügbar sein sollen. Bedenken Sie dabei aber, daß ein Element, das an irgendeiner Position innerhalb einer Unit-Datei verfügbar ist, in der gesamten Datei zur Verfügung steht. Wenn Sie also zwei Klassen in derselben Unit definieren, können diese Klassen auf die als **private** deklarierten Methoden der jeweils anderen Klasse zugreifen. Wenn Sie eine Klasse in einer anderen Unit als der ihres Vorfahren ableiten, sind alle Klassen in der neuen Unit in der Lage, auf die als **protected** deklarierten Methoden des Vorfahren zuzugreifen.

Implementierungsdetails verbergen

Ein als **private** deklarierter Abschnitt einer Klasse ist für Quelltext, der sich außerhalb der Unit-Datei der Klasse befindet, unsichtbar. Innerhalb der Unit, die die Deklaration enthält, kann jeder Quelltext auf diesen Abschnitt zugreifen, als wäre er als **public** deklariert.

Im folgenden sehen Sie ein Beispiel für ein als **private** deklariertes Feld, das für Anwendungsentwickler unsichtbar ist. Das Listing enthält zwei Units. Jede Unit besitzt eine Behandlungsroutine für ihr Ereignis *OnCreate*, das einem als **private** deklarierten Feld einen Wert zuweist. Der Compiler erlaubt Zuweisungen an das Feld nur in dem Formular, in dem es deklariert ist.

```
unit HideInfo;
interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;
type
   TSecretForm = class(TForm) { Neues Formular deklarieren  }
```

```
    procedure FormCreate(Sender: TObject);
  private { private-Abschnitt deklarieren }
    FSecretCode: Integer; { Ein Feld als private deklarieren }
  end;

var
  SecretForm: TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42; { Wird fehlerfrei compiliert }
end;
end. { Ende der Unit }
```

```
unit TestHide; { Hauptformulardatei }
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo; { Unit mit TSecretForm verwenden }

type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;   { Compiler stoppt mit der Meldung "Feldbezeichner
                                    erwartet"  }
end;
end. { Ende der Unit }
```

Obwohl ein Programm, das die Unit *HideInfo* einbindet, Objekte vom Typ *TSecretForm* verwenden kann, ist ihm der Zugriff auf das Feld *FSecretCode* in diesen Objekten verwehrt.

Die Schnittstelle des Komponentenentwicklers definieren

Ein als **protected** deklarierter Abschnitt einer Klasse macht diesen nur für die Klasse selbst und für ihre Nachkommen (und für andere Klassen, die dieselben Unit-Dateien verwenden) sichtbar.

Mit Hilfe von **protected**-Deklarationen können Sie für Komponentenentwickler eine Schnittstelle zu der Klasse definieren. Anwendungs-Units haben keinen Zugriff auf die als **protected** deklarierten Abschnitte, abgeleitete Klassen sehr wohl. Das bedeutet, daß Sie als Komponentenentwickler die Funktionsweise einer Klasse ändern können, ohne daß die Details für die Anwendungsentwickler sichtbar werden.

Die Laufzeit-Schnittstelle definieren

Ein als **public** deklarierter Abschnitt einer Klasse macht diesen nur für den Quelltext sichtbar, der Zugriff auf die gesamte Klasse hat.

Als **public** deklarierte Abschnitte stehen zur Laufzeit jedem Quelltext zur Verfügung, so daß diese Abschnitte die *Laufzeit-Schnittstelle* einer Klasse definieren. Die Laufzeit- Schnittstelle ist für Merkmale nützlich, die zur Entwurfszeit nicht sinnvoll oder nicht korrekt sind. Dazu gehören beispielsweise Eigenschaften, die von Benutzereingaben abhängen oder auf die nur Lesezugriffe möglich sind. Methoden, die von Anwendungsentwicklern aufgerufen werden können, müssen als **public** deklariert werden.

Das folgende Beispiel zeigt, wie zwei Nur-Lesen-Eigenschaften als Teil der Laufzeit-Schnittstelle einer Komponente deklariert werden:

```
type
   TSampleComponent = class(TComponent)
   private
    FTempCelsius: Integer; { Implementierungsdetails sind private }
    function GetTempFahrenheit: Integer;
   public
    property TempCelsius: Integer read FTempCelsius;   { Eigenschaften werden als public
                                                              deklariert  }
    property TempFahrenheit: Integer read GetTempFahrenheit;
   end;
⋮
function TSampleComponent.GetTempFahrenheit: Integer;
begin
   Result := FTempCelsius * 9 div 5 + 32;
end;
```

Die Entwurfszeit-Schnittstelle definieren

Ein als **public** deklarierter Abschnitt einer Klasse macht diesen öffentlich und generiert Laufzeit-Typinformationen. Diese Typinformationen ermöglichen z. B. dem Objektinspektor, auf Eigenschaften und Ereignisse zuzugreifen.

Da die als **published** deklarierten Abschnitte einer Klasse im Objektinspektor angezeigt werden, definieren sie die *Entwurfszeit-Schnittstelle* der Klasse. Die Schnittstelle sollte diejenigen Merkmale einer Klasse enthalten, die der Anwendungsentwickler unter Umständen zur Entwurfszeit anpassen möchte. Eigenschaften, die von speziellen Informationen über die Umgebung zur Laufzeit abhängig sind, gehören nicht hierher.

Nur-Lesen-Eigenschaften können nicht Teil der Entwurfszeit-Schnittstelle sein, weil der Anwendungsentwickler ihnen nicht direkt Werte zuweisen kann. Sie sollten deshalb als **public** und nicht als **published** deklariert werden.

Im folgenden sehen Sie ein Beispiel für eine als **published** deklarierte Eigenschaft namens *Temperatur*. Sie wird aufgrund ihrer Deklaration zur Entwurfszeit im Objektinspektor angezeigt.

```
type
   TSampleComponent = class(TComponent)
   private
    FTemperature: Integer; { Implementierungsdetails sind als private deklariert }
   published
    property Temperature: Integer read FTemperature write FTemperature; { mit
                                                                          Schreibzugriff! }
   end;
```

Dispatch-Methoden

Der Begriff *Dispatch* (verteilen) bezeichnet die Art und Weise, in der ein Programm festlegt, auf welche Implementierung einer Methode sich ein Methodenaufruf bezieht. Der Quelltext, der eine Methode aufruft, sieht wie jeder andere Prozedur- oder Funktionsaufruf aus. Klassen haben jedoch verschiedene Möglichkeiten zur Verteilung von Methoden.

Es gibt drei Arten der Verteilung von Methoden:

- Statisch
- Virtuell
- Dynamisch

Statische Methoden

Alle Methoden, bei deren Deklaration Sie keine Direktive angeben, sind statisch. Statische Methoden funktionieren wie normale Prozeduren oder Funktionen. Ein als **public** deklarierter Abschnitt einer Klasse macht Methoden nur für den Quelltext sichtbar, der Zugriff auf die gesamte Klasse hat. Der Compiler legt die exakte Adresse der Methoden fest und linkt sie beim Compilieren.

Der Hauptvorteil statischer Methoden besteht darin, daß sie sehr schnell verteilt werden können. Da der Compiler die genaue Adresse der Methode bestimmen kann, linkt er die Methode direkt. Virtuelle und dynamische Methoden verwenden im Gegensatz dazu indirekte Verfahren. Ihre Adressen werden erst zur Laufzeit ermittelt, was etwas mehr Zeit in Anspruch nimmt.

Eine statische Methode ändert sich nicht, wenn sie von einer abgeleiteten Klasse geerbt wird. Wenn Sie eine Klasse deklarieren, die eine statische Methode enthält, und danach eine neue Klasse ableiten, verwendet die abgeleitete Klasse genau dieselbe Methode an derselben Adresse. Folglich können Sie statische Methoden nicht überschreiben. Eine statische Methode führt immer dieselben Aktionen aus, unabhängig davon, in welcher Klasse sie aufgerufen wird. Wenn Sie in einer abgeleiteten Klasse eine Methode deklarieren, die denselben Namen wie eine statische Methode der Vorfahrklasse hat, ersetzt die neue Methode in der abgeleiteten Klasse einfach die geerbte Methode.

Beispiel für statische Methoden

Im folgenden Quelltextbeispiel deklariert die erste Komponente zwei statische Methoden. Die zweite Komponente deklariert zwei gleichnamige, ebenfalls statische Methoden. Diese Methoden ersetzen diejenigen, die von der ersten Komponente geerbt werden.

```
type
   TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
   end;

   TSecondComponent = class(TFirstComponent)
    procedure Move; { Trotz derselben Deklaration anders als die geerbte Methode }
    function Flash(HowOften: Integer): Integer; { Ebenfalls unterschiedlich }
   end;
```

Virtuelle Methoden

Virtuelle Methoden arbeiten mit einem komplizierteren und flexibleren Dispatch-Mechanismus als statische Methoden. Eine virtuelle Methode kann in der abgeleiteten Klasse neu definiert werden, während sie weiterhin in der Vorfahrklasse aufgerufen wird. Die Adresse einer virtuellen Methode wird nicht während der Compilierung festgelegt. Statt dessen sucht das Objekt, in dem die Methode definiert wird, die Adresse zur Laufzeit.

Um eine Methode als virtuell zu definieren, fügen Sie der Methodendeklaration die Direktive **virtual** hinzu. Diese Direktive erzeugt einen Eintrag in der *virtuellen Methodentabelle des Objekts* (VMT), die die Adressen der virtuellen Methoden eines Objekttyps enthält.

Wenn Sie eine neue Klasse von einer bestehenden ableiten, erhält die neue Klasse eine eigene VMT, die alle Einträge aus der VMT des Vorfahren und zusätzlich die Methoden enthält, die in der neuen Klasse deklariert wurden.

Methoden überschreiben

Anstatt eine Methode zu ersetzen, können Sie sie überschreiben und dabei erweitern oder neu definieren. Eine abgeleitete Klasse kann jede der geerbten virtuellen Methoden überschreiben.

Um eine Methode in einer abgeleiteten Klasse zu überschreiben, fügen Sie am Ende der Methodendeklaration die Direktive **override** hinzu.

Das Überschreiben einer Methode führt unter folgenden Bedingungen zu einem Compilierungsfehler:

- Die Methode existiert nicht in der Vorfahrklasse.
- Die Methode der Vorfahrklasse ist statisch.
- Die Deklarationen sind nicht identisch (Anzahl und Typen der Parameter sind unterschiedlich).

Der folgende Quelltextausschnitt deklariert zwei einfache Komponenten. In der ersten Komponente werden drei Methoden deklariert, jede mit einer anderen Verteilungsart. Die zweite Komponente, die von der ersten abgeleitet ist, ersetzt die statische Methode und überschreibt die virtuellen Methoden.

```
type
  TFirstComponent = class(TCustomControl)
   procedure Move;                { statische Methode }
   procedure Flash; virtual;      { virtuelle Methode }
   procedure Beep; dynamic;       { dynamische virtuelle Methode }
  end;

  TSecondComponent = class(TFirstComponent)
   procedure Move;                { deklariert eine neue Methode }
   procedure Flash; override;     { überschreibt die geerbte Methode }
   procedure Beep; override;      { überschreibt die geerbte Methode }
  end;
```

Dynamische Methoden

Dynamische Methoden sind virtuelle Methoden, die sich durch einen etwas anderen Dispatch-Mechanismus auszeichnen. Da dynamische Methoden keinen Eintrag in der VMT des Objekts haben, tragen sie zur Reduzierung des Speicherbedarfs bei. Das Verteilen dynamischer Methoden dauert jedoch etwas länger als das Verteilen gewöhnlicher virtueller Methoden. Wenn eine Methode häufig aufgerufen wird oder ihre Ausführung zu lange dauert, sollten Sie sie als virtuelle und nicht als dynamische Methode deklarieren.

Objekte müssen die Adressen ihrer dynamischen Methoden speichern. Dynamische Methoden verfügen aber über keinen Eintrag in der virtuellen Methodentabelle (VMT). Statt dessen werden sie separat aufgeführt. Die Liste der dynamischen Methoden enthält nur Einträge für Methoden, die von einer bestimmten Klasse eingeführt oder überschrieben wurden (die virtuelle Methodentabelle umfaßt im Gegensatz dazu alle virtuellen Methoden des Objekts, die geerbten ebenso wie die eingeführten). Geerbte dynamische Methoden werden verteilt, indem die dynamische Methodenliste des Vorfahren durchsucht wird (in der Vererbungshierarchie von unten nach oben).

Um eine Methode als dynamische Methode zu definieren, fügen Sie am Ende der Methodendeklaration die Direktive **dynamic** hinzu.

Abstrakte Klassenelemente

Wenn eine Methode in einer Vorfahrklasse als **abstract** deklariert ist, müssen Sie sie (durch Neudeklaration und Implementierung) in einer abgeleiteten Komponente sichtbar machen, damit sie in Anwendungen eingesetzt werden kann. Delphi kann keine Instanzen einer Klasse erzeugen, die abstrakte Elemente enthält. Informationen über das Sichtbarmachen geerbter Elemente einer Klasse finden Sie in Kapitel 33, »Eigenschaften erstellen«, und in Kapitel 35, »Methoden erzeugen«.

Klassen und Zeiger

Alle Klassen (und deshalb auch alle Komponenten) sind im Grunde Zeiger. Der Compiler dereferenziert Klassenzeiger automatisch, so daß Sie sich im Normalfall nicht damit befassen müssen. Der Status von Klassen, die als Zeiger fungieren, wird in dem Augenblick bedeutsam, in dem Sie eine Klasse als Parameter übergeben. Generell sollten Sie Klassen als Wert und nicht als Referenz übergeben. Der Grund dafür liegt darin, daß Klassen bereits Zeiger sind. Wenn Sie eine Klasse als Referenz übergeben, übergeben Sie also eigentlich eine Referenz auf eine Referenz.

Kapitel

33

Eigenschaften erstellen

Eigenschaften sind die sichtbaren Teile einer Komponente. Der Anwendungsentwickler kann sie zur Entwurfszeit sehen und bearbeiten. Er erhält eine sofortige Rückmeldung, wenn die Komponenten im Formular-Designer reagieren. Gut programmierte Eigenschaften erleichtern anderen Benutzern die Verwendung Ihrer Komponenten. Sie selbst können sich damit die Pflege der Komponenten vereinfachen.

Um Eigenschaften in Ihren Komponenten optimal einsetzen zu können, sollten Sie folgende Themen beherrschen:

- Wozu dienen Eigenschaften?
- Typen von Eigenschaften
- Geerbte Eigenschaften als published deklarieren
- Eigenschaften definieren
- Array-Eigenschaften erzeugen
- Eigenschaften speichern und laden

Wozu dienen Eigenschaften?

Aus der Sicht des Anwendungsentwicklers stellen sich Eigenschaften wie Variablen dar. Entwickler können die Eigenschaftswerte so setzen und lesen, wie sie es bei Feldern ausführen würden. (Das einzige, was Sie zwar mit einer Variable, nicht aber mit einer Eigenschaft tun können, ist die Übergabe als **var**-Parameter.)

Eigenschaften sind aus folgenden Gründen leistungsfähiger als einfache Felder:

- Anwendungsentwickler können Eigenschaften zur Entwurfszeit setzen. Im Gegensatz zu Methoden, die nur zur Laufzeit verfügbar sind, ermöglichen Eigenschaften dem Entwickler die Anpassung von Komponenten, bevor eine Anwendung ausgeführt wird. Eigenschaften können im Objektinspektor angezeigt wer-

den, wodurch die Programmierarbeit erleichtert wird. Sie müssen nun nicht mehr verschiedene Parameter zur Konstruktion eines Objekts bearbeiten, weil Delphi die Werte aus dem Objektinspektor ermittelt. Der Objektinspektor überprüft außerdem sofort die erfolgten Eigenschaftszuweisungen auf ihre Gültigkeit.

- Eigenschaften können Implementierungsdetails verbergen. Beispielsweise lassen sich Daten, die intern in einem verschlüsselten Formular gespeichert sind, unverschlüsselt als Wert einer Eigenschaft anzeigen. Auch wenn der Wert eine einfache Zahl ist, kann die Komponente unter Umständen den Wert in einem Lookup-Vorgang in der Datenbank ermitteln oder mit Hilfe komplexerer Berechnungen erstellen. Über Eigenschaften können Sie komplexe Effekte mit einfach erscheinenden Zuweisungen erreichen. Was wie eine Zuweisung an ein Feld aussieht, kann der Aufruf einer Methode sein, die eine ausgefeilte Bearbeitung implementiert.
- Eigenschaften können virtuell sein. Was für einen Anwendungsentwickler wie eine einzige Eigenschaft aussieht, kann daher in verschiedenen Komponenten unterschiedlich implementiert sein.

Ein einfaches Beispiel dafür stellt die Eigenschaft *Top* dar, die zu allen Steuerelementen gehört. Die Zuweisung eines neuen Wertes an *Top* ändert nicht nur den gespeicherten Wert, sondern positioniert und zeichnet das Element erneut. Wenn eine Eigenschaft gesetzt wird, betrifft das nicht zwangsläufig nur eine Komponente. Wird beispielsweise die Eigenschaft *Down* eines Mauspalettenschalters auf *True* gesetzt, erhält diese Eigenschaft für alle anderen Mauspalettenschalter in derselben Gruppe den Wert *False*.

Typen von Eigenschaften

Eine Eigenschaft kann einen beliebigen Typ haben. Unterschiedliche Typen werden auch unterschiedlich im Objektinspektor angezeigt. Dieser überprüft die Eigenschaftszuweisungen auf ihre Gültigkeit, und zwar unmittelbar bei ihrer Erstellung zur Entwurfszeit.

Tabelle 33.1 Die Anzeige von Eigenschaften im Objektinspektor

Eigenschaftstyp	Behandlung im Objektinspektor
Einfach	Numerische, Zeichen- und String-Eigenschaften werden als Zahlen, Zeichen oder Strings angezeigt. Der Anwendungsentwickler kann den Wert der Eigenschaft direkt ändern.
Aufzählung	Eigenschaften mit Aufzählungstyp (einschließlich Boolescher Typen) werden als bearbeitbare Strings angezeigt. Der Entwickler kann die verschiedenen Werte durchsehen, indem er in der Wertspalte doppelklickt. Es ist eine Dropdown-Liste vorhanden, in der die möglichen Werte aufgeführt sind.
Menge	Eigenschaften mit Mengentyp werden in der entprechenden Klammernotation angezeigt. Durch Doppelklicken auf die Eigenschaft kann der Entwickler die Menge erweitern und jedes Element als Booleschen Wert behandeln (wenn es zur Menge gehört).

Tabelle 33.1 Die Anzeige von Eigenschaften im Objektinspektor (Fortsetzung)

Eigenschaftstyp	Behandlung im Objektinspektor
Objekt	Eigenschaften, die selbst Klassen sind, besitzen häufig eigene Eigenschafts- Editoren, die in der Registrierungsprozedur der Komponente angegeben sind. Wenn die von einer Eigenschaft gebildete Klasse eigene Eigenschaften besitzt, die als **published** deklariert sind, kann der Entwickler im Objektinspektor die Liste erweitern (durch Doppelklicken), um diese Eigenschaften hinzuzufügen und einzeln zu bearbeiten. Objekteigenschaften müssen von der Klasse *TPersistent* abgeleitet werden.
Array	Array-Eigenschaften müssen über eigene Eigenschafts-Editoren verfügen. Der Objektinspektor unterstützt ihre Bearbeitung nicht. Sie können einen Eigenschafts-Editor erstellen, wenn Sie die Komponenten registrieren.

Geerbte Eigenschaften als published deklarieren

Alle Komponenten erben Eigenschaften von ihren Vorfahrklassen. Wenn Sie eine neue Komponente von einer vorhandenen ableiten, erbt diese alle Eigenschaften des direkten Vorfahren. Führen Sie eine Ableitung von einer abstrakten Klasse durch, sind viele der geerbten Eigenschaften als **protected** oder **public**, nicht aber als **published** deklariert.

Um eine als **protected** oder **public** deklarierte Eigenschaft zur Entwurfszeit im Objektinspektor bereitzustellen, müssen Sie die Eigenschaft als **published** neu deklarieren. Neu deklarieren bedeutet, daß Sie zu der Deklaration der untergeordneten Klasse eine Deklaration für die geerbte Eigenschaft hinzufügen.

Wenn Sie beispielsweise eine Komponente von *TWinControl* ableiten, erbt diese die als **protected** deklarierte Eigenschaft *Ctl3D*. Durch die Neudeklaration von *Ctl3D* in der neuen Komponente ändern Sie das Sichtbarkeitsattribut in **public** oder **published**.

Der folgende Quelltextausschnitt zeigt die Neudeklaration von *Ctl3D* als **published**, wodurch die Eigenschaft zur Entwurfszeit verfügbar wird.

```
type
  TSampleComponent = class(TWinControl)
  published
    property Ctl3D;
  end;
```

Wenn Sie eine Eigenschaft neu deklarieren, legen Sie nur den Namen der Eigenschaft fest, nicht aber den Typ oder andere Informationen, die im folgenden Abschnitt, »Eigenschaften definieren«, beschrieben sind. Sie können außerdem neue Standardwerte deklarieren und angeben, ob die Eigenschaft gespeichert werden soll.

Durch eine Neudeklaration können die Beschränkungen einer Eigenschaft verringert werden. Eine stärkere Beschränkung durch eine Neudeklaration ist nicht möglich. So können Sie eine als **protected** deklarierte Eigenschaft als **public** deklarieren, nicht aber eine als **public** deklarierte Eigenschaft verbergen, indem Sie sie als **protected** neu deklarieren.

Eigenschaften definieren

In diesem Abschnitt wird beschrieben, wie neue Eigenschaften deklariert werden. Außerdem finden Sie hier die Erläuterung einiger Konventionen für Standardkomponenten. Folgende Themen werden behandelt:

- Die Deklaration von Eigenschaften
- Interne Datenspeicherung
- Direkter Zugriff
- Zugriffsmethoden
- Standardwerte von Eigenschaften

Die Deklaration von Eigenschaften

Eine Eigenschaft wird in der Deklaration ihrer Komponentenklasse deklariert. Sie müssen dazu folgendes angeben:

- Den Namen der Eigenschaft.
- Den Typ der Eigenschaft.
- Die Methoden, die zum Lesen und Schreiben des Eigenschaftswertes verwendet werden. Werden keine Zuweisungsmethoden deklariert, hat die Eigenschaft das Attribut »Nur Lesen«.

Eigenschaften, die im **published**-Deklarationsabschnitt der Komponentenklasse deklariert sind, können im Objektinspektor zur Entwurfszeit bearbeitet werden. Der Wert einer **published**-Eigenschaft wird mit der Komponente in der Formulardatei gespeichert. Eigenschaften, die in einem **public**-Abschnitt deklariert sind, stehen dagegen nur zur Laufzeit zur Verfügung und können im Quelltext des Programms gelesen oder gesetzt werden.

Im folgenden sehen Sie eine typische Deklaration für eine Eigenschaft namens *Count*.

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;                        { wird für interne Speicherung verwendet}
    procedure SetCount (Value: Integer);    { write-Methode }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

Interne Datenspeicherung

Für die Art und Weise, in der Daten für eine Eigenschaft gespeichert werden, gibt es keine Beschränkungen. Grundsätzlich gelten hierzu in Delphi folgende Konventionen:

- Eigenschaftsdaten werden in Klassenfeldern gespeichert.
- Die zum Speichern verwendeten Felder sind als private deklariert. Der Zugriff darauf sollte nur von der Komponente selbst aus erfolgen. Abgeleitete Komponenten sollten die geerbte Eigenschaft verwenden. Sie benötigen keinen direkten Zugriff auf den internen Datenspeicher der Eigenschaft.
- Die Bezeichner für diese Felder bestehen aus dem Buchstaben *F* und dem Namen der Eigenschaft. So werden beispielsweise die Rohdaten für die Eigenschaft *Width*, die in *TControl* definiert ist, in einem Feld namens *FWidth* gespeichert.

Das diesen Konventionen zugrundeliegende Prinzip beruht darauf, daß nur die Implementierungsmethoden für eine Eigenschaft auf die dahinterliegenden Daten zugreifen dürfen. Wenn eine Methode oder eine andere Eigenschaft diese Daten ändern muß, muß dies über die Eigenschaft und nicht über einen direkten Zugriff auf die gespeicherten Daten erfolgen. Dadurch ist sichergestellt, daß die Implementierung einer geerbten Eigenschaft geändert werden kann, ohne daß dabei abgeleitete Komponenten ungültig werden.

Direkter Zugriff

Die einfachste Art, Eigenschaftsdaten verfügbar zu machen, ist der direkte Zugriff. Mit anderen Worten: In den **read**- und **write**-Abschnitten der Eigenschaftsdeklaration wird festgelegt, daß Schreib- und Lesezugriffe auf den Eigenschaftswert direkt an das interne Speicherfeld gehen, ohne daß eine Zugriffsmethode aufgerufen wird. Der direkte Zugriff ist von Nutzen, wenn Sie eine Eigenschaft im Objektinspektor bereitstellen wollen, die Änderung des Wertes aber keine sofortige Aktion auslösen soll.

Es ist üblich, einen direkten Zugriff auf den **read**-Abschnitt einer Eigenschaftsdeklaration zu gestatten, während für den **write**-Abschnitt eine Zugriffsmethode verwendet werden muß. Dadurch kann der Status der Komponente aktualisiert werden, wenn sich der Eigenschaftswert ändert.

Die folgende Deklaration eines Komponententyps zeigt eine Eigenschaft mit direktem Zugriff auf die **read**- und **write**-Abschnitte.

```
type
      TSampleComponent = class(TComponent)
      private { Interner Speicher ist als private deklariert}
         FMyProperty: Boolean; {Feld zum Speichern des Eigenschaftswerts deklarieren}
      published { Eigenschaft zur Entwurfszeit bereitstellen }
         property MyProperty: Boolean read FMyProperty write FMyProperty;
      end;
```

Zugriffsmethoden

Sie können in den **read**- und **write**-Abschnitten einer Eigenschaftsdeklaration anstelle eines Feldes eine Zugriffsmethode definieren. Zugriffsmethoden sind normalerweise als **virtuell** deklariert und sollten außerdem als **protected** deklariert werden. Dadurch ist es abgeleiteten Komponenten möglich, die Implementierung der Eigenschaft zu überschreiben.

Vermeiden Sie es, Zugriffsmethoden als **public** zu deklarieren. Die Deklaration als **protected** stellt sicher, daß Anwendungsentwickler eine Eigenschaft nicht versehentlich durch einen Aufruf einer dieser Methoden ändern.

Die folgende Klasse deklariert drei Eigenschaften mit Hilfe des Index-Spezifizierers. Er erlaubt den drei Eigenschaften, die gleichen Zugriffsmethoden zu verwenden:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write
SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; {Beachten Sie den Indexparameter}
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
```

Da jedes Datumselement (Tag, Monat und Jahr) vom Typ **int** ist und jede Zuweisung die Neucodierung des Datums erfordert, können sich die drei Eigenschaften die **read**- und **write**-Methoden teilen. Sie benötigen nur eine Methode, um ein Datenelement zu lesen, und eine andere, um in ein Datenelement zu schreiben.

Hier die **read**-Methode, die das Datenelement ausliest:

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);     { Zerlegt das Datum in Einzelelemente}
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

Hier die **write**-Methode, die das entsprechende Datumselement zuweist:

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                              { Alle Elemente müssen positiv sein }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);      { Datumselemente ermitteln }
```

```
    case Index of        { Neues Element in Abhängigkeit vom Index setzen }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);     { Geändertes Datum kodieren}
    Refresh;                                      { Sichtbaren Kalender aktualisieren}
  end;
end;
```

Die Methode read

Die read-Methode einer Eigenschaft ist eine Funktion ohne Parameter (mit einer noch im folgenden erläuterten Ausnahme). Sie liefert einen Wert zurück, der denselben Typ wie die Eigenschaft aufweist. Per Konvention wird der Name der Funktion aus dem Wort *Get* und dem Eigenschaftsnamen gebildet. So würde beispielsweise die Abrufmethode für eine Eigenschaft namens *Count GetCount* lauten. Die Abrufmethode ändert den internen Datenspeicher, wenn dies für die Erstellung des Eigenschaftswertes im entsprechenden Typ erforderlich ist.

Die einzige Ausnahme zu der obengenannten Parameter-Regel bilden die Array-Eigenschaften und diejenigen Eigenschaften, die Index-Spezifizierer verwenden (siehe »Array-Eigenschaften erstellen« auf Seite 33-9). Diese beiden Eigenschaftstypen übergeben ihre Indexwerte als Parameter. (Verwenden Sie Index-Spezifizierer zur Erstellung einer read-Methode, die von mehreren Eigenschaften gemeinsam genutzt wird. Informationen über Index-Spezifizierer finden Sie in der *Object Pascal Sprachreferenz*.)

Wenn Sie keine read-Methode deklarieren, hat die Eigenschaft das Attribut »Nur-Schreiben«. Nur-Schreiben-Eigenschaften werden jedoch nur selten verwendet.

Die Methode write

Die Methode write für eine Eigenschaft ist eine Prozedur mit einem einzigen Parameter (mit einer noch im folgenden erläuterten Ausnahme), der denselben Typ wie die Eigenschaft hat. Der Parameter kann als Referenz oder als Wert übergeben werden und einen beliebigen Namen erhalten. Per Konvention wird der Name der Methode write aus dem Wort *Set* und dem Namen der Eigenschaft gebildet. So würde beispielsweise die write-Methode für eine Eigenschaft namens *Count SetCount* lauten. Der im Parameter übergebene Wert wird zum neuen Wert der Eigenschaft. Die Methode write muß die Bearbeitungen durchführen, die für das Einbringen der entsprechenden Daten in den internen Speicher der Eigenschaft erforderlich sind.

Die einzige Ausnahme zu der oben erwähnten Parameter-Regel bilden die Array-Eigenschaften und diejenigen Eigenschaften, die Index-Spezifizierer verwenden. Diese beiden Eigenschaftstypen übergeben ihre Indexwerte als zweiten Parameter. (Verwenden Sie Index-Spezifizierer zur Erstellung einer write-Methode, die von mehreren Eigenschaften gemeinsam genutzt wird. Informationen über Index-Spezifizierer finden Sie in der *Object Pascal Sprachreferenz*.)

Wenn Sie keine write-Methode deklarieren, hat die Eigenschaft das Attribut »Nur-Lesen«. Damit eine **published**-Eigenschaft zur Entwurfszeit verwertbar ist, muß Sie als read / write definiert sein.

write-Methoden überprüfen normalerweise vor der Änderung der Eigenschaft, ob sich ein neuer Wert vom aktuellen Wert unterscheidet. Im folgenden sehen Sie eine einfache write-Methode für eine Integer-Eigenschaft namens *Count*, die ihren aktuellen Wert in einem Feld namens *FCount* speichert.

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

Standardwerte für Eigenschaften

Bei der Deklaration einer Eigenschaft können Sie einen Standardwert angeben. Delphi legt mit Hilfe des Standardwertes fest, ob die Eigenschaft in einer Formulardatei gespeichert wird. Wenn Sie für eine Eigenschaft keinen Standardwert festlegen, speichert Delphi die Eigenschaft immer.

Zur Definition eines Standardwertes für eine Eigenschaft fügen Sie die Direktive **default** und den Standardwert an die Deklaration (oder Neudeklaration) der Eigenschaft an, z. B.:

```
property Cool Boolean read GetCool write SetCool default True;
```

Hinweis Durch die Deklaration eines Standardwertes wird die Eigenschaft nicht mit diesem Wert belegt. Die Eigenschaftswerte müssen bei Bedarf mit Hilfe des Methodenkonstruktors initialisiert werden. Da jedoch die Felder von Objekten immer mit 0 initialisiert werden, ist es nicht unbedingt erforderlich, daß der Konstruktor Integer-Eigenschaften auf 0 und Boolesche Eigenschaften auf *False* setzt.

Keinen Standardwert angeben

Bei einer Neudeklaration können Sie festlegen, daß die Eigenschaft keinen Standardwert besitzt. Dies ist auch dann möglich, wenn die geerbte Eigenschaft einen Standardwert festlegt.

Wenn Sie angeben wollen, daß eine Eigenschaft keinen Standardwert hat, fügen Sie die Direktive nodefault zu der Eigenschaftsdeklaration hinzu. Ein Beispiel:

```
property FavoriteFlavor string nodefault;
```

Bei der erstmaligen Deklaration einer Eigenschaft muß die Direktive **nodefault** nicht angegeben werden. Das Nichtvorhandensein eines deklarierten Standardwertes impliziert, daß kein Standardwert existiert.

Im folgenden sehen Sie die Deklaration einer Komponente, die eine einzige Boolesche Eigenschaft namens *IsTrue* mit dem Standardwert *True* enthält. Nach der Dekla-

ration (im Implementierungsabschnitt der Unit) folgt der Konstruktor, der die Eigenschaft initialisiert.

```
type
      TSampleComponent = class(TComponent)
      private
         FIsTrue: Boolean;
      public
         constructor Create(AOwner: TComponent); override;
      published
         property IsTrue: Boolean read FIsTrue write FIsTrue default True;
      end;
⋮
constructor TSampleComponent.Create(AOwner: TComponent);
begin
      inherited Create(AOwner);              { Geerbten Konstruktor aufrufen }
      FIsTrue := True;                       { Standardwert setzen }
end;
```

Array-Eigenschaften erstellen

Einige Eigenschaften können wie Arrays indiziert werden. Beispielsweise handelt es sich bei der Eigenschaft *Lines* von *TMemo* um eine indizierte Liste mit Strings, die den Text des Memofeldes bilden. Sie können diese Eigenschaft als String-Array behandeln. Lines ermöglicht einen Zugriff auf ein bestimmtes Element (einen String) in einer größeren Datenmenge (dem Memofeldtext).

Array-Eigenschaften werden wie andere Eigenschaften deklariert, wobei jedoch folgende Ausnahmen zu beachten sind:

- Die Deklaration enthält einen oder mehrere Indizes mit angegebenen Typen. Die Indizes können von einem beliebigen Typ sein.
- Die **read**- und **write**-Abschnitte der Eigenschaftsdeklaration müssen (falls vorhanden) Methoden sein. Es darf sich bei ihnen nicht um Felder handeln.

Die read- und write-Methoden für eine Array-Eigenschaft übernehmen zusätzliche Parameter, die den Indizes entsprechen. Diese Parameter müssen in derselben Reihenfolge angeordnet und von demselben Typ wie die in der Deklaration angegebenen Indizes sein.

Zwischen Array-Eigenschaften und Arrays bestehen einige wichtige Unterschiede. Im Gegensatz zum Index eines Arrays muß der Index einer Array-Eigenschaft nicht vom Typ *Integer* sein. Sie können beispielsweise eine Eigenschaft für einen String indizieren. Außerdem lassen sich nur einzelne Elemente einer Array-Eigenschaft referenzieren, nicht aber der gesamte Bereich der Eigenschaft.

Das folgende Beispiel zeigt die Deklaration einer Eigenschaft, die einen String, basierend auf einem Integer-Index, zurückliefert.

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
```

```
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
⋮
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

Eigenschaften speichern und laden

Delphi speichert Formulare und ihre Komponenten in Formulardateien (.DFM). Eine Formulardatei ist die binäre Entsprechung der Eigenschaften eines Formulars und seiner Komponenten. Wenn Delphi-Entwickler die von Ihnen geschriebenen Komponenten ihren Formularen hinzufügen, müssen diese Komponenten die Fähigkeit besitzen, ihre Eigenschaften in die Formulardatei zu schreiben, wenn diese gespeichert wird. Ebenso müssen sie sich selbst aus der Formulardatei wiederherstellen können, wenn sie in Delphi geladen oder als Teil einer Anwendung ausgeführt werden.

In den meisten Fällen brauchen Sie sich nicht mit dem Zusammenwirken Ihrer Komponenten und der Formulardatei zu beschäftigen, da die erforderlichen Fähigkeiten bereits zum geerbten Verhalten der Komponenten gehören. Manchmal ist es allerdings doch erforderlich, in den Speicher- und Ladevorgang einzugreifen. In diesen Fällen sollten Sie den zugrundeliegenden Mechanismus kennen.

Dazu gehören die folgenden Punkte:

- Der Speicher- und Lademechanismus
- Standardwerte festlegen
- Was soll gespeichert werden?
- Initialisieren nach dem Laden
- Nicht als published deklarierte Eigenschaften speichern und laden

Der Speicher- und Lademechanismus

Die Beschreibung eines Formulars besteht neben einer Liste der Formulareigenschaften aus den Beschreibungen aller Komponenten des Formulars. Dabei ist jede Komponente einschließlich des Formulars selbst für das Speichern und Laden seiner eigenen Beschreibung verantwortlich.

Standardmäßig speichert eine Komponente die Werte aller als **public** und **published** deklarierten Eigenschaften in der Reihenfolge ihrer Deklaration. Beim Laden baut

sich die Komponente zunächst selbst auf, indem sie alle Eigenschaften auf die Standardwerte setzt. Erst dann werden die gespeicherten, von der Vorgabe abweichenden Werte gelesen.

Dieser Standardmechanismus reicht für die meisten Komponenten aus und erfordert von seiten des Entwicklers keinerlei Eingriffe. Es gibt aber verschiedene Möglichkeiten, den Speicher- und Ladevorgang für bestimmte Komponenten anzupassen.

Standardwerte festlegen

Delphi-Komponenten speichern ihre Eigenschaftswerte nur, wenn sie von der Vorgabe abweichen. Wenn Sie nichts gegenteiliges festlegen, geht Delphi davon aus, daß eine Eigenschaft keinen Standardwert hat, was bedeutet, daß Komponenten ihre Eigenschaften unabhängig vom Wert immer speichern.

Um für eine Eigenschaft einen Standardwert festzulegen, fügen Sie die Direktive **default** und den neuen Standardwert an das Ende der Eigenschaft-Deklaration ein.

Sie können auch einen Standardwert festlegen, wenn Sie eine Eigenschaft redeklarieren. Tatsächlich ist dies einer der möglichen Gründe, eine Eigenschaft zu redeklarieren.

Hinweis Wenn Sie für eine Eigenschaft einen Standardwert festlegen, wird dieser beim Erzeugen des Objekts nicht automatisch zugewiesen. Sie müssen dafür sorgen, daß der Konstruktor des Objekts den Wert zuweist. Eine Eigenschaft, deren Wert nicht vom Konstruktor der Komponente gesetzt wird, erhält den Wert, der für die Komponente dem Wert Null entspricht. Bei numerischen Eigenschaften ist dies 0, bei Booleschen Eigenschaften *False*, bei Zeigern **nil** usw. Wenn Sie sich nicht ganz sicher sind, weisen Sie auf jeden Fall im Konstruktor einen Wert zu.

Der folgende Quelltext zeigt eine Komponentendeklaration, in der ein Standardwert für die Eigenschaft *Align* festgelegt wird, und die Implementation des Konstruktors, in dem der Standardwert zugewiesen wird. Die neue Komponente, abgeleitet von *TPanel*, soll in Fenstern als Statusleiste verwendet werden. Als Standardausrichtung bietet sich daher der untere Rand der übergeordneten Komponente an.

```
type
      TStatusBar = class(TPanel)
      public
         constructor Create(AOwner: TComponent); override; { überschreiben, um neuen
                                                             Standard zu setzen }
      published
         property Align default alBottom; { Neudeklaration mit neuem Standardwert }
      end;
⋮
constructor TStatusBar.Create(AOwner: TComponent);
begin
      inherited Create(AOwner); { Geerbte Initialisierung ausführen }
      Align := alBottom; { Align einen neuen Standardwert zuweisen}
end;
```

Was soll gespeichert werden?

Sie können festlegen, ob Delphi alle Eigenschaften Ihrer Komponenten speichern soll. Standardmäßig werden alle Eigenschaften im **published**-Abschnitt der Klassendeklaration gespeichert. Sie können eine bestimmte Eigenschaft grundsätzlich nicht speichern oder eine Funktion bereitstellen, die zur Laufzeit bestimmt, ob die Eigenschaft gespeichert wird.

Um festzulegen, ob Delphi eine Eigenschaft speichern soll, fügen Sie der Eigenschaftsdeklaration die Direktive **stored** hinzu, gefolgt von *True, False* oder dem Namen einer Booleschen Methode.

Der folgende Quelltext zeigt eine Komponente, die drei neue Eigenschaften deklariert. Die erste Eigenschaft wird immer gespeichert, die zweite nie, und die dritte in Abhängigkeit vom Rückgabewert einer Booleschen Methode:

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public                                          { In der Regel nicht gespeichert }
    property Important: Integer stored True;      { Immer gespeichert }
  published                                       { In der Regel immer gespeichert }
    property Unimportant: Integer stored False;   { Niemals gespeichert}
    property Sometimes: Integer stored StoreIt;   { Abhängig vom Funktionswert}
  end;
```

Initialisieren nach dem Laden

Nachdem eine Komponente alle Eigenschaftswerte aus ihrer gespeicherten Beschreibung gelesen hat, ruft sie eine virtuelle Methode namens *Loaded* auf, die alle nötigen Initialisierungen vornimmt. Da dieser Aufruf stattfindet, bevor das Formular und seine Komponenten angezeigt werden, verursacht er kein Bildschirmflackern.

Um eine Komponente zu initialisieren, nachdem sie ihre Eigenschaftswerte geladen hat, müssen Sie die Methode *Loaded* überschreiben.

Hinweis Die erste Anweisung in einer *Loaded*-Methode ist immer der Aufruf der geerbten Methode *Loaded*. Dadurch ist sichergestellt, daß vor allen benutzerdefinierten Initialisierungen alle geerbten Eigenschaften korrekt initialisiert werden.

Der folgende Quelltext stammt aus der Komponente *TDatabase*. Nachdem Sie geladen wird, versucht die Datenbank, alle Verbindungen wiederherzustellen, die zum Zeitpunkt geöffnet waren, als sie gespeichert wurde. Sie gibt außerdem an, wie Exceptions, die eventuell während des Verbindungsversuchs auftreten, zu behandeln sind.

```
procedure TDatabase.Loaded;
begin
      inherited Loaded; { Als erstes die geerbte Methode aufrufen }
      try
         if FStreamedConnected then Open { Die Verbindungen wiederherstellen }
         else CheckSessionName(False);
```

```
      except
        if csDesigning in ComponentState then { zur Entwurfszeit soll ... }
        Application.HandleException(Self) { ... Delphi die Exception behandeln,}
        else raise; { ... ansonsten erneut auslösen }
      end;
end;
```

Nicht als published deklarierte Eigenschaften speichern und laden

Standardmäßig werden nur **published**-Eigenschaften mit einer Komponente geladen und gespeichert. Es ist jedoch möglich, auch nicht als **published** deklarierte Eigenschaften zu laden und zu speichern. Sie können also persistente Eigenschaften verwenden, die nicht im Objektinspektor angezeigt werden. Außerdem können Komponenten Eigenschaftswerte speichern und laden, die Delphi nicht lesen oder schreiben kann, da der Wert der Eigenschaft zu komplex strukturiert ist. Ein Objekt des Typs *TStrings* kann beispielsweise nicht den automatischen Delphi-Mechanismus zum Laden und Speichern der repräsentierten Strings nutzen, sondern muß statt dessen mit dem folgenden Verfahren verarbeitet werden.

Sie können nicht als **published** deklarierte Eigenschaften laden und speichern, indem Sie Quelltext schreiben, der Delphi zum Laden bzw. Speichern des Eigenschaftswertes veranlaßt.

Mit den folgenden Schritten können Sie Quelltext zum Laden und Speichern von Eigenschaften schreiben:

1. Methoden zum Speichern und Laden von Eigenschaftswerten erstellen.
2. Die Methode *DefineProperties* überschreiben. Die Methoden werden an ein Filer-Objekt übergeben.

Methoden zum Speichern und Laden von Eigenschaftswerten erstellen

Damit nicht als **published** deklarierte Eigenschaften gespeichert und geladen werden können, müssen Sie zunächst eine Methode zum Speichern und eine weitere zum Laden des Eigenschaftswerts erstellen. Sie haben zwei Möglichkeiten:

- Erstellen Sie eine Methode des Typs *TWriterProc* zum Speichern und eine des Typs *TReaderProc* zum Laden des Eigenschaftswerts. Dieses Vorgehen ermöglicht die Nutzung der integrierten Delphi-Funktionen zum Speichern und Laden einfacher Typen. Verwenden Sie dieses Verfahren, wenn der betreffende Eigenschaftswert aus Typen gebildet wurde, die Delphi speichern und laden kann.
- Erstellen Sie zwei Methoden des Typs *TStreamProc*, eine zum Speichern und eine zum Laden des Eigenschaftswerts. *TStreamProc* nimmt einen Stream als Argument entgegen. Mit den Methoden des Stream können Sie die Eigenschaftswerte schreiben und lesen.

Ein Beispiel: Eine Eigenschaft repräsentiert eine Komponente, die zur Laufzeit erstellt wird. Delphi kann diesen Wert zwar grundsätzlich schreiben, das Schreiben erfolgt jedoch nicht, da die Komponente nicht im Formular-Designer erstellt wurde. Da das Stream-System Komponenten speichern und laden kann, können Sie das erste Verfahren einsetzen. Die folgenden Methoden laden und speichern die dynamisch erstellte Komponente, die ein Wert einer Eigenschaft namens *MyCompProperty* ist:

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

Die Methode DefineProperties überschreiben

Nachdem Sie Methoden zum Speichern und Laden des Eigenschaftswertes erstellt haben, können Sie die Methode *DefineProperties* der Komponente überschreiben. Delphi ruft diese Methode auf, wenn die Komponente geladen oder gespeichert wird. In der Methode *DefineProperties* müssen Sie die Methode *DefineProperty* oder *DefineBinaryProperty* des aktuelle Filer-Objekts aufrufen und dieser die Methode zum Laden oder Speichern des Eigenschaftswertes übergeben. Haben Sie Methoden des Typs *TWriterProc* und *TReaderProc* zum Speichern bzw. Laden erstellt, rufen Sie die Methode *DefineProperty* des Filer-Objekts auf. Sind die Methoden dagegen vom Typ *TStreamProc*, müssen Sie statt dessen *DefineBinaryProperty* aufrufen.

Unabhängig von der zum Definieren der Eigenschaft verwendeten Methode übergeben Sie die Methoden zum Speichern und Laden des Eigenschaftswertes sowie einen Booleschen Wert, der bestimmt, ob der Eigenschaftswert geschrieben werden muß. Kann der Wert geerbt werden oder liegt ein Standardwert vor, ist das Schreiben nicht erforderlich.

Liegen beispielsweise die Methode *LoadCompProperty* des Typs *TReaderProc* und die Methode *StoreCompProperty* des Typs *TWriterProc* vor, können Sie *DefineProperties* folgendermaßen überschreiben:

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { Vorfahr auf zu erbenden Wert prüfen }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else if MyCompProperty = nil or
         TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
        Result := True
      else Result := False;
    end
```

```
    else { Kein geerbter Wert -- auf Standardwert (nil) prüfen }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { Basisklassen können die Eigenschaften definieren }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
```

Kapitel

34

Ereignisse erzeugen

Ein Ereignis ist eine Verknüpfung zwischen einem Vorgang im System (z. B. Benutzeraktion oder Fokusänderung) und einem Quelltextabschnitt, der auf diesen Vorgang reagiert. Der antwortende Quelltext steht in einer *Ereignisbehandlungsroutine*, die nahezu immer vom Anwendungsentwickler geschrieben wird. Über Ereignisse kann der Anwendungsentwickler das Verhalten von Komponenten anpassen, ohne daß die Klassen selbst geändert werden müssen. Dieses Prinzip wird als *Delegieren* bezeichnet.

Ereignisse für die gängigsten Benutzeraktionen (z. B. Mausaktionen) sind in allen Standardkomponenten integriert. Sie können aber jederzeit neue Ereignisse definieren. Wenn Sie Ereignisse in einer Komponente erstellen wollen, sollten Ihnen folgende Themen geläufig sein:

- Was sind Ereignisse?
- Standardereignisse implementieren
- Eigene Ereignisse definieren

Da Ereignisse als Eigenschaften implementiert werden, sollten Sie mit den in Kapitel 33, »Eigenschaften erstellen«, behandelten Themen bereits vertraut sein, bevor Sie mit der Erstellung oder Änderung eines Komponentenereignisses beginnen.

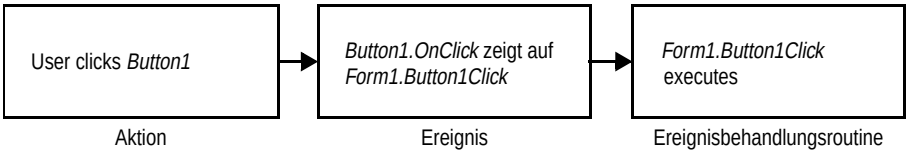
Was sind Ereignisse?

Ein Ereignis ist ein Mechanismus, der einen Vorgang mit einem Programmquelltext verknüpft. Genauer gesagt handelt es sich bei einem Ereignis um einen Methodenzeiger, der auf eine Methode in einer bestimmten Klasseninstanz zeigt.

Aus der Sicht des Anwendungsentwicklers besteht ein Ereignis lediglich aus einem mit einem Systemvorgang verknüpften Namen (z. B. *OnClick*), dem ein bestimmter Programmquelltext zugeordnet werden kann. Beispielsweise verfügt eine Schaltfläche namens *Button1* über die Methode *OnClick*. Per Voreinstellung generiert Delphi

eine Ereignisbehandlungsroutine namens *Button1Click* in dem Formular, das die Schaltfläche enthält, und weist diese Routine dem Ereignis *OnClick* zu. Sobald ein Klick-Ereignis auf der Schaltfläche eintritt, ruft die Schaltfläche die Methode auf, die *OnClick* zugewiesen wurde. Im vorliegenden Fall ist dies *Button1Click*.

Wenn Sie ein Ereignis schreiben wollen, sollte Ihnen folgendes bekannt sein:



- Ereignisse sind Methodenzeiger.
- Ereignisse sind Eigenschaften.
- Ereignistypen sind Methodenzeigertypen.
- Ereignisbehandlungstypen sind Prozeduren.
- Ereignisbehandlungsroutinen sind optional.

Ereignisse sind Methodenzeiger

Delphi implementiert Ereignisse mit Hilfe von Methodenzeigern. Ein Methodenzeiger ist ein spezieller Zeigertyp, der auf eine bestimmte Methode in einem Instanzobjekt zeigt. Als Komponentenentwickler können Sie den Methodenzeiger als Platzhalter verwenden. Sobald der Quelltext feststellt, daß ein Ereignis eintritt, wird die Methode aufgerufen (falls vorhanden), die vom Benutzer für dieses Ereignis vorgesehen wurde.

Methodenzeiger verhalten sich wie andere prozedurale Typen, verwalten aber einen verborgenen Zeiger auf ein Objekt. Wenn der Anwendungsentwickler einem Komponentenereignis eine Behandlungsroutine zuweist, erfolgt die Zuweisung nicht an eine Methode mit einem bestimmten Namen, sondern an eine Methode in einem bestimmten Instanzobjekt. Dieses Objekt ist normalerweise das Formular, das die Komponente enthält (was aber nicht der Fall sein muß).

Alle Steuerelemente erben eine dynamische Methode mit der Bezeichnung *Click*. Diese wird für die Behandlung der Klick-Ereignisse verwendet:

```
procedure Click; dynamic;
```

Die Implementierung von *Click* ruft die *Click*-Ereignisbehandlungsroutine auf, falls eine vorhanden ist. Hat der Anwender dem Ereignis *OnClick* eines Steuerelements eine Behandlungsroutine zugewiesen, bewirkt das Anklicken des Steuerelements den Aufruf dieser Methode. Wurde keine Ereignisbehandlungsroutine zugewiesen, passiert nichts.

Ereignisse sind Eigenschaften

Komponenten implementieren ihre Ereignisse in Form von Eigenschaften. Im Gegensatz zu den meisten anderen Eigenschaften setzen Ereignisse keine Methoden zur Implementierung ihrer **read**- und **write**-Abschnitte ein. Sie bedienen sich statt dessen eines als **private** deklarierten Klassenfeldes, das denselben Typ wie die Eigenschaft aufweist.

Per Konvention besteht der Name des Feldes aus dem Namen der Eigenschaft, dem der Buchstabe *F* vorangestellt wird. Beispielsweise wird der Zeiger der Methode *OnClick* in einem Feld namens *FOnClick* vom Typ *TNotifyEvent* gespeichert. Die Deklaration der Ereigniseigenschaft *OnClick* sieht folgendermaßen aus:

```
type
   TControl = class(TComponent)
   private
    FOnClick: TNotifyEvent;              { Deklaration eines Feldes für den Methodenzeiger }
    ...
   protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
   end;
```

Informationen über *TNotifyEvent* und andere Ereignistypen finden Sie nachfolgend im Abschnitt »Ereignistypen sind Methodenzeigertypen«.

Genauso wie bei anderen Eigenschaften können Sie auch bei Ereignissen den Wert zur Laufzeit setzen oder ändern. Daß Ereignisse als Eigenschaften eingesetzt werden können, hat einen großen Vorteil: Die Benutzer der Komponenten können den Ereignissen zur Entwurfszeit mit Hilfe des Objektinspektors Routinen zuweisen.

Ereignistypen sind Methodenzeigertypen

Da ein Ereignis ein Zeiger auf eine Ereignisbehandlungsroutine ist, muß der Typ der Ereigniseigenschaft ein Methodenzeigertyp sein. Ebenso muß der Quelltext, der für eine Ereignisbehandlungsroutine eingesetzt wird, eine entsprechend typisierte Methode eines Objekts sein.

Alle Ereignisbehandlungsroutinen sind Prozeduren. Damit eine Ereignisbehandlungsroutine zum Ereignis eines bestimmten Typs kompatibel ist, muß die betreffende Methode über dieselbe Anzahl und dieselben Typen von Parametern verfügen, die in derselben Reihenfolge angeordnet sind und auf dieselbe Weise übergeben werden.

Delphi definiert Methodentypen für alle seine Standardereignisse. Wenn Sie Ihre eigenen Ereignisse erstellen, können Sie entweder einen vorhandenen (geeigneten) Typ verwenden oder einen eigenen definieren.

Ereignisbehandlungstypen sind Prozeduren

Obwohl der Compiler zuläßt, daß Sie Methodenzeiger als Funktionen deklarieren, dürfen Sie diese Möglichkeit niemals für die Behandlung von Ereignissen in Betracht ziehen. Da eine leere Funktion ein undefiniertes Ergebnis zurückliefert, kann eine

leere Ereignisbehandlungsroutine, die als Funktion deklariert wurde, unter Umständen ungültig sein. Aus diesem Grund müssen Ereignisse und die mit ihnen verbundenen Behandlungsroutinen immer Prozeduren sein.

Obwohl eine Ereignisbehandlungsroutine keine Funktion sein kann, können Sie trotzdem über den Quelltext des Anwendungsentwicklers mit Hilfe der **var**-Parameter Informationen ermitteln. Dazu müssen Sie sicherstellen, daß dem Parameter vor dem Aufruf der Behandlungsroutine ein gültiger Wert zugewiesen wird. In diesem Fall ist es nicht erforderlich, daß der Benutzerquelltext den Wert ändert.

Ein Beispiel für die Übergabe von **var**-Parametern an eine Ereignisbehandlungsroutine ist das Ereignis *OnKeyPress* vom Typ *TKeyPressEvent*. *TKeyPressEvent* definiert zwei Parameter. Ein Parameter gibt an, von welchem Objekt das Ereignis generiert wurde, der zweite stellt fest, welche Taste gedrückt wurde:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normalerweise enthält der Parameter *Key* das Zeichen, das der Benutzer eingegeben hat. Es gibt aber Situationen, in denen der Benutzer der Komponente das Zeichen ändern möchte. Ein Beispiel dafür wäre die Umwandlung aller Zeichen zu Großbuchstaben mit Hilfe eines Editors. Zu diesem Zweck könnte der Benutzer die folgende Behandlungsroutine für Tastenanschläge definieren:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

Sie können dem Benutzer aber auch mit Hilfe der **var**-Parameter die Möglichkeit zum Überschreiben der Standardbehandlung geben.

Ereignisbehandlungsroutinen sind optional

Beachten Sie bei der Erstellung von Ereignissen, daß die Entwickler, die Ihre Komponenten verwenden, eventuell keine Behandlungsroutinen mit den Ereignissen verbinden. Das heißt, daß Ihre Komponente nicht fehlschlagen oder Fehler verursachen darf, nur weil keine Behandlungsroutine mit einem bestimmten Ereignis verknüpft ist. (Das Aufrufen von Behandlungsroutinen und der Umgang mit Ereignissen, mit denen keine Routinen verknüpft sind, wird im Abschnitt »Ereignisse aufrufen« auf Seite 34-9 erläutert.)

Ereignisse treten überwiegend in Windows-Anwendungen auf. Beispielsweise veranlaßt das einfache Bewegen des Mauszeigers über eine visuelle Komponente Windows dazu, zahlreiche Maus-Ereignisbotschaften zu senden. Diese werden von der Komponente in *OnMouseMove*-Ereignisse übersetzt. In den meisten Fällen werden die Entwickler derartige Mausbewegungsereignisse nicht behandeln. Diese Tatsache darf nicht zum Problem werden. Aus diesem Grund sollten die Komponenten, die Sie erstellen, Behandlungsroutinen für ihre Ereignisse anfordern.

Anwendungsentwickler können jeden gewünschten Quelltext in eine Ereignisbehandlungsroutine integrieren. Die Ereignisse der VCL-Komponenten sind so konstruiert, daß die Möglichkeit des Fehlschlagens einer Behandlungsroutine auf ein Mi-

nimum reduziert wird. Natürlich lassen sich logische Fehler im Anwendungsquelltext nicht völlig ausschließen. Sie können aber sicherstellen, daß die Datenstrukturen vor dem Aufruf von Ereignissen initialisiert werden, so daß die Anwendungsentwickler nicht in die Lage kommen, auf unzulässige Daten zuzugreifen.

Die Standardereignisse implementieren

Die zu Delphi gehörenden Steuerelemente erben die Ereignisse für die geläufigsten Windows-Vorgänge. Sie werden Standardereignisse genannt. Diese Ereignisse sind zwar in die Steuerelemente integriert, aber häufig als **protected** deklariert, so daß die Entwickler keine Behandlungsroutinen mit ihnen verbinden können. Wenn Sie ein Steuerelement erstellen, können Sie festlegen, daß die Ereignisse für die Benutzer des Steuerelements sichtbar sein sollen.

Auf drei Punkte müssen Sie achten, wenn Sie die Standardereignisse in Ihre Steuerelemente integrieren:

- Identifizieren der Standardereignisse
- Sichtbarmachen der Ereignisse
- Ändern der Standard-Ereignisbehandlung

Standardereignisse identifizieren

Es gibt zwei Kategorien von Standardereignissen: Ereignisse, die für alle Steuerelemente definiert sind, und Ereignisse, die nur für fensterorientierte Standard-Steuerelemente gelten.

Standardereignisse für alle Steuerelemente

Die meisten Standardereignisse sind in der Klasse *TControl* definiert. Alle Steuerelemente – fensterorientierte, grafische oder benutzerdefinierte – erben diese Ereignisse. Die folgende Tabelle enthält die Ereignisse, die in allen Steuerelementen vorhanden sind:

OnClick	*OnDragDrop*	*OnEndDrag*	*OnMouseMove*
OnDblClick	*OnDragOver*	*OnMouseDown*	*OnMouseUp*

Zu den Standardereignissen gehören als **protected** deklarierte virtuelle Methoden, die in *TControl* deklariert sind. Ihre Namen entsprechen den Namen der Ereignisse. Beispielsweise rufen die *OnClick*-Ereignisse eine Methode namens *Click* auf, und die *OnEndDrag*-Ereignisse eine Methode namens *DoEndDrag*.

Standardereignisse für Standard-Steuerelemente

Zusätzlich zu den Ereignissen, die allen Steuerelementen gemeinsam sind, besitzen fensterorientierte Standard-Steuerelemente (die von *TWinControl* abgeleitet sind) folgende Ereignisse:

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

Wie die Standardereignisse in *TControl* verfügen auch die fensterorientierten Steuerelement-Ereignisse über entsprechende Methoden.

Ereignisse sichtbar machen

Die Deklarationen der Standardereignisse in *TControl* und *TWinControl* sind als **protected** deklariert. Dies gilt auch für die Methoden, die zu ihnen gehören. Wenn Sie von einer dieser abstrakten Klassen Ableitungen durchführen und deren Ereignisse zur Lauf- oder Entwurfszeit zugreifbar machen wollen, müssen Sie die Ereignisse als **public** oder **published** umdeklarieren.

Bei der Neudeklaration einer Eigenschaft ohne Angabe ihrer Implementierung werden dieselben Implementierungsmethoden übernommen, die Schutzebene wird aber geändert. Sie können deshalb ein Ereignis, das in *TControl* definiert, aber nicht sichtbar gemacht ist, übernehmen und sichtbar machen, indem Sie es als **public** oder **published** deklarieren.

Um beispielsweise eine Komponente zu erstellen, die zur Entwurfszeit das Ereignis *OnClick* sichtbar macht, müßten Sie folgende Klassendeklaration zur Komponente hinzufügen:

```
type
   TMyControl = class(TCustomControl)
   ...
   published
    property OnClick;
   end;
```

Die Standard-Ereignisbehandlung ändern

Wenn Sie die Art und Weise ändern wollen, in der die Komponente auf ein bestimmtes Ereignis antwortet, sind Sie vielleicht versucht, einen entsprechenden Quelltext zu schreiben und dem Ereignis zuzuweisen. Als Anwendungsentwickler würden Sie genau dies tun. Wenn Sie aber eine Komponente erstellen, müssen Sie das Ereignis für die Entwickler bereitstellen, welche die Komponente verwenden.

Hierin liegt der Grund für die als **protected** deklarierten Implementierungsmethoden, die mit jedem Standardereignis verbunden sind. Durch Überschreiben der Implementierungsmethode können Sie die interne Ereignisbehandlung ändern. Durch Aufrufen der geerbten Methode läßt sich dagegen die Standardbehandlung beibehalten, einschließlich des Ereignisses für den Quelltext des Anwendungsentwicklers.

Die Reihenfolge, in der Sie die Methoden aufrufen, ist sehr wichtig. Generell gilt folgendes: Rufen Sie zuerst die geerbte Methode auf, welche die Ausführung der Ereignisbehandlungsroutine des Anwendungsentwicklers ermöglicht. Erst danach dürfen Ihre eigenen Anpassungen ausgeführt werden (bzw. in einigen Fällen deren Ausführung unterbunden werden). Es kann jedoch Situationen geben, in denen Sie Ihren Quelltext vor dem Aufruf der geerbten Methode ausführen wollen. Wenn beispielsweise der geerbte Quelltext in irgendeiner Form vom Status der Komponente abhängig ist und Ihr Quelltext diesen Status ändert, sollten Sie die Änderungen durchführen und danach dem Quelltext des Benutzers erlauben, darauf zu antworten.

Angenommen, Sie schreiben eine Komponente und möchten die Art und Weise ändern, in der sie auf Mausklicks antwortet. Anstatt dem Ereignis *OnClick* eine Routine zuzuweisen (wie es ein Anwendungsentwickler tun würde), überschreiben Sie die als **protected** deklarierte Methode *Click*:

```
procedure click override { Vorwärtsdeklaration }
...
procedure TMyControl.Click;
begin
   inherited Click; { Standardbehandlung, einschließlich Aufruf der Behandlungsroutine  }
... { Hier stehen Ihre Anpassungen }
end;
```

Eigene Ereignisse definieren

Das Definieren vollständig neuer Ereignisse ist relativ unüblich. Wenn jedoch eine Komponente ein Verhalten einführt, das sich völlig von dem jeder anderen Komponente unterscheidet, müssen Sie dafür ein entsprechendes Ereignis definieren.

Folgende Punkte sind bei der Definition eines Ereignisses zu beachten:

- Ereignisse auslösen
- Den Typ der Routine definieren
- Ereignisse deklarieren
- Ereignisse aufrufen

Ereignisse auslösen

Sie müssen wissen, wodurch ein Ereignis ausgelöst wird. Bei einigen Ereignissen liegt die Antwort auf der Hand. Beispielsweise tritt ein Mausereignis ein, wenn der Benutzer die linke Maustaste drückt und Windows eine WM_LBUTTONDOWN-Botschaft an die Anwendung sendet. Nach dem Empfang dieser Botschaft ruft die Komponente ihre Methode *MouseDown* auf, die ihrerseits einen Quelltext aufruft, den der Benutzer mit dem Ereignis *OnMouseDown* verbunden hat.

Es gibt aber auch Ereignisse, die nicht so eindeutig an bestimmte externe Vorgänge geknüpft sind. So verfügt beispielsweise eine Bildlaufleiste über das Ereignis *OnChange*, das von unterschiedlichen Vorgängen ausgelöst werden kann. Dazu gehören Tastenanschläge, Mausklicks und Änderungen in anderen Steuerelementen. Wenn

Sie Ereignisse definieren, müssen Sie sicherstellen, daß die betreffenden Vorgänge die richtigen Ereignisse aufrufen.

Zwei Arten von Ereignissen

Es gibt zwei Arten von Vorgängen, für die Sie Ereignisse für Benutzerinteraktionen und Statusänderungen bereitstellen müssen. Ereignisse für Benutzerinteraktionen werden nahezu immer durch eine Botschaft von Windows ausgelöst. Diese besagt, daß der Benutzer eine Aktion ausgeführt hat, auf die eine Komponente antworten muß. Ereignisse für Statusänderungen können auch mit Botschaften von Windows in Verbindung stehen (z. B. Fokusänderungen oder -aktivierungen). Sie können aber auch eintreten, wenn Eigenschaften oder ein anderer Quelltext geändert werden. Sie haben die vollständige Kontrolle über das Auslösen der von Ihnen definierten Ereignissee. Definieren Sie also die Ereignisse mit Bedacht, so daß Entwickler sie verstehen und verwenden können.

Den Typ der Behandlungsroutine definieren

Nachdem Sie festgelegt haben, wann das Ereignis eintritt, müssen Sie definieren, wie es behandelt werden soll. Mit anderen Worten: Sie müssen den Typ der Ereignisbehandlungsroutine bestimmen. In den meisten Fällen sind die Routinen für Ereignisse, die Sie selbst definieren, einfache Benachrichtigungen oder ereignisspezifische Typen. Es ist auch möglich, von der Routine Informationen zurückgeben zu lassen.

Einfache Benachrichtigungen

Der einzige Zweck eines Benachrichtigungsereignisses besteht in der Mitteilung, daß ein bestimmtes Ereignis eingetreten ist. Wann und wo dies der Fall war, wird nicht angegeben. Benachrichtigungen sind vom Typ *TNotifyEvent*, der nur einen Parameter akzeptiert, nämlich den Sender des Ereignisses. Eine Routine für eine Benachrichtigung weiß von dem Ereignis nur, um welche Art von Ereignis es sich handelt und für welche Komponente das Ereignis eingetreten ist. Beispielsweise sind Klick-Ereignisse Benachrichtigungen. Wenn Sie eine Routine für ein Klick-Ereignis schreiben, wissen Sie nur, daß ein Klick eingetreten ist und auf welche Komponente geklickt wurde.

Eine Benachrichtigung ist ein eingleisiger Prozeß. Es gibt keinen Mechanismus, der eine Rückmeldung bereitstellt oder eine weitere Behandlung einer Benachrichtigung verhindert.

Ereignisspezifische Behandlungsroutinen

In einigen Fällen reicht es nicht aus zu wissen, welches Ereignis eingetreten ist und welche Komponente davon betroffen war. Wenn es sich z. B. um ein Tastendruckereignis handelt, ist es wahrscheinlich, daß die Behandlungsroutine wissen will, welche Taste der Benutzer gedrückt hat. In diesen Fällen benötigen Sie Behandlungsroutinentypen, die Parameter für weitere Informationen enthalten.

Wenn das Ereignis als Antwort auf eine Botschaft generiert wurde, stammen die Parameter, die Sie an die Ereignisbehandlungsroutine übergeben, direkt von den Botschaftsparametern.

Informationen von der Behandlungsroutine zurückliefern

Da alle Ereignisbehandlungsroutinen Prozeduren sind, ist ein Zurückliefern von Informationen nur über einen **var**-Parameter möglich. Die Komponenten können mit Hilfe dieser Informationen festlegen, wie und ob ein Ereignis bearbeitet wird, nachdem die Behandlungsroutine des Benutzers ausgeführt wurde.

Beispielsweise übergeben alle Tastaturereignisse (*OnKeyDown*, *OnKeyUp* und *OnKeyPress*) den Wert der gedrückten Taste per Referenz in einem Parameter namens *Key*. Die Ereignisbehandlungsroutine kann den Parameter *Key* ändern, so daß die Anwendung eine andere als diejenige Taste erkennt, die von dem Ereignis betroffen war. Auf diese Weise erfolgt z. B. die Darstellung von eingegebenen Zeichen in Großbuchstaben.

Ereignisse deklarieren

Sobald Sie den Typ einer Ereignisbehandlungsroutine festgelegt haben, können Sie den Methodenzeiger und die Eigenschaft für das Ereignis deklarieren. Geben Sie dem Ereignis unbedingt einen sinnvollen und beschreibenden Namen, aus dem die Benutzer ersehen können, was das Ereignis ausführt. Versuchen Sie, eine konsistente Namensgebung für ähnliche Eigenschaften in anderen Komponenten durchzuhalten.

Ereignisnamen beginnen mit »On«

Die Namen der meisten Ereignisse in Delphi beginnen mit »On«. Das ist nur eine Konvention, der Compiler verlangt dies nicht. Der Objektinspektor stellt fest, daß eine Eigenschaft ein Ereignis ist, indem er den Typ der Eigenschaft überprüft. Bei allen Methodenzeiger-Eigenschaften wird angenommen, daß es sich um Ereignisse handelt. Sie werden in der Registerkarte *Ereignisse* angezeigt.

Entwickler gehen davon aus, daß Ereignisse in einer alphabetisch sortierten Liste angeordnet sind und Namen tragen, die mit »On« beginnen. Die Verwendung anderer Namen würde nur Verwirrung stiften.

Ereignisse aufrufen

Sie sollten Ereignisaufrufe an einer Position zusammenfassen. Dazu erstellen Sie in der Komponente eine virtuelle Methode, welche die Ereignisbehandlungsroutine der Anwendung (falls vorhanden) aufruft und die Standardbehandlung durchführt.

Die Zusammenfassung aller Ereignisaufrufe an einer Position stellt sicher, daß bei der Ableitung einer neuen Komponente die Ereignisbehandlung durch Überschreiben einer einzigen Methode angepaßt werden kann. Das Durchsuchen des Quelltexts nach Stellen, an denen das Ereignis aufgerufen wird, entfällt bei diesem Vorgehen.

Zwei weitere Punkte sind beim Aufruf des Ereignisses zu bedenken:

- Leere Behandlungsroutinen müssen gültig sein.
- Benutzer können die Standardbehandlung überschreiben.

Leere Behandlungsroutinen müssen gültig sein

Sie dürfen niemals eine Situation schaffen, in der eine leere Behandlungsroutine einen Fehler verursacht. Außerdem darf das korrekte Funktionieren der Komponente nicht von einer bestimmten Antwort abhängig sein, die von der Ereignisbehandlungsroutine der Anwendung stammt.

Eine leere Behandlungsroutine muß dasselbe Resultat wie das Nichtvorhandensein einer derartigen Routine ergeben. Aus diesem Grund sollte der Quelltext für den Aufruf einer Ereignisbehandlungsroutine der Anwendung folgendermaßen aussehen:

```
if Assigned(OnClick) then OnClick(Self);
...  { Standardbehandlung ausführen }
Folgendes ist nicht zulässig:
if Assigned(OnClick) then OnClick(Self)
else { Standardbehandlung ausführen };
```

Benutzer können die Standardbehandlung überschreiben

Für manche Ereignisarten möchten Entwickler unter Umständen die Standardbehandlung ersetzen oder sogar alle Antworten unterdrücken. Damit dies möglich ist, müssen Sie ein Argument per Referenz an die Behandlungsroutine übergeben und auf einen bestimmten Wert hin überprüfen, wenn die Routine zurückkehrt.

Dies steht in Zusammenhang mit der Regel, daß eine leere Behandlungsroutine denselben Effekt wie das Nichtvorhandensein einer derartigen Routine haben muß. Da eine leere Behandlungsroutine die per Referenz übergebenen Argumentwerte nicht ändern würde, findet die Standardbehandlung immer nach dem Aufruf der leeren Routine statt.

Bei der Behandlung von Tastaturereignissen können Anwendungsentwickler beispielsweise die Standardbehandlung der Komponente für den Tastendruck ändern, indem der **var**-Parameter *Key* auf das Zeichen Null (#0) gesetzt wird. Die entsprechende Logik sieht folgendermaßen aus:

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ...  { Ausführen der Standardbehandlung }
```

Der konkrete Quelltext sieht etwas anders aus, weil er Windows-Botschaften behandelt, die Logik ist jedoch dieselbe. Per Voreinstellung ruft die Komponente eine vom Benutzer zugewiesene Behandlungsroutine auf und führt dann deren Standardbehandlung aus. Wenn die Behandlungsroutine des Benutzers den Parameter *Key* mit dem NULL-Zeichen belegt, führt die Komponente die Standardbehandlung nicht aus.

Kapitel

35

Methoden erzeugen

Komponentenmethoden sind Prozeduren und Funktionen, welche in die Struktur einer Klasse integriert sind. Sie unterscheiden sich in nichts von anderen Klassenmethoden. Sie sind Elementfunktionen, die in die Struktur einer Klasse integriert sind. Technisch betrachtet gibt es kaum Einschränkungen für das, was Sie mit den Methoden einer Komponente tun können. In Delphi gelten jedoch einige Standards, die Sie berücksichtigen sollten. Dazu gehören folgende Aspekte:

- Abhängigkeiten vermeiden
- Methoden benennen
- Methoden schützen
- Methoden virtuell machen
- Methoden deklarieren

Grundsätzlich sollten Komponenten nicht zu viele Methoden enthalten, und entsprechend sollte auch die Anwendung nicht zu viele Methoden aufrufen müssen. Sie werden feststellen, daß man oft versucht ist, bestimmte Aktionen als Methode zu implementieren, obwohl die Kapselung in einer Eigenschaft viel sinnvoller wäre. Eigenschaften bieten eine unproblematische Anbindung an die Delphi-Umgebung und sind auch während des Entwurfs im Zugriff.

Abhängigkeiten vermeiden

Bei der Entwicklung von Komponenten sollte immer die Sicht des Programmierers berücksichtigt werden, der später damit arbeitet. Mit dem Einsatz einer Komponente sollten möglichst wenig Nebenbedingungen verbunden sein, das heißt, der Programmierer sollte freien funktionalen und zeitlichen Zugriff auf die Komponente haben. Natürlich wird sich das nie vollständig erreichen lassen.

Die folgende Liste nennt einige Abhängigkeiten, die Sie unbedingt vermeiden sollten:

- Es sollten keine Methoden aufgerufen werden müssen, um die Komponente verwenden zu können.
- Der Aufruf von Methoden sollte nicht an eine bestimmte Reihenfolge gebunden sein.
- Methoden sollten die Komponente nie in einen Zustand versetzen, in dem bestimmte Ereignisse und Methoden ungültig sind.

Sie sollten immer Strategien bereitstellen, mit denen solche unerwünschten Verflechtungen abgefangen werden können. Wenn es beispielsweise eine Methode gibt, die die Komponente in einen Zustand versetzt, in dem eine andere Methode ungültig wird, schreiben Sie eine weitere Methode, die von der Anwendung aufgerufen wird, um diesen unerwünschten Zustand zu korrigieren. Zumindest sollten Sie dafür sorgen, daß eine Exception ausgelöst wird, sobald der Benutzer eine aktuell ungültige Methode aufruft.

Mit anderen Worten, wenn Teile Ihres Quelltextes aufeinander zugreifen, sollten Sie sicherstellen, daß nicht eine inkorrekte Verwendung Ihres Quelltextes zu Problemen führt. So ist beispielsweise die Ausgabe einer Warnmeldung einem Systemfehler selbstverständlich vorzuziehen, falls der Anwender die Abhängigkeiten Ihrer Komponente nicht berücksichtigt.

Methoden benennen

In Delphi gibt es grundsätzlich keinerlei Einschränkungen bezüglich der Benennung von Methoden und Parametern. Allerdings ist es sinnvoll, sich an einige Konventionen zu halten. Schließlich bringt es die komponentenorientierte Architektur mit sich, daß Ihre Komponenten wiederverwendet werden können, und das sollten Sie erleichtern.

Falls Sie daran gewöhnt sind, Quelltext nur für sich oder eine kleine Gruppe von Programmierern zu schreiben, könnte es durchaus sein, daß Sie sich nicht allzu viele Gedanken über die Benennung der einzelnen Bestandteile Ihres Quelltextes machen. Es ist jedoch sehr empfehlenswert, die Bezeichnung Ihrer Methoden eindeutig zu gestalten, da auch Anwender, die mit Ihrem Quelltext (oder sogar mit der Programmierung allgemein) nicht vertraut sind, Ihre Komponenten vielleicht verwenden werden.

Hier sind eine Anregungen, wie Sie die Bezeichnung Ihrer Methoden klar gestalten können:

- Namen sollten aussagekräftig sein (verwenden Sie hierzu Verben).

  Ein Methodenname wie *AusZwischenablageEinfügen* ist wesentlich informativer als einfach nur *Einfügen* oder *AZE*.

- Funktionsnamen sollten den Charakter des Rückgabewertes beschreiben.

Ihnen als Programmierer ist natürlich klar, daß eine Funktion namens *X* die horizontale Position von irgend etwas liefert, trotzdem sollte man deutlicher werden und sich beispielsweise zu *HorizontalePosition* aufraffen.

Überlegen Sie auch immer, ob eine Methode wirklich eine Methode sein muß und nicht vielleicht als Eigenschaft realisiert werden könnte. Ein guter Anhaltspunkt bei dieser Entscheidung ist der Name, den Sie der Methode gegeben haben. Wenn dieser Name ohne Verb auskommt, läßt sich der Effekt der Methode vielleicht auch mit einer Eigenschaft erreichen.

Methoden schützen

Alle Teile von Klassen, einschließlich der Felder, Methoden und Eigenschaften, befinden sich auf einer bestimmten Schutz- oder Sichtbarkeitsebene. Näheres hierzu finden Sie unter »Zugriffssteuerung« auf Seite 32-4. Die Auswahl der geeigneten Sichtbarkeitsebene für eine Methode ist einfach.

Meist werden Sie Methoden in Komponenten als **public** oder **protected** deklarieren. Nur selten ist das Sichtbarkeitsattribut **private** nötig, das bewirkt, daß nicht einmal abgeleitete Komponenten Zugriff erhalten.

Methoden, die als public deklariert sein sollten

Sämtliche Methoden, die Programmierern verfügbar sein sollen, müssen als **public** deklariert werden. Bedenken Sie, daß Methodenaufrufe überwiegend in Ereignisbehandlungsroutinen erfolgen. Vermeiden Sie also Methoden, die übermäßig viel Systemressourcen belegen oder Windows in einen Status versetzen, in dem nicht mehr auf Benutzeraktionen reagiert werden kann.

Hinweis Konstruktoren und Destruktoren sollten immer als **public** deklariert sein.

Methoden, die als protected deklariert sein sollten

Alle Implementierungsmethoden einer Komponente sollten als **protected** deklariert werden, also »geschützt« sein, damit sie in Anwendungen nicht zum falschen Zeitpunkt aufgerufen werden können. Dies gilt auch für Methoden, die nur in abgeleiteten Klassen verwendet werden sollen.

Angenommen, eine Methode setzt voraus, daß bestimmte Daten bereitstehen. Wenn Sie diese Methode als **public** deklarieren, kann es geschehen, daß eine Anwendung die Methode zu früh aufruft. Durch die Deklaration als **protected** wird hingegen erreicht, daß die Anwendung keinen direkten Zugriff mehr auf die Methode hat. Sie können dann andere, als **public** deklarierte Methoden vorsehen, die dafür sorgen, daß die benötigten Daten vorliegen, bevor die als **protected** deklarierte Methode aufgerufen wird.

Methoden zur Eigenschaftsimplementierung sollten als **virtual** und als **protected** deklariert werden. Dadurch wird es Anwendungsentwicklern möglich, die Eigenschaft funktional abzuwandeln. Solche Eigenschaften sind vollständig polymorph. Zugriffsmethoden sollten als **protected** deklariert werden, damit sie nicht versehentlich aufgerufen werden können und unbeabsichtigt eine Eigenschaft ändern.

Abstrakte Methoden

In einer Delphi-Komponente sind Methoden manchmal als abstrakt deklariert. In der VCL gilt dies gewöhnlich für Klassen, deren Name mit »Custom...« beginnt, wie etwa *TCustomGrid*. Solche Klassen sind selbst abstrakt, was bedeutet, daß sie nur als Basis für abgeleitete Klassen dienen sollen.

Sie können zwar eine Instanz einer Klasse erzeugen, die ein abstraktes Element enthält, doch ist dies keineswegs ratsam. Der Aufruf des abstrakten Elements führte nämlich zu einer *EAbstractError*-Exception.

Die Direktive **abstract** kennzeichnet Teile von Klassen, die grundsätzlich erst in abgeleiteten Komponenten definiert und als **published** deklariert werden. Komponentenentwickler müssen abstrakte Elemente dann in abgeleiteten Klassen erst erneut deklarieren, bevor Instanzen der Klasse erzeugt werden können.

Methoden virtuell machen

Methoden werden immer dann **virtuell** gemacht, wenn verschiedene Typen als Reaktion auf denselben Methodenaufruf verschiedenen Quelltext ausführen sollen.

Wenn Sie Komponenten entwickeln, die direkt von anderen Programmierern verwendet werden sollen, werden Sie wahrscheinlich keine Methode virtuell machen. Wenn Sie andererseits abstrakte Komponenten erstellen, von denen andere Komponenten abgeleitet werden, kann es sinnvoll sein, die hinzugefügten Methoden virtuell zu machen. In abgeleiteten Komponenten können die geerbten virtuellen Methoden überschrieben werden.

Methoden deklarieren

Die Deklaration einer Methode in einer Komponente verläuft genauso wie die Deklaration einer Klassenmethode.

So deklarieren Sie eine neue Methode in einer Komponente:

- Die Deklaration muß in die Objekttypdeklaration der Komponente aufgenommen werden.
- Die Implementierung muß im **implementation**-Abschnitt der Komponenten-Unit erfolgen.

Der folgende Quelltext zeigt, wie in einer Komponente zwei neue Methoden definiert werden: eine statische **protected**-Methode und eine virtuelle **public**-Methode:

```
type
   TSampleComponent = class(TControl)
   protected
    procedure MakeBigger;                        { statische Methode als public deklarieren }
   public
    function CalculateArea: Integer; virtual;    { virtuelle Methode als public deklarieren }
   end;
```

```
...
implementation
...
procedure TSampleComponent.MakeBigger; { erste Methode implementieren }
begin
   Height := Height + 5;
   Width := Width + 5;
end;
function TSampleComponent.CalculateArea: Integer; { zweite Methode implementieren }
begin
   Result := Width * Height;
end;
```

Kapitel

36

Grafiken in Komponenten

Windows bietet eine leistungsstarke Grafik-Schnittstelle (GDI = Graphics Device Interface) zur Darstellung geräteunabhängiger Grafiken. Die GDI stellt jedoch zusätzliche Anforderungen an den Programmierer, wie z. B. das Verwalten von Grafik-Ressourcen. Delphi führt alle GDI-Operationen durch, so daß der Benutzer sich auf die produktive Arbeit konzentrieren kann, anstatt nach verlorenen Handles oder nach nicht freigegebenen Ressourcen zu suchen.

Wie bei jedem Teil der Windows-API können Sie GDI-Funktionen direkt aus Ihrer Delphi-Anwendung heraus aufrufen. Sie werden aber wahrscheinlich herausfinden, daß es schneller und einfacher ist, mit der Delphi-Kapselung der Grafikfunktionen zu arbeiten.

Folgende Themen werden in diesem Kapitel behandelt:

- Überblick über die Grafikfunktionen
- Die Zeichenfläche
- Arbeiten mit Bildern
- Offscreen-Bitmaps
- Auf Änderungen reagieren

Überblick über die Grafikfunktionen

Delphi kapselt die Windows-GDI auf verschiedenen Stufen. Am wichtigsten ist es für Sie als Komponentenentwickler, wie Komponenten ihre Bilder auf dem Bildschirm darstellen. Beim direkten Aufruf von GDI-Funktionen brauchen Sie ein Handle für einen Gerätekontext, in den Sie verschiedene Grafikwerkzeuge, wie z. B. Stifte, Pinsel und Schriften, eingegeben haben. Nach der Darstellung Ihrer grafischen Bilder müssen Sie den Gerätekontext wieder in seinen Ausgangszustand zurückversetzen, bevor Sie ihn freigeben.

Anstatt sich ständig mit den Einzelheiten von Grafiken befassen zu müssen, verfügen Sie in Delphi über eine einfache, aber vollständige Schnittstelle: die Eigenschaft *Canvas* Ihrer Komponente. Diese Zeichenfläche sorgt dafür, daß ein gültiger Gerätekontext vorliegt, und gibt ihn frei, wenn Sie ihn nicht benutzen. Analog verfügt die Zeichenfläche über ihre eigenen Eigenschaften, die den aktuellen Stift, Pinsel und die aktuelle Schrift bestimmen.

Die Zeichenfläche verwaltet all diese Ressourcen für Sie. Sie müssen sich also nicht selbst damit befassen, Dinge wie Stift-Handles zu erzeugen, auszuwählen und freizugeben. Sie müssen der Zeichenfläche nur mitteilen, welche Art Stift sie verwenden soll, und sie erledigt alles übrige.

Einer der Vorteile bei der Verwaltung der Grafik-Ressourcen durch Delphi besteht darin, daß Delphi Ressourcen für den späteren Gebrauch zwischenspeichern kann, wodurch Wiederholungsvorgänge beschleunigt werden. Wenn Sie beispielsweise mit einem Programm arbeiten, das eine bestimmte Art von Stift immer wieder einrichtet, verwendet und freigibt, müssen Sie diese Schritte nicht ständig wiederholen. Da Delphi Grafik-Ressourcen zwischenspeichert, befindet sich ein Tool, das Sie wiederholt verwenden, mit großer Wahrscheinlichkeit noch im Zwischenspeicher. Anstatt also ein Tool neu zu erzeugen, wird eine bereits bestehende Version davon verwendet.

Ein Beispiel dafür wäre eine Anwendung mit Dutzenden von geöffneten Formularen und Hunderten von Steuerelementen. Jedes dieser Steuerelemente kann eine oder mehrere *TFont*-Eigenschaften haben. Obwohl sich daraus hunderte oder tausende von *TFont*-Objektinstanzen ergeben könnten, würde die Anwendung dank des Schrift-Zwischenspeichers der VCL mit nur zwei oder drei Schrift-Handles auskommen.

Es folgen zwei Beispiele, die zeigen, wie einfach in Delphi Grafikfunktionen realisiert werden können. Im ersten Beispiel werden Standard-GDI-Funktionen verwendet, um in einer mit ObjectWindows geschriebenen Anwendung eine gelbe Ellipse mit blauer Umrißlinie in einem Fenster zu zeichnen. Im zweiten Beispiel wird eine Zeichenfläche verwendet, um die gleiche Ellipse in einer mit Delphi geschriebenen Anwendung zu zeichnen.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
   PenHandle, OldPenHandle: HPEN;
   BrushHandle, OldBrushHandle: HBRUSH;
begin
   PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255)); { Blauen Stift erzeugen }
   OldPenHandle := SelectObject(PaintDC, PenHandle); { Gerätekontext anweisen, diesen Stift
           zu verwenden }
   BrushHandle := CreateSolidBrush(RGB(255, 255, 0)); { Gelben Pinsel erzeugen }
   OldBrushHandle := SelectObject(PaintDC, BrushHandle); { Gerätekontext anweisen, den
           Pinsel zu verwendenh }
   Ellipse(HDC, 10, 10, 50, 50); { Ellipse zeichnen }
   SelectObject(OldBrushHandle); { Vorherigen Pinsel wiederherstellen }
   DeleteObject(BrushHandle); { Gelben Pinsel freigeben }
   SelectObject(OldPenHandle); { Vorherigen Stift wiederherstellen }
   DeleteObject(PenHandle); { Blauen Stift freigeben }
end;
procedure TForm1.FormPaint(Sender: TObject);
```

```
begin
   with Canvas do
   begin
    Pen.Color := clBlue; { Stift blau färben }
    Brush.Color := clYellow; { Pinsel gelb färben }
    Ellipse(10, 10, 50, 50); { Ellpise zeichnen  }
   end;
end;
```

Die Zeichenfläche

Die Zeichenflächen-Klasse kapselt Windows-Grafiken auf verschiedenen Stufen. Sie bietet High-Level-Funktionen zum Zeichnen von einzelnen Linien, Formen und Text, Eigenschaften zur Handhabung der Funktionen der Zeichenfläche und Low-Level-Zugriff auf die Windows-GDI.

Tabelle 35.1 gibt einen Überblick über den Funktionsumfang einer Zeichenfläche.

Tabelle 36.1 Funktionsumfang von Zeichenflächen

Stufe	Operation	Werkzeuge
High	Zeichnen von Linien und Formen	Methoden wie z. B. *MoveTo*, *LineTo*, *Rectangle* und *Ellipse*
	Anzeigen und Dimensionieren von Text	Methoden *TextOut*, *TextHeight*, *TextWidth* und *TextRect*
	Ausfüllen von Flächen	Methoden *FillRect* und *FloodFill*
Zwischenstufe	Text und Grafiken anpassen	Eigenschaften *Pen*, *Brush* und *Font*
	Einstellen der Pixel	Eigenschaft *Pixeln*
	Kopieren und Zusammenfügen von Bildern	Methoden *Draw*, *StretchDraw*, *BrushCopy* und *CopyRect*, Eigenschaft *CopyMode*
Low	Aufrufen von Windows-GDI-Funktionen	Eigenschaft *Handle*

Nähere Informationen über Zeichenflächen-Klassen und ihre Methoden und Eigenschaften finden Sie in der Online-Hilfe.

Mit Bildern arbeiten

Der größte Teil der Arbeiten mit Grafik, die Sie in Delphi ausführen, beschränkt sich darauf, direkt auf den Zeichenflächen von Komponenten und Formularen zu zeichnen. Delphi bietet auch die Möglichkeit zur Bearbeitung von selbständigen Grafikbildern, wie z. B. Bitmaps, Metadateien und Symbolen, einschließlich der automatischen Verwaltung von Paletten.

Bei der Arbeit mit Bildern in Delphi sind drei wichtige Aspekte zu unterscheiden:

- Arbeiten mit einem Bild, einer Grafik oder einer Zeichenfläche.

- Laden und Speichern von Grafiken.
- Paletten.

Mit einem Bild, einer Grafik oder einer Zeichenfläche arbeiten

Delphi umfaßt drei Arten von Klassen für die Arbeit mit Grafiken:

- Eine *Zeichenfläche* stellt eine Bitmap-Zeichenoberfläche in einem Formular, einem grafischen Steuerelement, einem Drucker oder einem Bitmap dar. Eine Zeichenfläche ist immer eine Eigenschaft von etwas anderem und nie eine Klasse für sich.
- Eine *Grafik* stellt ein grafisches Bild dar, wie man es normalerweise in einer Datei oder Ressource findet, wie z. B. ein Bitmap, ein Symbol oder eine Metadatei. Delphi definiert die Klassen *TBitmap*, *TIcon* und *TMetafile*, die alle von einem generischen *TGraphic*-Objekt abgeleitet sind. Sie können auch Ihre eigenen Grafik-Klassen definieren. Durch die Definition einer universellen Schnittstelle mit minimaler Standardisierung bietet *TGraphic* einen Mechanismus für die Behandlung der verschiedensten Grafiken.
- Ein *Bild* ist ein Container für eine Grafik, was bedeutet, daß es jede der Grafik-Klassen enthalten kann. Ein Element des Typs *TPicture* kann somit ein Bitmap, ein Symbol, eine Metadatei oder einen benutzerdefinierten Grafiktyp enthalten, und eine Anwendung kann durch die Bildklasse auf alle in der gleichen Weise zugreifen. Beispielsweise beinhaltet das Bild-Steuerelement *TImage* eine Eigenschaft namens *Picture* des Typs *TPicture*, die das Steuerelement befähigt, Bilder verschiedensten Typs anzuzeigen.

Denken Sie daran, daß zu einer Bildklasse immer eine Grafik gehört, und daß zu einer Grafik eine Zeichenfläche gehören könnte (die einzige Standard-Grafik, zu der ein Zeichenfläche gehört, ist *TBitmap*). Normalerweise arbeiten Sie in einem Bild nur mit den Teilen der Grafik-Klasse, die über *TPicture* zugänglich sind. Falls Sie auf die spezifischen Eigenschaften der Grafik-Klasse selbst zugreifen möchten, benutzen Sie die Eigenschaft *Graphic* des Bildes.

Grafiken laden und speichern

Alle Bilder und Grafiken in Delphi können ihre Bilddaten aus Dateien laden und dort (oder in anderen Dateien) auch wieder speichern. Sie können die Bilddaten eines Bildes zu jedem Zeitpunkt laden oder speichern.

Das Laden eines Bildes aus einer Datei erfolgt durch Aufrufen der Methode *LoadFromFile* des Bildes.

Das Speichern eines Bildes in einer Datei erfolgt durch Aufrufen der Methode *SaveToFile*.

LoadFromFile und *SaveToFile* benötigen als einzigen Parameter einen Dateinamen. *LoadFromFile* bestimmt anhand der Dateinamenserweiterung, welche Art Grafikobjekt erzeugt und geladen wird. *SaveToFile* speichert in jedem Dateiformat, das sich für den Typ des zu speichernden Grafikobjekts eignet.

Um beispielsweise ein Bitmap in ein Bild zu laden, übergeben Sie der Methode *LoadFromFile* des Bildes den Namen einer Bitmap-Datei:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
   Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

Das Bild erkennt BMP als Standarderweiterung für Bitmap-Dateien und erzeugt dessen Grafik als *TBitmap*-Objekt. Dann ruft es die Methode *LoadFromFile* für die Grafik auf. Entsprechend dem Typ der Grafik wird das Bild als Bitmap aus der Datei geladen.

Paletten

Wenn Delphi in einem Gerätekontext läuft (normalerweise mit 256 Farben), in dem Paletten unterstützt werden, steuert Delphi automatisch den Aufbau der Palette. Das heißt, wenn ein Steuerelement mit Palette vorliegt, können Sie zwei von *TControl* geerbte Methoden benutzen, um zu steuern, wie Windows diese Palette einrichtet.

In diesem Zusammenhang sind zwei Aspekte von Bedeutung:

- Zuordnung einer Palette zu einem Steuerelement
- Reaktion auf Änderungen der Palette

Die meisten Steuerelemente brauchen keine Palette. Bei Steuerelementen, die grafische Bilder mit vielen verschiedenen Farben enthalten, könnte sich jedoch die Notwendigkeit ergeben, in Dialog mit Windows und dem Bildschirmtreiber zu treten, um das richtige Erscheinungsbild des Steuerelements zu gewährleisten. In Windows wird dieser Vorgang als Erzeugung oder Aufbau von Paletten bezeichnet.

Mit dem Erzeugen von Paletten wird gewährleistet, daß das vorderste Fenster unter voller Ausnutzung seiner Palette erscheint und daß Fenster im Hintergrund so gut wie möglich entsprechend ihrer Palette dargestellt werden. Für alle Farben, die nicht in der Palette definiert sind, wird ein möglichst ähnlicher Ersatz gesucht. In Windows müssen Paletten ständig neu aufgebaut werden, etwa wenn ein Fenster aus dem Hintergrund nach vorn gebracht wird.

Hinweis Delphi selbst verfügt nicht über spezifische Unterstützungsmechanismen für die Erzeugung oder Verwaltung von Paletten, außer für Bitmaps. Aber wenn Sie über ein Paletten-Handle verfügen, können die Delphi-Steuerelemente die Verwaltung übernehmen.

Eine Palette für ein Steuerelement definieren

Wenn eine Palette für ein Steuerelement zusammengestellt werden soll, muß die Methode *GetPalette* so überschrieben werden, daß Sie das Handle der Palette zurückgibt.

Während der Definition einer Palette für ein Steuerelement laufen in der Anwendung folgende Vorgänge ab:

- Die Anwendung wird benachrichtigt, daß die Palette aufgebaut werden muß.

- Die Palette wird für die Erzeugung vorgemerkt.

Auf Änderungen der Palette reagieren

Wenn ein Steuerelement eine Palette durch Überschreiben von *GetPalette* definiert, antwortet Delphi automatisch auf Palettenbotschaften von Windows. Die Methode, welche die Palettenbotschaften bearbeitet, heißt *PaletteChanged*.

PaletteChanged hat in erster Linie die Aufgabe, zu bestimmen, ob die Palette des Steuerelements im Vordergrund oder im Hintergrund erzeugt werden soll. In Windows erfolgt dieses Erzeugen von Paletten, indem das oberste Fenster eine Vordergrundpalette erhält und die anderen Fenster mit Hintergrundpaletten erscheinen. Delphi geht einen Schritt weiter, indem Paletten für Steuerelemente in der Tabulatorreihenfolge erstellt werden, die innerhalb des Fensters gilt. Dieses Standardverhalten müßte nur dann überschrieben werden, wenn Sie wünschen, daß ein Steuerelement, das in der Tabulatorreihenfolge nicht an erster Stelle steht, die Vordergrundpalette erhält.

Offscreen-Bitmaps

Beim Darstellen von komplexen grafischen Bildern ist es in der Windows-Programmierung üblich, zuerst ein Offscreen-Bitmap anzulegen, das Bild darin zu zeichnen und dann das vollständige Bild aus dem Bitmap an den endgültigen Onscreen-Bestimmungsort zu kopieren. Mit Offscreen-Bildern läßt sich das Flimmern reduzieren, das durch wiederholtes Zeichnen direkt auf dem Bildschirm hervorgerufen wird.

Die in Delphi vorhandene Bitmap-Klasse, die Bitmap-Bilder in Ressourcen und Dateien darstellt, ist auch für Offscreen-Bilder geeignet.

Beim Arbeiten mit Offscreen-Bitmaps sind zwei Hauptaspekte zu unterscheiden:

- Offscreen-Bitmaps erzeugen und verwalten
- Bitmaps kopieren.

Offscreen-Bitmaps erzeugen und verwalten

Bei der Erzeugung von komplexen grafischen Bildern sollte man es vermeiden, diese direkt auf dem Bildschirm darzustellen. Anstatt auf der Zeichenfläche für ein Formular oder Steuerelement zu zeichnen, können Sie ein Bitmap-Objekt im Speicher erstellen, auf dessen virtueller Zeichenfläche zeichnen und dann das vollständige Bild auf die Onscreen-Zeichenfläche kopieren.

Die häufigste Verwendung findet ein Offscreen-Bitmap in der Methode *Paint* eines Grafik-Steuerelements. Wie bei jedem temporären Objekt sollte das Bitmap mit einem **try..finally**-Block geschützt werden:

```
type
   TFancyControl = class(TGraphicControl)
   protected
    procedure Paint; override; { Methode Paint überschreiben }
   end;
```

```
procedure TFancyControl.Paint;
var
   Bitmap: TBitmap; { Temporäre Variable für das Offscreen-Bitmap }
begin
   Bitmap := TBitmap.Create; { Bitmap-Objekt erzeugen }
   try
    { in Bitmap zeichnen }
    { Ergebnis in Zeichenfläche des Steuerelements kopieren  }
   finally
    Bitmap.Free; { Bitmap-Objekt freigeben }
   end;
end;
```

Bitmaps kopieren

Delphi bietet vier verschiedene Möglichkeiten, um Bilder von einer Zeichenfläche in eine andere zu kopieren. Je nach dem von Ihnen gewünschten Vorgang rufen Sie unterschiedliche Methoden auf.

Tabelle 35.2 enthält die Methoden zum Kopieren von Bildern in Zeichenflächen.

Tabelle 36.2 Methoden zum Kopieren von Bildern

Vorgang	Geeignete Methode
Komplette Grafik kopieren.	*Draw*
Grafik kopieren und in der Größe verändern.	*StretchDraw*
Teil einer Zeichenfläche kopieren.	*CopyRect*
Ein Bitmap mit Rasteroperationen kopieren.	*BrushCopy*

Auf Änderungen reagieren

In allen grafischen Objekten, einschließlich Zeichenflächen und deren eigenen Objekten (Stifte, Pinsel und Schriften), sind Ereignisse vorgesehen, um auf Änderungen in dem Objekt zu reagieren. Mit Hilfe dieser Ereignisse können Sie erreichen, daß Komponenten (oder die Anwendungen, die damit arbeiten) auf die Änderungen reagieren, indem Sie deren Bilder neu zeichnen.

Die Reaktion auf Änderungen an Grafikobjekten ist von besonderer Wichtigkeit, wenn Sie sie als Teil der Entwurfsmodus-Schnittstelle ihrer Komponenten als **published** deklarieren. Die einzige Möglichkeit, dafür zu sorgen, daß das Aussehen der Komponente während des Entwurfs den Eigenschaftswerten im Objektinspektor entspricht, besteht darin, auf Änderungen an den Objekten zu reagieren.

Hierfür weisen Sie dem Ereignis *OnChange* der Klasse eine Behandlungsroutine zu.

Die Komponente *TShape* deklariert Eigenschaften als **published**, die den Stift und den Pinsel repräsentieren, die zum Zeichnen geometrischer Figuren verwendet werden. Der Konstruktor der Komponente weist jeweils dem Ereignis *OnChange* eine Methode zu, die dafür sorgt, daß die Komponente jedesmal die Bilddarstellung erneuert, wenn sich der Stift oder der Pinsel ändert:

```
type
   TShape = class(TGraphicControl)
   public
    procedure StyleChanged(Sender: TObject);
   end;
...
implementation
...
constructor TShape.Create(AOwner: TComponent);
begin
   inherited Create(AOwner); { Immer den geerbten Konstruktor aufrufen! }
   Width := 65;
   Height := 65;
   FPen := TPen.Create; { Stift erzeugen  }
   FPen.OnChange := StyleChanged; { Ereignis OnChange mit Methode verknüpfen }
   FBrush := TBrush.Create; { Pinsel erzeugen }
   FBrush.OnChange := StyleChanged; { Ereignis OnChange mit Methode verknüpfen  }
end;
procedure TShape.StyleChanged(Sender: TObject);
begin
   Invalidate(); { Komponente löschen und erneut darstellen }
end;
```

Kapitel

37

Botschaftsbehandlung

Einer der wichtigsten Aspekte in der traditionellen Windows-Programmierung betrifft die Behandlung von Botschaften, die Windows an Anwendungen sendet. Delphi nimmt Ihnen die Behandlung der meisten dieser Botschaften ab. Es gibt aber auch Situationen, in denen Sie selbst eine Reaktion auf Botschaften implementieren müssen, die nicht von Delphi behandelt werden oder die Sie selbst erzeugt haben.

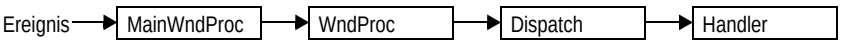
Dieses Kapitel befaßt sich mit drei Themenbereichen, die für die Arbeit mit Botschaften wichtig sind:

- Das Botschaftsbehandlungssystem
- Die Behandlung von Botschaften ändern
- Neue Routinen zur Botschaftsbehandlung erstellen

Das Botschaftsbehandlungssystem

Alle Delphi-Klassen verfügen über einen integrierten Mechanismus zur Verarbeitung von Botschaften, die sogenannten Botschaftsbehandlungsmethoden (auch Botschaftsbehandlungsroutinen genannt). Der Botschaftsbehandlung liegt folgendes Prinzip zugrunde: Die Klasse empfängt Botschaften und ruft – in Abhängigkeit von der Art der Botschaft – eine bestimmte Methode aus einer Methodengruppe auf, die vorher für die Botschaftsbehandlung definiert wurde. Falls für eine Botschaft keine entsprechende Behandlungsmethode existiert, erfolgt ein Aufruf der Standard-Behandlungsroutine.

Die folgende Abbildung illustriert das System der Botschaftsverteilung.



In der Bibliothek visueller Komponenten (VCL) ist ein Botschaftsverteilungssystem definiert, das alle Windows-Botschaften (einschließlich der benutzerdefinierten Botschaften), die für eine bestimmte Klasse festgelegt sind, in Methodenaufrufe über-

setzt. Normalerweise muß dieser Mechanismus zur Botschaftsverteilung nie geändert werden. Ihre Aufgabe besteht ausschließlich in der Erstellung der Botschaftsbehandlungsmethoden. Im Abschnitt »Eine neue Botschaftsbehandlungsmethode deklarieren« auf Seite 37-7 finden Sie weitere Informationen zu diesem Thema.

Was enthält eine Windows-Botschaft?

Eine Windows-Botschaft ist ein Record, der verschiedene Felder enthält. Das wichtigste Feld enthält einen Wert im Integer-Format, der die Botschaft identifiziert. Für alle Windows-Botschaften sind in der Unit *Messages* entsprechende Bezeichner deklariert. Neben diesem Identifikationsfeld liefert eine Botschaft zwei weitere wichtige Informationen in Form zweier Parameterfelder und eines Ergebnisfeldes.

Ein Parameter enthält 16 Bits, der andere 32 Bits. In Windows-Quelltext werden diese Werte häufig als *wParam* bzw. *lParam* bezeichnet, wobei *wParam* für »Word-Parameter« und *lParam* für »Long-Parameter« steht. Diese Parameter enthalten oft mehrere Informationsbestandteile. Beispielsweise werden Sie Referenzen auf Namen wie z. B. *lParamHi* antreffen (hier ist das höherwertige Word im Long-Parameter gemeint).

Ursprünglich mußten sich Windows-Programmierer merken, was in den einzelnen Parametern enthalten ist (oder in der Windows-API nachschlagen). Seit einiger Zeit verwendet Microsoft aber Namen für die Parameter. Diese sogenannte »Botschaftszerlegung« erleichtert das Erkennen der Informationen, die mit einer Botschaft verbunden sind. Die Parameter für die Botschaft WM_KEYDOWN heißen nun *nVirtKey* und *lKeyData*. Diese Namen haben mehr Informationsgehalt als *wParam* und *lParam*.

Für jeden Botschaftstyp definiert Delphi einen Record-Typ, der wiederum für jeden Parameter einen mnemonischen Namen definiert. Mausbotschaften übergeben beispielsweise die X- und Y-Koordinaten des Mausereignisses im Long-Parameter. Dabei ist eine Koordinate im höherwertigen Word enthalten, die andere im niederwertigen Word. Bei Verwendung der Mausbotschaftsstruktur müssen Sie sich nicht um den Inhalt der einzelnen Word kümmern, da Sie die Parameter über ihre Namen referenzieren können. Diese lauten *XPos* (für *lParamLo*) und *YPos* (für *lParamHi*).

Botschaften verteilen

Wenn eine Anwendung ein Fenster erzeugt, registriert sie eine Fensterprozedur im Windows-Kernel. Diese Fensterprozedur ist die Routine, die Botschaften für das Fenster behandelt. Üblicherweise enthält die Fensterprozedur eine umfangreiche **case**-Anweisung mit einem Eintrag für jede Botschaft, die das Fenster behandeln muß. Der Begriff »Fenster« bezieht sich hier auf alles, was sich auf dem Bildschirm befindet (alle Fenster, alle Steuerelemente usw.). Bei jeder Erstellung eines neuen Fenstertyps müssen Sie auch eine vollständige Fensterprozedur definieren.

Delphi vereinfacht die Botschaftsverteilung mit den folgenden Mechanismen:

- Jede Komponente erbt ein vollständiges System zur Botschaftsverteilung.
- Das Verteilungssystem stellt eine Standardbehandlung für die Botschaften bereit. Sie brauchen deshalb Behandlungsroutinen nur für diejenigen Botschaften zu definieren, die eine spezielle Antwort erfordern.

- Sie können kleinere Abschnitte der Botschaftsbehandlungsroutine ändern, die meisten Verarbeitungsschritte aber den geerbten Methoden überlassen.

Der größte Vorteil dieses Botschaftsbehandlungssystems liegt in der Tatsache, daß jede Botschaft jederzeit an jede beliebige Komponente gesendet werden kann. Wenn die Komponente keine Behandlungsroutine für die Botschaft bereitstellt, wird die Standard-Botschaftsbehandlung aktiviert (im Normalfall heißt dies, die Botschaft wird ignoriert).

Den Botschaftsfluß verfolgen

Delphi registriert für jeden Komponententyp in einer Anwendung eine Methode mit dem Namen *MainWndProc* als Fensterprozedur. *MainWndProc* enthält einen Block zur Exception-Behandlung, in dem die Botschaftsstruktur von Windows an eine virtuelle Methode namens *WndProc* übergeben wird. Die Behandlung von Exceptions erfolgt durch einen Aufruf der Methode *HandleException* der Anwendungsklasse.

MainWndProc ist eine nicht-virtuelle Methode, in der keine spezielle Behandlung für bestimmte Botschaften definiert ist. Anpassungen müssen in der Methode *WndProc* vorgenommen werden, da jeder Komponententyp die Methode für seine speziellen Bedürfnisse überschreiben kann.

WndProc-Methoden testen alle Bedingungen, die sich auf ihre Verarbeitung auswirken könnten. Sie sind damit in der Lage, unerwünschte Botschaften »abzufangen«. Beispielsweise ignorieren Komponenten, die gerade mit der Maus gezogen werden, alle Tastaturereignisse. Die *WndProc*-Methode von *TWinControl* gibt Tastaturereignisse also nur dann weiter, wenn die Komponente nicht gezogen wird. Zum Schluß ruft *WndProc* immer die Methode *Dispatch* auf. *Dispatch* ist eine nicht-virtuelle, von *TObject* geerbte Methode, die feststellt, welche Methode zur Behandlung der Botschaft aufgerufen werden muß.

Dispatch ermittelt mit Hilfe des Feldes *Msg* in der Botschaftsstruktur, wie eine bestimmte Botschaft verteilt werden muß. Wenn eine Komponente eine Behandlungsroutine für die Botschaft definiert, wird diese von *Dispatch* aufgerufen. Andernfalls ruft *Dispatch* die Methode *DefaultHandler* auf.

Die Behandlung von Botschaften ändern

Bevor Sie die Botschaftsbehandlung für Ihre Komponenten modifizieren, müssen Sie sich über die gewünschte Wirkung genau im klaren sein. Delphi übersetzt die meisten Windows-Botschaften in Ereignisse, die sowohl vom Komponentenentwickler als auch vom Benutzer der Komponente behandelt werden können. Es kann deshalb ratsam sein, anstelle der Botschaftsbehandlung das Ereignisbehandlungsverhalten zu ändern.

Zur Änderung der Botschaftsbehandlung überschreiben Sie die Botschaftsbehandlungsmethode. In bestimmten Situationen kann es auch angebracht sein, eine Botschaft abzufangen, um die Komponente an der Behandlung der Botschaft zu hindern.

Die Behandlungsmethode überschreiben

Sie ändern die Art und Weise, in der eine Komponente auf eine bestimmte Botschaft reagiert, indem Sie die entsprechende Botschaftsbehandlungsmethode überschreiben. Für Botschaften, die nicht von der Komponente behandelt werden, muß eine neue Botschaftsbehandlungsmethode deklariert werden.

Zum Überschreiben einer Botschaftsbehandlungsmethode deklarieren Sie eine neue Methode in der Komponente, die denselben Botschaftsindex wie die zu überschreibende Methode aufweist. Verwenden Sie nicht die Direktive **override**. Die Verwendung der Direktive **message** zusammen mit einem Botschaftsindex ist obligatorisch.

Der Name der Methode und der Typ des einzelnen **var**-Parameters müssen nicht mit der überschriebenen Methode übereinstimmen. Relevant ist nur der Botschaftsindex. Damit die Zusammenhänge ersichtlich sind, sollten aber bei der Benennung der Botschaftsbehandlungsmethoden die Namen der behandelten Botschaften benutzt werden.

Wenn Sie beispielsweise die Behandlungsroutine für die WM_PAINT-Botschaft einer Komponente überschreiben, deklarieren Sie dazu die Methode *WMPaint* neu:

```
type
   TMyComponent = class(...)
   ...
   procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

Botschaftsparameter verwenden

Nach dem Eintritt in eine Botschaftsbehandlungsmethode hat die Komponente Zugriff auf alle Parameter der Botschaftsstruktur. Da an die Botschaftsbehandlungsroutine ein **var**-Parameter übergeben wird, kann die Routine den Parameter bei Bedarf ändern. Der einzige Parameter, bei dem häufig eine Änderung auftritt, ist das Ergebnisfeld der Botschaft. Es enthält den Wert, der vom *SendMessage*-Aufruf zurückgeliefert wird, durch den die Botschaft gesendet wurde.

Der Typ des Parameters *Message* in der Botschaftsbehandlungsmethode ist von der behandelten Botschaft abhängig. In der Dokumentation der Windows-Botschaften sind die Namen und Bedeutungen der einzelnen Parameter erläutert. Wenn Sie aus irgendeinem Grund gezwungen sind, die Botschaftsparameter über ihre alten Namen zu referenzieren (*wParam*, *lParam* usw.), können Sie *Message* in den generischen Typ *TMessage* umwandeln, der diese Parameternamen benutzt.

Botschaften abfangen

Es gibt Situationen, in denen Botschaften von den Komponenten ignoriert werden sollen. Die Komponente darf dann die Botschaft nicht an ihre Behandlungsroutine weiterleiten. Sie können eine Botschaft abfangen, indem Sie die virtuelle Methode *WndProc* überschreiben.

Die Methode *WndProc* überprüft Botschaften, bevor sie diese an die Methode *Dispatch* übergibt. *Dispatch* bestimmt dann, welche Methode die Behandlung der Botschaft übernimmt. Durch Überschreiben von *WndProc* können Sie der Komponente die Möglichkeit geben, Botschaften vor der Weiterleitung an die Behandlungsroutine auszufiltern. So könnte beispielsweise für ein Steuerelement, das von *TWinControl* abgeleitet ist, *WndProc* folgendermaßen überschrieben werden:

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
   { Feststellen, ob die Behandlung fortgesetzt werden soll }
   inherited WndProc(Message);
end;
```

Die Komponente *TControl* definiert komplette Gruppen von Mausbotschaften, die ausgefiltert werden, wenn der Benutzer Steuerelemente zieht und ablegt. Durch das Überschreiben von *WndProc* können Sie folgendes erreichen:

- Anstatt Behandlungsroutinen für jede einzelne Botschaft festzulegen, werden Botschaftsgruppen ausgefiltert.
- Die Weiterleitung der Botschaft wird unterbunden, d.h., es wird keine Behandlungsroutine aufgerufen.

Nachfolgend sehen Sie einen Abschnitt der *WndProc*-Methode für *TControl*:

```
procedure TControl.WndProc(var Message: TMessage);
begin
   ...
   if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then { Drag-Vorgänge speziell behandeln }
      DragMouseMsg(TWMMouse(Message))
    else
      ... { Alle anderen Aktionen normal behandeln }
    end;
... { Ansonsten normale Verarbeitung  }
end;
```

Neue Routinen zur Botschaftsbehandlung erstellen

Delphi stellt Behandlungsroutinen für die gängigsten Windows-Botschaften bereit. Meist werden Sie deshalb nur für eigene Botschaften eine neue Botschaftsbehandlungsroutine definieren müssen. Die Arbeit mit benutzerdefinierten Botschaften umfaßt zwei Hauptaufgaben:

- Eigene Botschaften definieren
- Eine neue Botschaftsbehandlungsmethode deklarieren

Eigene Botschaften definieren

Einige der Standardkomponenten definieren Botschaften für den internen Gebrauch. Botschaften werden meist zur Anzeige von Statusänderungen und zur Übermittlung

von Informationen definiert, die nicht durch die Standardbotschaften von Windows abgedeckt sind.

Die Definition einer Botschaft umfaßt zwei Schritte:

1 Deklaration eines Botschaftsbezeichners.

2 Deklaration eines Botschafts-Record-Typs.

Einen Botschaftsbezeichner deklarieren

Ein Botschaftsbezeichner ist eine Konstante im Integer-Format. Windows reserviert die Botschaften unterhalb von 1024 für den eigenen Gebrauch. Sie können deshalb erst über dieser Grenze mit der Deklaration Ihrer Botschaften beginnen.

Die Konstante *WM_APP* repräsentiert die Anfangsnummer für benutzerdefinierte Botschaften. Definieren Sie ihre Botschaftsbezeichner beginnend bei *WM_APP*.

Einige Standard-Steuerelemente von Windows benutzen Botschaften, die im Bereich der benutzerdefinierten Botschaften liegen. Dazu gehören Listenfelder, Kombinationsfelder, Eingabefelder und Befehlsschaltflächen. Wenn Sie eine Komponente aus einem dieser Elemente ableiten und dafür eine neue Botschaft definieren wollen, müssen Sie in der Unit *Messages* überprüfen, welche Botschaften Windows bereits für das Steuerelement definiert hat.

Der folgende Quelltextauschnitt zeigt zwei benutzerdefinierte Botschaften:

```
const
   WM_MYFIRSTMESSAGE = WM_APP + 400;
   WM_MYSECONDMESSAGE = WM_APP + 401;
```

Einen Botschafts-Record-Typ deklarieren

Damit Sie den Parametern Ihrer Botschaft aussagekräftige Namen geben können, müssen Sie für die Botschaft einen Botschafts-Record-Typ deklarieren. Der Botschafts-Record ist der Typ des an die Botschaftsbehandlungsmethode übergebenen Parameters. Wenn Sie die Botschaftsparameter nicht verwenden oder die alte Parameternotation benutzen (*wParam*, *lParam* usw.), können Sie den Standard-Botschafts-Record *TMessage* einsetzen.

Halten Sie sich bei der Deklaration eines Botschafts-Record-Typs an folgende Konventionen:

1 Benennen Sie den Typ nach der Botschaft, und stellen Sie ihm den Buchstaben T voran.

2 Weisen Sie dem ersten Feld im Record den Namen *Msg* und den Typ *TMsgParam* zu.

3 Definieren Sie die beiden nächsten Bytes so, daß sie dem Parameter *Word* entsprechen, und die beiden darauffolgenden Bytes als unbenutzt,

   oder

   definieren Sie die nächsten vier Bytes so, daß sie dem Parameter *Longint* entsprechen.

4 Fügen Sie ein letztes Feld namens *Result* des Typs *Longint* hinzu.

Nachstehend sehen Sie den Botschafts-Record für alle Mausereignisse namens *TWMMouse*. Er definiert mit Hilfe einer Variante zwei Namensgruppen für dieselben Parameter.

```
type
   TWMMouse = record
    Msg: TMsgParam; ( Zuerst kommt die Botschafts-ID )
    Keys: Word; ( Dies ist der wParam )
    case Integer of ( lParam kann auf zwei Arten interpretiert werden )
      0: {
         XPos: Integer; ( Entweder als X-/Y-Koordinaten ...)
         YPos: Integer);
      1: {
         Pos: TPoint; ( ... oder als Einzelpunkt  )
         Result: Longint); ( Zuletzt kommt das Ergebnisfeld  )
end;
```

Eine neue Botschaftsbehandlungsmethode deklarieren

Es gibt zwei Situationen, in denen die Deklaration neuer Botschaftsbehandlungsmethoden erforderlich ist:

- Ihre Komponente muß eine Windows-Botschaft verarbeiten, die von den Standardkomponenten noch nicht behandelt wird.
- Sie haben eine eigene Botschaft für die interne Verwendung in Ihren Komponenten definiert.

Zur Deklaration einer Botschaftsbehandlungsmethode führen Sie folgende Arbeitsschritte aus:

1 Deklarieren Sie die Methode im **protected**-Abschnitt der Klassendeklaration der Komponente.

2 Machen Sie aus der Methode eine Prozedur.

3 Benennen Sie die Methode nach der Botschaft, die sie behandeln soll. Lassen Sie dabei aber den Unterstrich weg.

4 Übergeben Sie einen einzelnen **var**-Parameter namens *Message*. Dieser muß dem Typ des Botschafts-Records entsprechen.

5 Fügen Sie Quelltext, der eine komponentenspezifische Behandlung durchführt, in die Implementierung der Botschaftsbehandlungsmethode ein.

6 Rufen Sie die geerbte Botschaftsbehandlungsroutine auf.

Nachstehend sehen Sie ein Beispiel für eine benutzerdefinierte Botschaft. Ihr Name lautet *CM_CHANGECOLOR*.

```
const
   CM_CHANGECOLOR = WM_APP + 400;
type
   TMyComponent = class(TControl)
```

```
    ...
protected
    procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;
procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
    Color := Message.lParam;
    inherited;
end;
```

Kapitel

38

Komponenten zur Entwurfszeit verfügbar machen

Dieses Kapitel beschreibt, wie Sie Ihre Komponenten in der IDE verfügbar machen. Sie müssen dazu folgende Aktionen durchführen:

- Komponenten registrieren
- Paletten-Bitmaps hinzufügen
- Hilfe für die Komponenten bereitstellen
- Eigenschaftseditoren hinzufügen
- Komponenteneditoren hinzufügen
- Komponenten in Packages compilieren

Sie brauchen diese Aktionen nicht bei allen Komponenten auszuführen. Wenn Sie beispielsweise keine neuen Eigenschaften oder Ereignisse definieren, brauchen Sie dafür natürlich auch keine Hilfeinformationen bereitzustellen. Die beiden einzigen Arbeitsschritte, die immer durchgeführt werden müssen, sind Registrieren und Compilieren.

Nachdem Sie Ihre Komponenten registriert und in Packages compiliert haben, können Sie sie an andere Entwickler weitergeben und in der IDE installieren. Informationen über das Installieren von Packages in der IDE finden Sie im Abschnitt »Komponenten-Packages installieren« auf Seite 9-6.

Komponenten registrieren

Die Registrierung erfolgt auf der Grundlage von Compilierungs-Units. Komponenten, deren Implementierung in einer einzigen Unit enthalten ist, können daher in einem Schritt registriert werden.

Um eine Komponente zu registrieren, fügen Sie der Unit eine Register-Prozedur hinzu. Registrieren Sie in der Prozedur Ihre Komponenten, und geben Sie an, in welcher Registerkarte der Komponentenpalette sie installiert werden sollen.

Hinweis Wenn Sie eine Komponente in der IDE mit Hilfe der Menüoption *Komponente / Neue Komponente* erstellen, wird der Quelltext für die Registrierung automatisch eingefügt.

So registrieren Sie eine Komponente manuell:

- Die Register-Prozedur deklarieren
- Die Prozedur *Register implementieren*

Die Register-Prozedur deklarieren

Damit eine Komponente registriert werden kann, muß in der Unit eine Prozedur namens *Registe*r deklariert werden. Diese Funktion muß sich innerhalb eines **interface**-Abschnitts der Unit befinden.

Das folgende Beispiel zeigt die Struktur einer einfachen Unit-Datei, die neue Komponenten erzeugt und registriert:

```
unit MyBtns;
interface
type
      ...   { Die Komponententypen werden hier deklariert }
procedure Register;     { Muß Bestandteil des interface-Abschnitts sein }
implementation
      ...   { Hier folgt die Komponentenimplementierung }
procedure Register;
begin
      ...   { Komponenten registrieren }
end;
end.
```

In der Prozedur *Register* rufen Sie für jede Komponente, die Sie der Komponentenpalette hinzufügen wollen, die Routine *RegisterComponents* auf. Befinden sich in der Unit-Datei mehrere Komponenten, können sie in einem Arbeitsgang registriert werden.

Die Prozedur Register implementieren

Sie müssen jede Komponente, die Sie der Komponentenpalette hinzufügen möchten, innerhalb der Prozedur *Register* einer Unit registrieren. Sind mehrere Komponenten in der Datei vorhanden, können sie in einem Arbeitsgang registriert werden.

Beim Registrieren rufen Sie die Prozedur *RegisterComponents* für jede Registerkarte der Komponentenpalette auf, zu der Komponenten hinzugefügt werden sollen. *RegisterComponents* benötigt drei wichtige Informationen:

1. Die Komponenten angeben
2. Die Palettenseite angeben

3 Die Prozedur *RegisterComponents* aufrufen

Die Komponenten angeben

In der Prozedur *Register* übergeben Sie die Komponentennamen in einem Open-Array, welches Sie innerhalb des Aufrufs von *RegisterComponents* erzeugen können:

```
RegisterComponents('Verschiedenes', [TMyComponent]);
```

Sie können auch mehrere Komponenten gleichzeitig auf derselben Seite registrieren oder die Registrierung auf verschiedenen Seiten vornehmen, wie im folgenden Beispiel gezeigt:

```
procedure Register;
begin
      RegisterComponents('Verschiedenes', [TFirst, TSecond]); { Zwei auf einer Seite... }
      RegisterComponents('Gemischtes', [TThird]); { eine auf einer anderen ... }
      RegisterComponents(LoadStr(srStandard), [TFourth]); { und eine auf der Standardseite }
end;
```

Die Palettenseite angeben

Der Name der Palettenseite *Registerkarte* ist ein String-Wert. Wenn in der Komponentenpalette noch keine Registerkarte dieses Namens vorhanden ist, wird sie von Delphi automatisch erstellt. Die Namen der Standard-Registerkarten sind in einer String-Ressource gespeichert und können daher einfach an unterschiedliche Landessprachen angepaßt werden. Wenn Sie eine Komponente in einer der Standard-Registerkarten installieren wollen, ermitteln Sie den String für den Namen durch einen Aufruf der Funktion *LoadStr*. Übergeben Sie dabei die Konstante für die String-Ressource der Registerkarte (z. B. srSystem für die Registerkarte *System*).

Die Prozedur RegisterComponents aufrufen

Rufen Sie in *Register* die Prozedur *RegisterComponents* auf, um die Komponenten im Klassen-Array zu registrieren. *RegisterComponents* müssen drei Parameter übergeben werden: der Name einer Registerkarte in der Komponentenpalette, das Array mit den Komponentenklassen und der Index des letzten Array-Elements.

Geben Sie über den Parameter *Seite* den Namen der Registerkarte auf der Komponentenpalette an, unter welchem die Komponente erscheinen soll. Falls die angegebene Registerkarte bereits vorhanden ist, werden die Komponenten auf dieser Seite eingefügt. Falls die angegebene Registerkarte nicht vorhanden ist, erzeugt Delphi eine neue Registerkarte auf der Komponentenpalette mit der angegebenen Bezeichnung.

Rufen Sie *RegisterComponents* aus der Implementierung der Prozedur *Register* in einer der Units auf, die die benutzerdefinierten Komponenten definieren. Die Units, in denen die Komponenten definiert sind, müssen anschließend in ein Package compiliert werden. Dieses Package muß installiert werden, bevor die Komponenten der Komponentenpalette hinzugefügt werden.

```
procedure Register;
begin
```

```
        RegisterComponents('System', [TSystem1, TSystem2]); { Der Seite System hinzufügen }
        RegisterComponents('MeineEigeneSeite',[TCustom1, TCustom2]); { Neue Seite }
end;
```

Paletten-Bitmaps hinzufügen

Jede Komponente benötigt eine Bitmap-Grafik für die Anzeige in der Komponentenpalette. Wenn Sie kein eigenes Bitmap angeben, wird eine Standardgrafik verwendet.

Die Paletten-Bitmaps werden nur zur Entwurfszeit benötigt und dürfen daher nicht in die Unit der Komponente compiliert werden. Sie werden statt dessen in einer Ressourcen-Datei mit dem Namen der Unit-Datei und der Namenserweiterung DCR (Dynamic Component Resource) gespeichert. Sie können diese Ressourcen-Datei in Delphi mit Hilfe des Bildeditors erstellen. Jede Bitmap-Grafik sollte 24x24 Pixel groß sein.

Geben Sie für jede zu installierende Komponente eine Datei mit einer Bitmap-Grafik an. Die Grafikdatei muß denselben Namen wie die Komponente haben. Speichern Sie die Bitmap-Datei im selben Verzeichnis wie die compilierte Unit, damit sie beim Installieren der Komponente in der Palette gefunden werden kann.

Wenn Sie zum Beispiel eine Komponente mit der Bezeichnung *TMeinSteuerelement* in einer Unit mit der Bezeichnung *ToolBox* erzeugen, müssen Sie eine Ressource mit dem Namen TOOLBOX.DCR anlegen, die eine Bitmap mit dem Namen TMEINSTEUERELEMENT enthält. Bei Ressourcen-Namen wird nicht zwischen Groß- und Kleinschreibung unterschieden. Sie werden aber normalerweise in Großbuchstaben angegeben.

Hilfe für Komponenten bereitstellen

Wenn Sie in einem Formular eine Standardkomponente oder im Objektinspektor eine Eigenschaft bzw. ein Ereignis auswählen, können Sie durch Drücken von *F1* Hilfeinformationen zu diesem Element abrufen. Durch die Bereitstellung entsprechender Hilfedateien können Sie den Benutzern Ihrer Komponenten ebenfalls eine solche Hilfe zur Verfügung stellen.

Legen Sie Ihren Komponenten eine kleine Hilfedatei mit Informationen bei, die in das Hilfesystem von Delphi integriert wird.

Im Abschnitt »Die Hilfedatei erstellen« auf Seite 38-4 finden Sie Informationen bezüglich der Erzeugung von Hilfedateien für Komponenten.

Die Hilfedatei erstellen

Zur Erstellung der Quelldatei einer Windows-Hilfedatei (im RTF-Format) können Sie jedes dazu geeignete Programm verwenden. Im Lieferumfang von Delphi ist der Microsoft Help Workshop enthalten, mit dem Sie Hilfedateien compilieren können.

Die Anwendung verfügt über eine Online-Dokumentation mit ausführlichen Informationen zur Entwicklung von Hilfedateien.

Die Erzeugung einer Hilfedatei für Komponenten umfaßt folgende Schritte:

- Einträge erstellen
- Kontextsensitive Hilfe erstellen
- Der Delphi-Hilfe weitere Hilfedateien hinzufügen

Einträge erstellen

Damit die Hilfeinformationen zu Ihrer Komponente nahtlos in die Hilfe der anderen Komponenten in der Bibliothek integriert werden können, beachten Sie folgende Regeln:

1 Für jede Komponente sollte ein eigenes Hilfethema (Hilfebildschirm) erstellt werden.

   Der Hilfebildschirm sollte einen Hinweis auf die Unit mit der Komponentendeklaration und eine kurze Beschreibung enthalten. Außerdem sollten verschiedene Hyperlinks vorhanden sein, um Informationen über die Position der Komponente in der Objekthierarchie und über die verfügbaren Eigenschaften, Ereignisse und Methoden anzuzeigen. Anwendungsentwickler können auf den Hilfebildschirm zugreifen, indem sie die Komponente in einem Formular plazieren und *F1* drükken. Sehen Sie sich einige Beispiele an, indem Sie eine beliebige Komponente in ein Formular einfügen und *F1* drücken.

   Das Komponententhema muß über eine »#«-Fußnote (Kontext-Fußnote) verfügen und einen Wert enthalten, der nur zu diesem Thema gehört. Die »#«-Fußnote sorgt für die eindeutige Identifizierung eines Themas in einem Hilfesystem.

   Das Thema sollte eine »K«-Fußnote (Schlüsselwortfußnote) mit dem Namen der Komponente für die Schlüsselwortsuche enthalten. So lautet beispielsweise die Schlüsselwortfußnote für die Komponente *TMemo* »TMemo«.

   Außerdem sollte eine »$«-Fußnote (Titel-Fußnote) mit dem Titel des Themas vorhanden sein. Dieser Titel wird unter *Gefundene Themen und Lesezeichen* und im Fenster *Bisherige Themen* angezeigt.

2 Für jede Komponente sollten folgende Sekundärthemen definiert werden:

   - Ein Hierarchiethema mit Hyperlinks zu den Vorfahren der Komponente in der Komponentenhierarchie.
   - Eine Liste aller verfügbaren Eigenschaften mit Hyperlinks zu den zugehörigen Beschreibungen.
   - Eine Liste aller verfügbaren Ereignisse mit Hyperlinks zu den zugehörigen Beschreibungen.
   - Eine Liste aller verfügbaren Methoden mit Hyperlinks zu den zugehörigen Beschreibungen.

   Verknüpfungen mit Objektklassen, Eigenschaften, Methoden oder Ereignissen können im Delphi-Hilfesystem durch *Alinks* hergestellt werden. Bei der Ver-

knüpfung mit einer Objektklasse wird für den Alink der Klassenname des Objekts mit einem nachfolgenden Unterstrich und dem String »object« verwendet. Folgendermaßen erstellen Sie beispielsweise eine Verknüpfung mit dem Objekt TCustomPanel:

```
!AL(TCustomPanel_object,1)
```

Wenn Sie eine Verknüpfung mit einer Eigenschaft, einer Methode oder einem Ereignis definieren wollen, stellen Sie dem Element den Namen des zugehörigen Objekts und einen Unterstrich voran. Folgendermaßen wird beispielsweise eine Verknüpfung mit der Eigenschaft *Text* von *TControl* erstellt:

```
!AL(TControl_Text,1)
```

Sie können sich einen Überblick über die verschiedenen Sekundärthemen verschaffen, indem Sie das Hilfethema einer beliebigen Komponente anzeigen und auf die Hyperlinks mit der Bezeichnung *Hierarchie, Eigenschaften, Methoden* und *Ereignisse* klicken.

3 Für jede Eigenschaft oder Methode und jedes Ereignis der Komponente sollte ein Thema erstellt werden.

Der Bildschirm sollte eine kurze Beschreibung und einen Hinweis auf die Unit mit der Deklaration enthalten. Anwendungsentwickler können auf diese Themen zugreifen, indem sie das betreffende Element im Objektinspektor auswählen und *F1* drücken oder den Cursor im Quelltext-Editor in den Elementnamen setzen und *F1* drücken. Sehen Sie sich einige Beispiele an, indem Sie ein beliebiges Element im Objektinspektor auswählen und *F1* drücken.

Das Thema sollte eine »K«-Fußnote mit dem Namen des Elements und einer Kombination von Element- und Komponentenname enthalten. Die Eigenschaft *Text* von *TControl* hat beispielsweise folgende »K«-Fußnote:

```
Text,TControl;TControl,Text;Text,
```

Außerdem muß eine »$«-Fußnote mit dem Titel des Themas vorhanden sein (z. B. TControl.Text).

Alle Themen müssen eine eindeutige ID als »#«-Fußnote haben.

Kontextsensitive Hilfe erstellen

Jedes Thema für eine Komponente, eine Eigenschaft, eine Methode oder ein Ereignis muß eine »A«-Fußnote haben. Sie wird zum Anzeigen des Themas verwendet, wenn der Benutzer die Komponente auswählt und *F1* drückt oder eine Eigenschaft bzw. ein Ereignis im Objektinspektor auswählt und *F1* drückt. Für die »A«-Fußnote gelten folgende Benennungskonventionen:

Bezieht sich das Hilfethema auf eine Komponente, besteht die Fußnote wie im folgenden Beispiel aus zwei durch ein Semikolon getrennten Einträgen:

```
Komponentenklasse_Object;Komponentenklasse
```

Hier ist *Komponentenklasse* der Name der Komponentenklasse.

Betrifft das Hilfethema eine Eigenschaft oder ein Ereignis, besteht die Fußnote aus drei durch Semikolons getrennten Einträgen:

```
Komponentenklasse_Element;Element_Typ;Element
```

Komponentenklasse ist der Name der Klasse der Komponente, Element ist der Name der Eigenschaft, der Methode oder des Ereignisses, und Typ ist entweder *Property*, *Method* oder *Event* (Eigenschaft, Methode oder Ereignis).

So lautet beispielsweise die »A«-Fußnote für die Eigenschaft *Hintergrundfarbe* der Komponente *TMeinGitter* folgendermaßen:

```
TMeinGitter_Hintergrundfarbe;Hintergrundfarbe_Property;Hintergrundfarbe
```

Hilfedateien für Komponenten hinzufügen

Sie können eine Hilfedatei in Delphi mit dem Dienstprogramm OpenHelp (OH.EXE) im Verzeichnis BIN hinzufügen. In der IDE können Sie mit *Hilfe / Anpassen* auf das Dienstprogramm zugreifen.

In der Datei OpenHelp.hlp finden Sie Informationen zur Verwendung von OpenHelp, unter anderem auch dazu, wie Sie Ihre eigenen Hilfedateien dem Hilfesystem hinzufügen können.

Eigenschaftseditoren hinzufügen

Im Objektinspektor stehen Ihnen Standardtechniken für das Bearbeiten aller Arten von Eigenschaften zur Verfügung. Sie können jedoch für bestimmte Eigenschaften einen anderen Editor bereitstellen, indem Sie einen eigenen Eigenschaftseditor schreiben und registrieren. Sie können Editoren nur für die Eigenschaften Ihrer selbsterstellten Komponenten oder für alle Eigenschaften eines bestimmten Typs erstellen.

Im einfachsten Fall dient ein Eigenschaftseditor zum Anzeigen und Bearbeiten des aktuellen Wertes als String oder zum Anzeigen eines Dialogfeldes, in dem andere Bearbeitungen durchgeführt werden können. In Abhängigkeit von der zu bearbeitenden Eigenschaft können Sie eine oder beide dieser Möglichkeiten implementieren.

Ein Eigenschaftseditor wird in diesen fünf Arbeitsschritten erstellt:

1. Ableiten einer Eigenschaftseditor-Klasse
2. Bearbeiten der Eigenschaft als Text
3. Bearbeiten der Eigenschaft als Einheit
4. Festlegen der Editorattribute
5. Registrieren des Eigenschaftseditors

Ableiten einer Eigenschaftseditor-Klasse

In der Unit *DsgnIntf* sind mehrere Arten von Eigenschaftseditoren definiert, die von der Klasse *TPropertyEditor* abgeleitet sind. Beim Erstellen eines Editors können Sie Ihre Editorklasse entweder direkt von *TPropertyEditor* oder indirekt von einer der in der folgenden Tabelle aufgeführten Klassen ableiten.

Die Unit *DsgnIntf* enthält auch einige spezialisierte Editoren, die nur für eine bestimmte Eigenschaft verwendet werden (z. B. für den Komponentennamen). Die aufgelisteten Eigenschaftseditoren eignen sich am besten für benutzerdefinierte Eigenschaften.

Tabelle 38.1 Vordefinierte Eigenschaftseditor-Typen

Typ	Eigenschaften
TOrdinalProperty	Alle von *TOrdinalProperty* abgeleiteten Editoren für Integer-, Zeichen- und Aufzählungseigenschaften
TIntegerProperty	Alle Integer-Typen einschließlich vor- und benutzerdefinierter Teilbereiche
TCharProperty	Zeichen und Zeichen-Teilbereiche wie 'A'...'Z'
TEnumProperty	Alle Aufzählungstypen
TFloatProperty	Alle Gleitkommawerte
TStringProperty	String-Typen
TSetElementProperty	Einzelne Elemente in Mengen (angezeigt als Boolesche Werte)
TSetProperty	Alle Mengen. Mengen können nicht direkt bearbeitet werden. Sie werden als Liste der einzelnen Mengenelement-Eigenschaften angezeigt.
TClassProperty	Klassen. Der Klassenname wird angezeigt, und die Eigenschaften der Klasse können eingeblendet werden.
TMethodProperty	Methodenzeiger, vor allem Ereignisse.
TComponentProperty	Komponenten im selben Formular. Der Benutzer kann die Eigenschaften der Komponente nicht bearbeiten, aber auf eine bestimmte Komponente eines kompatiblen Typs verweisen.
TColorProperty	Komponentenfarben. Sie werden als Farbkonstanten oder hexadezimale Werte angezeigt. Die Farbkonstanten werden in einer Dropdown-Liste angeboten. Durch Doppelklicken wird ein Dialogfeld zur Farbauswahl geöffnet.
TFontNameProperty	Schriftnamen. In der Dropdown-Liste werden alle im System installierten Schriften angezeigt.
TFontProperty	Schriften. Die einzelnen Schrifteigenschaften werden angezeigt, und das Dialogfeld *Schriftart* kann geöffnet werden.

Im folgenden Beispielquelltext wird ein einfacher Eigenschaftseditor mit dem Namen *TMyPropertyEditor* deklariert:

```
type
      TFloatProperty = class(TPropertyEditor)
      public
         function AllEqual: Boolean; override;
         function GetValue: string; override;
         procedure SetValue(const Value: string); override;
      end;
```

Die Eigenschaft als Text bearbeiten

Bei allen Eigenschaften muß für die Anzeige im Objektinspektor eine String-Darstellung ihrer Werte bereitgestellt werden. Die meisten Eigenschaften ermöglichen dem

Benutzer auch, neue Werte einzugeben. Eigenschaftseditor-Klassen stellen virtuelle Methoden bereit, die Sie überschreiben können, um Konvertierungen zwischen der Textdarstellung und dem eigentlichen Wert durchzuführen.

Die zu überschreibenden Methoden heißen *GetValue* und *SetValue*. Ihr Editor erbt auch eine Reihe von Methoden, mit denen verschiedene Arten von Werten gelesen und geschrieben werden können (siehe der folgenden Tabelle).

Tabelle 38.2 Methoden zum Lesen und Schreiben von Eigenschaften

Eigenschaftstyp	Get-Methode (Lesen)	Set-Methode (Schreiben)
Gleitkomma	GetFloatValue	SetFloatValue
Methodenzeiger (Ereignis)	GetMethodValue	SetMethodValue
Ordinaltyp	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue

Wenn Sie eine *GetValue*-Methode überschreiben, wird eine der *Get*-Methoden, wenn Sie *SetValue* überschreiben, eine der *Set*-Methoden aufgerufen.

Den Eigenschaftswert anzeigen

Die Methode *GetValue* des Eigenschaftseditors gibt einen String mit dem aktuellen Wert der Eigenschaft zurück. Im Objektinspektor wird dieser String als Wert der Eigenschaft angezeigt. Standardmäßig gibt *GetValue* »unknown« zurück.

Sie erhalten eine String-Darstellung Ihrer Eigenschaft, indem Sie die Methode *GetValue* überschreiben.

Wenn der Wert der Eigenschaft kein String ist, wird er von der Methode konvertiert.

Den Eigenschaftswert angeben

Die Methode *SetValue* nimmt einen im Objektinspektor eingegebenen String entgegen, konvertiert ihn in den entsprechenden Typ und weist ihn der Eigenschaft als Wert zu. Enthält der String keinen für die Eigenschaft gültigen Wert, sollte *SetValue* eine Exception auslösen und den Wert verwerfen.

Um String-Werte in Eigenschaften zu speichern, überschreiben Sie die Methode *SetValue* des Eigenschaftseditors.

In *SetValue* sollte der String konvertiert und vor dem Aufrufen einer der Set-Methoden validiert werden.

Nachfolgend die Methoden *GetValue* und *SetValue* der Eigenschaft *TIntegerProperty*. Da Integer ein Ordinaltyp ist, ruft *GetValue* die Methode *GetOrdValue* auf und wandelt das Ergebnis in einen String um. *SetValue* wandelt den String in einen Integer-Wert um, führt Bereichsprüfungen durch und ruft anschließend *SetOrdValue* auf.

```
function TIntegerProperty.GetValue: string;
begin
      Result := IntToStr(GetOrdValue);
end;
procedure TIntegerProperty.SetValue(const Value: string);
```

```
var
      L: Longint;
begin
      L := StrToInt(Value); { String in Zahl umwandeln }
      with GetTypeData(GetPropType)^ do { verwendet Compiler-Daten für den Integertyp }
         if (L < MinValue) or (L > MaxValue) then { Bereichprüfung durchführen... }
         raise EPropertyError.Create( { ...und bei Fehler Exception auslösen}
            FmtLoadStr(SOutOfRange, [MinValue, MaxValue]));
      SetOrdValue(L); { Falls Bereichsprüfung OK, den Wert setzen}
end;
```

Bei den konkreten Beispielen ist das Konzept wichtig und weniger die Einzelheiten: *GetValue* wandelt den Wert in einen String um; *SetValue* wandelt den String um und prüft den Wert, bevor eine der »Set«-Methoden aufgerufen wird.

Die Eigenschaft als Einheit bearbeiten

Sie können optional ein Dialogfeld bereitstellen, in dem die Eigenschaft visuell bearbeitet werden kann. Solche Editoren werden oft für Eigenschaften verwendet, die selbst Klassen sind. Ein Beispiel ist die Eigenschaft *Font*, bei der alle Attribute in einem Dialogfeld festgelegt werden können.

Wenn Sie ein Dialogfeld zum Bearbeiten der Eigenschaft bereitstellen wollen, überschreiben Sie die Methode *Edit* des Editors.

Edit-Methoden verwenden die üblichen Get- und Set-Methoden. Tatsächlich ruft eine Edit-Methode sowohl eine Get- als auch eine Set-Methode auf. Da der Editor typspezifisch ist, brauchen die Eigenschaftswerte normalerweise nicht in Strings konvertiert zu werden. Der Editor verwendet den Wert im allgemeinen so, wie er ihn vorfindet.

Wenn der Benutzer auf die Ellipsen-Schaltfläche (...) klickt oder auf den Wert der Eigenschaft doppelklickt, wird die Methode *Edit* des Eigenschaftseditors aufgerufen.

Führen Sie in Ihrer Implementierung der Methode *Edit* folgende Operationen durch:

1. Instantiieren Sie den Editor für die Eigenschaft.
2. Lesen Sie den aktuellen Wert, und weisen Sie ihn der Eigenschaft mit einer *Get-Methode* zu.
3. Gibt der Benutzer einen neuen Wert ein, weisen Sie ihn der Eigenschaft mit einer *Set-Methode* zu.
4. Geben Sie den Editor frei.

Die Eigenschaften *Color*, die Sie in vielen Komponenten finden, verwenden als Eigenschaftseditor das Standard-Farb-Dialogfeld von Windows. Hier die Methode *Edit* der Eigenschaft *TColorProperty*, welche das Dialogfeld aufruft und das Ergebnis verwendet:

```
procedure TColorProperty.Edit;
var
      ColorDialog: TColorDialog;
begin
      ColorDialog := TColorDialog.Create(Application); { Editor erzeugen}
```

```
      try
        ColorDialog.Color := GetOrdValue; { Vorhandenen Wert verwenden }
        if ColorDialog.Execute then { Wenn der Anwender auf OK klickt ... }
        SetOrdValue(ColorDialog.Color); { ...das Ergebnis verwenden, um den Wert zu setzen}
      finally
        ColorDialog.Free; { Editor freigeben }
      end;
end;
```

Editorattribute festlegen

Der Eigenschaftseditor muß Informationen bereitstellen, die dem Objektinspektor mitteilen, welche Tools angezeigt werden sollen. Der Objektinspektor muß beispielsweise wissen, ob eine Eigenschaft Untereigenschaften hat oder ob er eine Liste ihrer möglichen Werte anzeigen kann.

Um Editorattribute anzugeben, überschreiben Sie die Methode *GetAttributes* des Eigenschaftseditors.

Die Methode gibt eine Menge von Werten des Typs *TPropertyAttributes* zurück, die einen oder alle der folgenden Werte enthalten kann:

Tabelle 38.3 Attribute für Eigenschaftseditoren

Attribut	Zugehörige Methode	Bedeutung
paValueList	GetValues	Der Editor kann eine Liste von Aufzählungswerten bereitstellen.
paSubProperties	GetProperties	Die Eigenschaft hat Untereigenschaften, die angezeigt werden können.
paDialog	Edit	Der Editor kann ein Dialogfeld zum Bearbeiten der gesamten Eigenschaft anzeigen.
paMultiSelect	Keine	Die Eigenschaft sollte anzeigen, wenn der Benutzer eine oder mehrere Komponenten auswählt.
paAutoUpdate	SetValue	Die Komponente wird nach jeder Änderung aktualisiert, anstatt auf die Bestätigung des Wertes zu warten.
paSortList	Keine	Der Objektinspektor sollte die Werteliste sortieren.
paReadOnly	Keine	Der Wert der Eigenschaft kann nicht geändert werden.
paRevertable	Keine	Die Menüoption *Zur Vererbung zurückkehren* im lokalen Menü des Objektinspektors wird aktiviert. Der aktuelle Eigenschaftswert wird verworfen und der vorherige Standardwert wiederhergestellt.

Farbeigenschaften unterscheiden sich von den meisten anderen Eigenschaften dadurch, daß der Benutzer ihre Werte im Objektinspektor auf mehrere Arten festlegen kann: Er kann sie direkt eingeben, aus einer Liste wählen oder einen benutzerdefinierten Editor verwenden. Der Rückgabewert der Methode *GetAttributes* der Klasse *TColorProperty* enthält daher mehrere Attribute. Der Object Pascal-Quelltext für die Methode lautet folgendermaßen:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
      Result := [paMultiSelect, paDialog, paValueList];
end;
```

Den Eigenschaftseditor registrieren

Nachdem Sie einen Eigenschaftseditor erstellt haben, müssen Sie ihn in Delphi registrieren. Bei diesem Vorgang wird ein Eigenschaftstyp einem bestimmten Editor zugeordnet. Sie können den Editor für alle Eigenschaften eines bestimmten Typs oder nur für eine Eigenschaft eines Komponententyps registrieren.

Um die Registrierung durchzuführen, rufen Sie die Funktion *RegisterPropertyEditor* auf.

Der Prozedur *RegisterPropertyEditor* müssen vier Argumente übergeben werden:

- Ein Typinformationszeiger für den zu bearbeitenden Eigenschaftstyp.
- Dies ist immer ein Aufruf der integrierten Funktion *TypeInfo*, wie zum Beispiel *TypeInfo*(*TMyComponent*).
- Der Komponententyp, für den der Editor verwendet werden soll. Hat dieser Parameter den Wert **nil**, wird der Editor für alle Eigenschaften des angegebenen Typs verwendet.
- Der Name der Eigenschaft. Dieser Parameter wird nur berücksichtigt, wenn Sie mit dem vorherigen Argument einen bestimmten Komponententyp angeben. In diesem Fall können Sie den Namen einer bestimmten Eigenschaft dieses Komponententyps angeben, für die der Editor verwendet werden soll.
- Der Editortyp, der zum Bearbeiten der angegebenen Eigenschaft verwendet werden soll.

Das folgende Beispiel zeigt einen Auszug aus der Prozedur, mit der die Editoren für die Standardkomponenten in der Komponentenpalette registriert werden:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

Die drei Anweisungen in der Prozedur zeigen die unterschiedlichen Verwendungsweisen von *RegisterPropertyEditor*:

- Die erste Anweisung ist die am häufigsten verwendete. Sie registriert den Eigenschaftseditor *TComponentProperty* für alle Eigenschaften des Typs *TComponent* (oder für von *TComponent* abgeleitete Klassen ohne eigenen Editor). Im allgemeinen erstellen Sie einen Editor für einen bestimmten Typ, um ihn für alle Eigenschaften dieses Typs zu verwenden. In diesem Fall werden als zweiter und dritter Parameter der Wert **nil** und ein Leerstring übergeben.

- Die zweite Anweisung registriert einen Editor für eine spezielle Eigenschaft. In diesem Beispiel wird ein Editor für die Eigenschaft *Name* (*TComponentName*) aller Komponenten registriert.
- Die dritte Anweisung registriert einen Editor für alle Eigenschaften des Typs *TMenuItem* in *TMenu*-Komponenten.

Komponenteneditoren hinzufügen

Komponenteneditoren bestimmen, was geschieht, wenn der Benutzer im Entwurfsfenster auf die Komponente doppelklickt. Sie fügen dem lokalen Menü der Komponente Befehle hinzu, und sie ermöglichen das Kopieren von Komponenten in benutzerdefinierten Formaten in die Zwischenablage.

Wenn Sie für Ihre Komponente keinen Editor definieren, wird der Standardeditor (*TDefaultEditor*) verwendet. *TDefaultEditor* fügt dem lokalen Menü der Komponente keine neuen Einträge hinzu. Wenn der Benutzer auf die Komponente doppelklickt, durchsucht *TDefaultEditor* die Eigenschaften der Komponente und generiert (oder aktiviert) die erste gefundene Ereignisbehandlungsroutine.

Wenn Sie neue Einträge in das lokale Menü aufnehmen, das Verhalten der Komponente bei einem Doppelklick ändern oder neue Zwischenablageformate hinzufügen wollen, leiten Sie eine Klasse von *TComponentEditor* ab und registrieren sie für die Komponente. In den überschriebenen Methoden können Sie über die Eigenschaft *Component* von *TComponentEditor* auf die zu bearbeitende Komponente zugreifen.

Das Hinzufügen eines benutzerdefinierten Komponenteneditors besteht aus folgenden Schritten:

- Einträge in das Kontextmenü einfügen
- Das Doppelklickverhalten ändern
- Zwischenablageformate hinzufügen
- Den Komponenteneditor registrieren

Einträge in das lokale Menü einfügen

Wenn der Benutzer mit der rechten Maustaste auf die Komponente klickt, werden die Methoden *GetVerbCount* und *GetVerb* des Komponenteneditors aufgerufen, um Befehle (Verben) in das lokale Menü einzufügen.

Das Hinzufügen von Einträgen in das lokale Menü erfordert folgende Schritte:

- Menüeinträge angeben
- Befehle implementieren

Menüeinträge angeben

Überschreiben Sie die Methode *GetVerbCount*, um die Anzahl der Menüeinträge festzulegen, die hinzugefügt werden sollen. Durch Überschreiben der Methode *GetVerb* geben Sie die Strings an, die im lokalen Menü für die Befehle angezeigt werden sollen. Wenn Sie in der Methode *GetVerb* in einen String das Zeichen & einfügen, wird das nachfolgende Zeichen unterstrichen angezeigt und kann als Tastenkürzel für den betreffenden Menüeintrag verwendet werden. Soll durch einen Befehl ein Dialogfeld geöffnet werden, fügen Sie am Ende des Strings eine Ellipse (...) hinzu. Der einzige Parameter von *GetVerb* gibt den Index des Befehls an.

Im folgenden Beispiel werden die Methoden *GetVerbCount* und *GetVerb* überschrieben, um zwei neue Befehle in das lokale Menü einzufügen.

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := "&DoThis ...";
    1: Result := "Do&That";
  end;
end;
```

Hinweis Achten Sie darauf, daß Ihre *GetVerb*-Methode für jeden von *GetVerbCount* angegebenen Index einen Wert zurückgibt.

Befehle implementieren

Wenn der Benutzer im Entwurfsfenster einen von *GetVerb* bereitgestellten Befehl auswählt, wird die Methode *ExecuteVerb* aufgerufen. Implementieren Sie für jeden auf diese Weise definierten Befehl eine Aktion in *ExecuteVerb*. Auf die zu bearbeitende Komponente können Sie über die Editoreigenschaft *Component* zugreifen.

In der folgenden Methode *ExecuteVerb* werden die Befehle für die Methode *GetVerb* im vorhergehenden Beispiel implementiert.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
      MySpecialDialog: TMyDialog;
begin
      case Index of
         0: begin
            MyDialog := TMySpecialDialog.Create(Application); { Editor-Instanz erzeugen }
            if MySpecialDialog.Execute then; { Wenn der Anwender auf OK klickt, ... }
            MyComponent.FThisProperty := MySpecialDialog.ReturnValue; { ...Wert verwenden }
            MySpecialDialog.Free; { Editor freigeben }
            end;
         1: That; { Methode That aufrufen }
      end;
end;
```

Das Doppelklickverhalten ändern

Wenn der Benutzer auf die Komponente doppelklickt, wird die Methode *Edit* des Komponenteneditors aufgerufen. Standardmäßig wird dadurch der erste dem lokalen Menü hinzugefügte Befehl ausgeführt. Im vorhergehenden Beispiel ist dies der Befehl *DoThis*.

Obwohl es in der Regel empfehlenswert ist, den ersten Befehl auszuführen, kann es vorkommen, daß Sie dieses Standardverhalten ändern möchten. Sie können beispielsweise in folgenden Situationen das Standardverhalten ändern:

- Wenn Sie dem lokalen Menü keine Befehle hinzufügen.
- Wenn Sie ein Dialogfeld anzeigen wollen, in dem mehrere Befehle zusammengefaßt sind.

Das Standardverhalten kann durch Überschreiben der Methode *Edit* geändert werden. Im folgenden Beispiel wird ein Dialogfeld geöffnet, wenn der Benutzer auf die Komponente doppelklickt:

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

Hinweis Wenn bei einem Doppelklick auf die Komponente der Quelltext-Editor für eine Ereignisbehandlungsroutine angezeigt werden soll, verwenden Sie als Basisklasse für Ihren Komponenteneditor *TDefaultEditor* anstelle von *TComponentEditor*. Überschreiben Sie dann anstelle von *Edit* die geschützte Methode *TDefaultEditor.EditProperty*. *EditProperty* durchsucht alle Ereignisbehandlungsroutinen der Komponente und öffnet die zuerst gefundene. Sie können jedoch auch wie im folgenden Beispiel nach einem bestimmten Ereignis suchen:

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

Zwischenablageformate hinzufügen

Wählt der Benutzer den Befehl *Kopieren*, während eine Komponente in der IDE ausgewählt ist, wird die Komponente standardmäßig im internen Format von Delphi

kopiert. Sie kann dann in ein anderes Formular bzw. Datenmodul eingefügt werden. Indem Sie die Methode *Copy* überschreiben, können Sie für Ihre Komponente weitere Zwischenablageformate definieren.

Das folgende Beispiel ermöglicht einer *TImage-Komponente*, ihre Grafik in die Zwischenablage zu kopieren. Die Grafik wird von der Delphi-IDE ignoriert, kann aber in andere Anwendungen eingefügt werden.

```
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
```

Den Komponenteneditor registrieren

Nachdem der Komponenteneditor definiert wurde, kann er für eine bestimmte Komponentenklasse registriert werden. Ein registrierter Editor wird automatisch für jede Komponente dieser Klasse erstellt, wenn sie im Formular-Designer ausgewählt wird.

Die Zuordnung zwischen einem Komponenteneditor und einer Komponentenklasse wird mit Hilfe der Methode *RegisterComponentEditor* hergestellt. Dieser Methode werden beim Aufruf die Namen der Komponentenklasse und der Editorklasse übergeben. Mit der folgenden Anweisung wird beispielsweise die Editorklasse *TMyEditor* für Komponenten des Typs *TMyComponent* registriert:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

Plazieren Sie den Aufruf von *RegisterComponentEditor* in der Register-Prozedur, in dem Sie Ihre Komponente registrieren. Wenn Sie beispielsweise eine neue Komponente mit dem Namen *TMyComponent* und deren Komponenteneditor *TMyEditor* in der gleichen Datei implementieren, werden die Komponente und ihre Verbindung zum Editor folgendermaßen registriert:

```
procedure Register;
begin
      RegisterComponents('Verschiedenes', [TMyComponent);
      RegisterComponentEditor(classes[0], TMyEditor);
end;
```

Eigenschaftskategorien

In der IDE von Delphi bietet der Objektinspektor dem Programmierer die Möglichkeit, Eigenschaften basierend auf Eigenschaftskategorien selektiv ein- bzw. auszublenden. Die Eigenschaften neuer benutzerdefinierter Komponenten können durch Registrieren der Eigenschaften in Kategorien in dieses Schema eingefügt werden. Dies kann beim Registrieren der Komponente durch Aufruf einer der Eigenschaftsregistrierungsfunktionen *RegisterPropertyInCategory* oder *RegisterPropertiesInCategory*

erfolgen. Verwenden Sie die erste Funktion zum Registrieren einer einzelnen Eigenschaft. Die zweite Funktion erlaubt das Registrieren mehrerer Eigenschaften mit einem einzelnen Funktionsaufruf. Diese Funktionen sind in der Unit *DsgnIntf* definiert.

Beachten Sie, daß weder das Registrieren von Eigenschaften noch das Registrieren aller Eigenschaften einer benutzerdefinierten Komponente erforderlich ist, wenn Sie nur einige ihrer Eigenschaften registrieren wollen. Eine nicht explizit einer Kategorie zugeordnete Eigenschaft wird automatisch in der Kategorie *TMiscellaneousCategory* verwaltet. Diese Kategorien werden im Objektinspektor nach Maßgabe der Standardkategorisierung ein- oder ausgeblendet.

Im Lieferumfang von Delphi befinden sich 13 Eigenschaftskategorien in der Form von Eigenschaftsklassen. Die können eine Eigenschaft einer benutzerdefinierten Komponente in einer dieser vorhandenen Kategorien registrieren oder basierend auf den integrierten Klassen eigene Eigenschaftskategorieklassen erstellen.

Außer den bereits erwähnten Funktionen zum Registrieren von Eigenschaften gibt es die Funktion *IsPropertyInCategory*. Mit dieser Funktion können Sie beispielsweise Lokalisierungsdienstprogramme erstellen, in denen ermittelt werden muß, ob eine Eigenschaft in einer bestimmten Eigenschaftskategorie registriert ist.

Eine Eigenschaft registrieren

Mit der Funktion *RegisterPropertyInCategory* können Sie jeweils eine Eigenschaft registrieren und einer Eigenschaftskategorie zuordnen. *RegisterPropertyInCategory* ist in vier überladenen Versionen verfügbar, die über eigene Kriterien zur Identifizierung der Eigenschaft in der benutzerdefinierten Komponente verfügen, die der Eigenschaftskategorie zugeordnet werden soll.

Die erste Version ermöglicht die Identifizierung der Eigenschaft über ihren Namen. Die folgende Zeile registriert eine Eigenschaft, die der Anzeige der Komponente dient. Die Eigenschaft wird über ihren Namen *AutoSize* bezeichnet.

```
RegisterPropertyInCategory(TVisualCategory, 'AutoSize');
```

Die zweite Version identifiziert die Eigenschaft über den charakteristischen Klassentyp der Komponente und den Eigenschaftsnamen. Das folgende Beispiel registriert eine Eigenschaft namens *HelpContext* einer Komponente der benutzerdefinierten Klasse *TMyButton* in der Kategorie *THelpCategory*.

```
RegisterPropertyInCategory(THelpCategory, TMyButton, 'HelpContext');
```

Die dritte Version verwendet zur Identifizierung der Eigenschaft deren Typ und Namen. Das folgende Beispiel registriert eine Eigenschaft basierend auf einer Kombination von Typ (Integer) und Name (*Width)*.

```
RegisterPropertyInCategory(TVisualCategory, TypeInfo(Integer), 'Width');
```

Die letzte Version identifiziert die Eigenschaft ausschließlich über ihren Eigenschaftstyp. Das folgende Beispiel registriert eine Eigenschaft basierend auf ihrem Typ (Integer).

```
RegisterPropertyInCategory(TVisualCategory, TypeInfo(Integer));
```

Im Abschnitt »Eigenschaftskategorieklassen« auf Seite 38-18 finden Sie eine Liste der verfügbaren Eigenschaftskategorien und eine kurze Beschreibung ihres Einsatzbereichs.

Mehrere Eigenschaften gleichzeitig registrieren

Mit der Funktion *RegisterPropertiesInCategory* können Sie mehrere Eigenschaften gleichzeitig registrieren und einer Eigenschaftskategorie zuordnen. *RegisterPropertiesInCategory* ist in drei überladenen Versionen verfügbar, die jeweils unterschiedliche Kriterien zur Identifizierung der Eigenschaft in der benutzerdefinierten Komponente verwenden, die der Eigenschaftskategorie zugeordnet werden soll.

Mit der ersten Version können Sie Eigenschaften basierend auf dem Eigenschaftsnamen für die Zuordnung zu einer Eigenschaftskategorie identifizieren. Eine Liste mit Eigenschaftsnamen wird als Array des Typs String übergeben. Jede über ihren Namen in der Liste identifizierte Eigenschaft wird in der angegebenen Eigenschaftskategorie registriert. Im folgenden Beispiel werden vier Eigenschaften in der Kategorie *THelpCategory* registriert. Diese vier Eigenschaften werden über ihren Namen mit den Strings »HelpContext«, »Hint«, »ParentShowHint« und »ShowHint« identifiziert.

```
RegisterPropertiesInCategory(THelpCategory, ['HelpContext', 'Hint', 'ParentShowHint',
  'ShowHint']);
```

Die zweite Version identifiziert die Eigenschaften über ihren Typ. Im folgenden Beispiel werden alle Eigenschaften des Typs String der benutzerdefinierten Komponente in der Kategorie *TLocalizableCategory* registriert.

```
RegisterPropertiesInCategory(TLocalizableCategory, TypeInfo(String));
```

Die dritte Version ermöglicht die Übergabe einer Liste verschiedener Kriterien, die nicht denselben Typ besitzen müssen und der Identifizierung der zu registrierenden Eigenschaften dienen. Die Liste wird als Array von Konstanten übergeben. Im folgenden Beispiel wird jede Eigenschaft in der Kategorie *TLocalizableCategory* registriert, die den Namen »Text« besitzt oder zu einer Klasse des Typs *TEdit* gehört.

```
RegisterPropertiesInCategory(TLocalizableCategory, ['Text', TEdit]);
```

Im Abschnitt »Eigenschaftskategorieklassen« auf Seite 38-18 finden Sie eine Liste der verfügbaren Eigenschaftskategorien und eine kurze Beschreibung ihres Einsatzbereichs.

Eigenschaftskategorieklassen

Integrierte Eigenschaftskategorien

Delphi stellt zwölf integrierte Eigenschaftskategorien bereit, denen Sie Eigenschaften in benutzerdefinierten Komponenten zuordnen können. Verwenden Sie einen dieser

Namen von Eigenschaftskategorien für den Parameter *ACategoryClass* der Funktionen *RegisterPropertyInCategory* und *RegisterPropertiesInCategory*:

Tabelle 38.4 Eigenschaftskategorien

Kategorie	Einsatzbereich
TActionCategory	Eigenschaften für Laufzeitaktionen wie *Enabled* und *Hint* von *TEdit*.
TDatabaseCategory	Eigenschaften für Datenbankoperationen wie *DatabaseName* und *SQL* von *TQuery*.
TDragNDropCategory	Eigenschaften für Drag&Drop- bzw. Drag&Dock-Operationen wie *DragCursor* und *DragKind* von *TImage*.
THelpCategory	Eigenschaften zur Nutzung von Online-Hilfe und Hinweisen wie *HelpContext* und *Hint* von *TMemo*.
TLayoutCategory	Eigenschaften zur Anzeige von Steuerelementen während des Entwurfs wie *Top* und *Left* von *TDBEdit*.
TLegacyCategory	Eigenschaften für veraltete Operationen wie *Ctl3D* und *ParentCtl3D* von *TComboBox*.
TLinkageCategory	Eigenschaften für die Zuordnung bzw. Verknüpfung von Komponenten wie *DataSet* von *TDataSource*.
TLocaleCategory	Eigenschaften für internationale Sprachtreiber wie *BiDiMode* und *ParentBiDiMode* von *TMainMenu*.
TLocalizableCategory	Eigenschaften für Lokalisierungszwecke.
TMiscellaneousCategor y	Eigenschaften, die keiner anderen Kategorie zugeordnet werden können oder nicht kategorisiert werden müssen (sowie Eigenschaften, die nicht explizit in einer Kategorie registriert sind), wie *AllowAllUp* und *Name* von *TSpeedButton*.
TVisualCategory	Eigenschaften für die Anzeige eines Steuerelements zur Laufzeit wie *Align* und *Visible* von *TScrollBox*.
TInputCategory	Eigenschaften für die Dateneingabe (nicht unbedingt in Zusammenhang mit Datenbankoperationen) wie *Enabled* und *ReadOnly* von *TEdit*.

Neue Eigenschaftskategorien ableiten

Sie können neue Eigenschaftskategorien erstellen, indem Sie eine Klasse von der Basisklasse *TPropertyCategory* oder einem ihrer integrierten Nachkommen ableiten. Im Abschnitt »Eigenschaftskategorieklassen« finden Sie eine Liste der verfügbaren Eigenschaftskategorien und eine kurze Beschreibung ihres Einsatzbereiches.

Überschreiben Sie die Methode *Name*, wenn Sie eine neue Eigenschaftskategorieklasse ableiten. Die Methode *Name* stellt den Namen der Kategorie für die Anzeige im Objektinspektor bereit. Diese Methode muß mit einer Methode überschrieben werden, die den Namen der benutzerdefinierten Kategorie zurückgibt. Die Methode *Name* kann einfach einen String zurückgeben oder einen Wert aus einer Ressource abzurufen. Letzteres erleichtert die Lokalisierung benutzerdefinierter Komponenten und Kategorien.

Mit der folgenden Methode *Name* einer benutzerdefinierten Klasse wird der Text »Spezial« im Objektinspektor als Eigenschaftskategorie angezeigt (wenn mindestens

eine der Eigenschaften des aktuellen Objekts in dieser Eigenschaftsklasse registriert ist.)

```
class function TMySpecialCategory.Name: String;
begin
  Result := 'Spezial';
end;
```

Die Funktion IsPropertyInCategory

Eine Anwendung kann die registrierten Eigenschaften abfragen und so feststellen, ob eine Eigenschaft bereits in einer bestimmten Kategorie registriert wurde. Dies ist insbesondere für Dienstprogramme zur Lokalisierung hilfreich, die als Vorbereitung der eigentlichen Lokalisierung die Kategorisierung der Eigenschaften überprüfen. Zwei überladene Versionen der Funktion *IsPropertyInCategory* sind verfügbar. Diese legen bei der Ermittlung der Kategorien, in denen eine Eigenschaft registriert ist, unterschiedliche Kriterien zugrunde.

Die erste Version verwendet als Vergleichskriterium eine Kombination aus dem Klassentyp der Eigentümerkomponente und dem Eigenschaftsnamen. Im folgenden Beispiel muß die Eigenschaft zu *TEdit* gehören, den Namen »Text« tragen und in der Eigenschaftskategorie *TLocalizableCategory* registriert sein, damit die Funktion *IsPropertyInCategory* den Wert *True* zurückgibt.

```
IsItThere := IsPropertyInCategory(TLocalizableCategory, TEdit, 'Text');
```

Die zweite Version verwendet als Vergleichskriterium eine Kombination aus dem Klassennamen der Eigentümerkomponente und dem Namen der Eigenschaft. Im folgenden Beispiel gibt *IsPropertyInCategory* den Wert *True* zurück, wenn die Eigenschaft zur Klasse *TEdit* gehört, den Namen »Text« trägt und sich in der Eigenschaftskategorie *TLocalizableCategory* befindet.

```
IsItThere := IsPropertyInCategory(TLocalizableCategory, 'TEdit', 'Text');
```

Komponenten in Packages compilieren

Nach dem Registrieren Ihrer Komponenten müssen Sie diese in Packages compilieren, damit sie in der IDE installiert werden können. Ein Package kann eine oder mehrere Komponenten und benutzerdefinierte Eigenschaftseditoren enthalten. Weitere Informationen über Packages finden Sie in Kapitel 9, »Packages und Komponenten«.

Informationen zur Erzeugung und Compilierung einer Komponente finden Sie unter »Packages erstellen und bearbeiten« auf Seite 9-8. Nehmen Sie die Quelltext-Units Ihrer Komponenten in die *Contains*-Liste des Package auf. Bestehen Abhängigkeiten von anderen Packages, fügen Sie diese der Liste Requires hinzu.

Informationen über das Installieren Ihrer Komponenten in der IDE finden Sie im Abschnitt »Komponenten-Packages installieren« auf Seite 9-6.

Fehlerbeseitigung für benutzerdefinierte Komponenten

Ein typisches Problem beim Registrieren und Installieren benutzerdefinierter Komponenten besteht darin, daß die Komponente nach erfolgreicher Installation des Packages nicht in der Komponentenliste angezeigt wird.

Nachstehend finden Sie die häufigsten Ursachen für dieses Problem:

- Fehlender Modifizierer PACKAGE in der Funktion *Register*.
- Fehlender Modifizierer PACKAGE in der Klasse.
- Fehlende Anweisung `#pragma package(smart_init)` in der C++-Quelltextdatei.
- Die Funktion *Register* befindet sich nicht mit demselben Namen wie das Quelltextmodul in einem *Namespace*.
- Register wird nicht erfolgreich exportiert. Verwenden Sie tdump für die BPL-Datei, um die exportierte Funktion zu suchen:

  ```
  tdump -ebpl mypack.bpl mypack.dmp
  ```

  Im Abschnitt exports des Ergebnisses sollte die exportierte Funktion *Register* im *Namespace* angezeigt werden.

Kapitel

39

Vorhandene Komponenten modifizieren

Das einfachste Verfahren, eine Komponente zu erzeugen, besteht darin, sie von einer anderen Komponente abzuleiten, die bereits einen Großteil der geforderten Merkmale besitzt. Im folgenden wird zur Illustration die Standard-Memofeldkomponente so abgewandelt, daß daraus ein Memofeld ohne voreingestellten Zeilenumbruch entsteht.

Das Memofeld besitzt eine Eigenschaft *WordWrap*, die mit dem Wert *True* initialisiert wird. Wenn Sie häufig Memofelder benötigen, die keinen Zeilenumbruch vornehmen, ist es sinnvoll, eine neue Memokomponente mit dem entsprechenden Verhalten zu erzeugen.

Hinweis Es kann Situationen geben, in denen es einfacher ist, eine *Komponentenschablone* zu verwenden, anstatt eine neue Klasse zu erzeugen.

Die Modifikation einer vorhandenen Komponente umfaßt nur zwei Schritte:

- Die Komponente erstellen und registrieren.
- Die Komponentenklasse ändern.

Die Komponente erstellen und registrieren

Die Erstellung einer Komponente beginnt immer auf dieselbe Weise: Sie legen eine Unit an, leiten eine Komponentenklasse ab, registrieren diese und installieren die Klasse in der Komponentenpalette. Dieser Prozeß wird unter »Eine neue Komponente erzeugen« auf Seite 31-9 beschrieben.

Für das vorliegende Beispiel halten Sie sich an das grundsätzliche Vorgehen, wobei folgende Schwerpunkte gesetzt werden sollen:

- Eine Komponenten-Unit namens *Memos* anlegen.

- Einen neuen Komponententyp *TWrapMemo* von *TMemo* ableiten.
- *TWrapMemo* auf der Registerkarte *Beispiele* der Komponentenpalette plazieren.
- Die daraus resultierende Unit sollte wie folgt aussehen:

```
unit Memos;
interface
uses
   SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
   Forms, StdCtrls;
type
   TWrapMemo = class(TMemo)
   end;
procedure Register;
implementation
procedure Register;
begin
   RegisterComponents('Beispiele', [TWrapMemo]);
end;
end.
```

Wenn Sie die neue Komponente nun compilieren und installieren, funktioniert sie exakt wie der Vorfahr *TMemo*. Im nächsten Abschnitt soll nun eine einfache Änderung vorgenommen werden.

Die Komponentenklasse ändern

Sobald eine neue Komponentenklasse erzeugt wurde, kann diese nahezu uneingeschränkt abgewandelt werden. Zu Demonstrationszwecken beschränken wir uns hier auf den Anfangswert einer Eigenschaft der Memokomponente. Dazu sind zwei kleinere Änderungen an der Komponentenklasse erforderlich:

- Der Konstruktor muß überschrieben werden.
- Der neue Standardwert für die Eigenschaft muß festgelegt werden.

Der Konstruktor setzt den Wert der Eigenschaft. Der Standardwert bestimmt, was in der Formulardatei (DFM) gespeichert wird. Es ist wichtig, daß die beiden oben genannten Schritte ausgeführt werden, weil Delphi nur solche Werte speichert, die von der Vorgabe abweichen.

Den Konstruktor überschreiben

Wenn eine Komponente während des Entwurfs in einem Formular plaziert oder zur Laufzeit generiert wird, setzt der Konstruktor die Eigenschaftswerte. Wenn eine Komponente aus einer Formulardatei geladen wird, setzt die Anwendung alle Eigenschaftswerte, die während des Entwurfs geändert wurden.

Hinweis Wenn Sie einen Konstruktor überschreiben, muß der neue Konstruktor vor allen anderen Operationen die geerbte Version des Konstruktors aufrufen. Weitere Informationen finden Sie unter »Methoden überschreiben« auf Seite 32-9.

Im vorliegenden Beispiel muß die neue Komponente den von *TMemo* geerbten Konstruktor so überschreiben, daß die Eigenschaft *WordWrap* mit *False* initialisiert wird. Zu diesem Zweck fügen Sie die Konstruktorüberschreibung in die Vorwärtsdeklaration ein und plazieren den neuen Konstruktor im Implementierungsabschnitt der Unit:

```
type
   TWrapMemo = class(TMemo)
   public { Konstruktoren sind immer als public deklariert }
    constructor Create(AOwner: TComponent); override; { Diese Syntax ist immer gleich }
   end;
...
constructor TWrapMemo.Create(AOwner: TComponent); { Folgt der Implementierung }
begin
   inherited Create(AOwner); { Diesen Schritt IMMER zuerst! }
   WordWrap := False; { Neuen Wert setzen }
end;
```

Nun kann die neue Komponente in der Komponentenpalette plaziert und dann in ein Formular eingefügt werden. Beachten Sie, wie die Eigenschaft *WordWrap* wunschgemäß mit dem Wert *False* initialisiert wird.

Wenn Sie den Anfangswert einer Eigenschaft ändern, sollten Sie diesen neuen Wert zum Standard machen. Falls Sie den vom Konstruktor gesetzten Wert nicht mit dem angegebenen Standardwert abgleichen, kann Delphi den korrekten Wert nicht speichern und wiederherstellen.

Den neuen Standardwert für die Eigenschaft festlegen

Wenn die Beschreibung eines Formulars in der zugehörigen Formulardatei hinterlegt ist, werden nur diejenigen Eigenschaftswerte gespeichert, die von den Standardwerten abweichen. Die Formulardateien werden dadurch kompakt gehalten. Wenn Sie eine Eigenschaft erzeugen oder den Standardwert ändern, sollten Sie die Deklaration der Eigenschaft entsprechend anpassen. Details hierzu finden Sie in Kapitel 38, »Komponenten zur Entwurfszeit verfügbar machen«.

Um den Standardwert einer Eigenschaft zu ändern, deklarieren Sie den Eigenschaftsnamen erneut und stellen Sie die Direktive *default* mit dem neuen Standardwert nach. Sie müssen nicht die gesamte Eigenschaft erneut deklarieren, es genügen Name und Standardwert.

Bezogen auf das Beispiel hieße das, die Eigenschaft *WordWrap* in einem **published**-Abschnitt der Objektdeklaration mit dem Standardwert *False* zu versehen:

```
type
   TWrapMemo = class(TMemo)
   ...
   published
    property WordWrap default False;
   end;
```

Allein die Festlegung eines neuen Standardwertes beeinflußt noch nicht die Funktionsweise der Komponente. Der Wert muß trotzdem noch im Konstruktor initialisiert

werden. Eine erneute Deklaration des Standardwertes stellt sicher, daß der Wert von *WordWrap* zum richtigen Zeitpunkt in die Formulardatei geschrieben wird.

Kapitel

40

Grafische Komponenten erzeugen

Ein grafisches Steuerelement ist eine sehr einfache Komponente. Da sie nie den Eingabefokus erhalten kann, benötigt sie auch kein Fenster-Handle. Der Benutzer kann die Komponente zwar mit der Maus bearbeiten, eine Tastaturschnittstelle gibt es jedoch nicht.

Die grafische Komponente, um die es in diesem Kapitel geht, ist *TShape* (Seite *Zusätzlich* der Komponentenpalette), die geometrische Figuren darstellen kann. Obwohl die Komponente, die hier erzeugt wird, mit *TShape* identisch ist, sollten Sie sie anders nennen, um Konflikte aufgrund doppelter Bezeichner zu vermeiden. In diesem Kapitel heißt die Komponente *TSampleShape*. Sie werden folgende Schritte durchführen, um die Komponente zu erzeugen:

- Die Komponente erzeugen und registrieren.
- Geerbte Eigenschaften als **published** deklarieren.
- Grafische Funktionen hinzufügen.

Die Komponente erzeugen und registrieren

Die ersten Schritte zu einer neuen Komponente sind immer gleich: Sie erzeugen eine Unit, leiten eine Komponentenklasse ab, registrieren sie, compilieren die Unit und installieren die Komponente in der Komponentenpalette. Dieser Vorgang wird in »Eine neue Komponente erzeugen« auf Seite 31-9 beschrieben.

In diesem Beispiel wird vorausgesetzt, daß Sie nicht den Komponenten-Experten verwenden, sondern die Komponente manuell erstellen.

- Geben Sie der Komponenten-Unit den Namen *Shapes*
- Leiten Sie von *TGraphicControl* den neuen Komponententyp *TSampleShape* ab.
- Registrieren Sie *TSampleShape* in der Seite *Beispiele* der Komponentenpalette.

Die erzeugte Unit sollte wie folgt aussehen:

```
unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
   TSampleShape = class(TGraphicControl)
   end;
procedure Register;
implementation
procedure Register;
begin
   RegisterComponent('Beispiele', [TSampleShape]);
end;
end.
```

Geerbte Eigenschaften als published deklarieren

Wenn Sie einen Komponententyp ableiten, können Sie entscheiden, welche der in der Vorfahrklasse als **protected** deklarierten Eigenschaften und Ereignisse Sie in der neuen Komponente zur Verfügung stellen wollen. *TGraphicControl* deklariert bereits alle Eigenschaften als **published**, die für grafische Komponenten typisch sind. Sie brauchen daher nur noch Reaktionen auf Mausereignisse und Drag&Drop-Funktionen bereitzustellen.

Wie geerbte Eigenschaften und Ereignisse als **published** deklariert werden, erfahren Sie in »Geerbte Eigenschaften als published deklarieren« auf Seite 33-3 und »Ereignisse sichtbar machen« auf Seite 34-6. In beiden Fällen werden einfach die Namen der Eigenschaften im **published**-Abschnitt der Klassendeklaration redeklariert.

Bei unserer neuen Komponente sind dies die drei Mausereignisse, die drei Drag&Drop-Ereignisse und die beiden Drag&Drop-Eigenschaften:

```
type
   TSampleShape = class(TGraphicControl)
   published
    property DragCursor;        { Drag&Drop-Eigenschaften }
    property DragMode;
    property OnDragDrop;        { Drag&Drop-Ereignisse }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { Mausereignisse }
    property OnMouseMove;
    property OnMouseUp;
   end;
```

Die Benutzer der Beispielkomponente können nun Maus- und Drag&Drop-Operationen durchführen.

Grafische Funktionen hinzufügen

Sobald Sie die Komponente deklariert und die gewünschten Eigenschaften veröffentlicht haben, können Sie ihr individuelle grafische Funktionen hinzufügen. Bei der Erzeugung grafischer Steuerelemente müssen Sie zwei Aufgaben durchführen:

1 Entscheiden, was gezeichnet werden soll.

2 Die Komponente zeichnen.

Außerdem werden Sie der Komponente einige Eigenschaften hinzufügen, mit denen Anwendungsentwickler das Erscheinungsbild der Komponente zur Entwurfszeit beeinflussen können.

Was soll gezeichnet werden?

Ein grafisches Steuerelement kann sein Erscheinungsbild in Abhängigkeit von dynamischen Bedingungen ändern (z. B. als Reaktion auf Benutzereingaben). Würde es immer gleich aussehen, wäre es kaum als Komponente geeignet. Wenn Sie eine statische Darstellung benötigen, importieren Sie ein Bild, statt eine Komponente zu verwenden.

Im allgemeinen hängt das Erscheinungsbild eines grafischen Steuerelements von den Werten einiger seiner Eigenschaften ab. Die Fortschrittsanzeige verfügt beispielsweise über Eigenschaften, die ihre Form und Ausrichtung und die Art der Anzeige (numerisch oder grafisch) festlegen. Bei unserer Beispielkomponente bestimmt eine Eigenschaft, welche geometrische Figur gezeichnet wird.

Um Ihr Steuerelement mit einer Eigenschaft zu erweitern, das die zu zeichnende Form festgelegt, fügen Sie eine Eigenschaft mit der Bezeichnung *Shape* hinzu. Hierzu müssen Sie folgende Schritte ausführen:

1 Deklarieren Sie den Eigenschaftstyp.

2 Deklarieren Sie die Eigenschaft.

3 Schreiben Sie die Implementierungsmethode.

Details dazu finden Sie in Kapitel 33, »Eigenschaften erstellen«.

Den Eigenschaftstyp deklarieren

Wenn Sie für eine Eigenschaft einen benutzerdefinierten Typ verwenden, müssen Sie diesen Typ vor der Klasse deklarieren, welche die Eigenschaft enthält. Der gebräuchlichste benutzerdefinierte Typ für Eigenschaften ist der Aufzählungstyp.

In unserem Beispiel benötigen wir einen Aufzählungstyp, der für jede mögliche Form, die das Steuerelement zeichnen kann, ein Element enthält.

Fügen Sie vor der Klassendeklaration von *TSampleShape* die folgende Typdeklaration ein:

```
type
   TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
```

```
 sstEllipse, sstCircle);
TSampleShape = class(TGraphicControl) { Diese Zeile stand bereits hier }
```

Mit diesem Typ können Sie nun in der Klasse eine neue Eigenschaft deklarieren.

Die Eigenschaft deklarieren

Wenn Sie eine neue Eigenschaft einführen, deklarieren Sie normalerweise ein **private**-Feld, das den Eigenschaftswert speichert, und Methoden zum Lesen und Schreiben dieses Wertes. Zum Lesen des Wertes brauchen Sie oft nicht einmal eine Methode.

Im Beispiel deklarieren Sie ein Feld, das die aktuelle Figur enthält, und eine Eigenschaft, die den Wert liest und ihn durch einen Methodenaufruf zuweist.

Fügen Sie *TSampleShape* folgende Deklarationen hinzu:

```
type
   TSampleShape = class(TGraphicControl)
   private
    FShape: TSampleShapeType;  { Feld für den Eigenschaftwert }
    procedure SetShape(Value: TSampleShapeType);
   published
    property Shape: TSampleShapeType read FShape write SetShape;
   end;
```

Nun brauchen Sie nur noch die Methode *SetShape* zu implementieren.

Die Implementierungsmethode schreiben

Wenn Sie im **read**- oder **write**-Abschnitt einer Eigenschaftsdeklaration eine Methode verwenden, statt direkt auf die gespeicherten Eigenschaftsdaten zuzugreifen, müssen Sie diese Methode implementieren.

Fügen Sie dem **implementation**-Abschnitt der Unit folgende Definition der Methode *SetShape* hinzu:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
   if FShape <> Value then { Ignorieren, wenn keine Änderung  }
   begin
    FShape := Value; { Neuen Wert speichern }
    Invalidate; { Erneutes Zeichnen mit der neuen Form }
   end;
end;
```

Konstruktor und Destruktor überschreiben

Um die voreingestellten Eigenschaftswerte zu ändern und die Objekte Ihrer Klasse zu initialisieren, müssen Sie den geerbten Konstruktor und Destruktor überschreiben. Vergessen Sie nicht, in beiden Fällen die geerbte Methode aufzurufen.

Voreingestellte Eigenschaftswerte überschreiben

Nachdem die Komponente in der Voreinstellung etwas klein ausfällt, sollten Sie im Konstruktor die anfängliche Breite und Höhe ändern. Details darüber finden Sie in Kapitel 39, »Vorhandene Komponenten modifizieren«.

In diesem Beispiel weisen Sie der Komponente eine Breite und Höhe von je 65 Pixel zu.

Fügen Sie der Deklaration der Komponentenklasse den überschriebenen Konstruktor hinzu:

```
type
   TSampleShape = class(TGraphicControl)
   public { Konstruktoren sind immer public }
    constructor Create(AOwner: TComponent); override { override-Direktive nicht vergessen }
   end;
```

1 Redeklarieren Sie die Eigenschaften *Width* und *Height*, und weisen Sie ihnen die neuen Standardwerte zu:

```
type
   TSampleShape = class(TGraphicControl)
   ...
   published
    property Height default 65;
    property Width default 65;
   end;
```

2 Schreiben Sie im **implementation**-Abschnitt der Unit den Konstruktor:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);  { Immer den geerbten Konstruktor aufrufen }
   Width := 65;
   Height := 65;
end;
```

Stift und Pinsel als published deklarieren

In der Voreinstellung hat eine Zeichenfläche einen dünnen schwarzen Stift und einen Pinsel mit Flächenfarbe. Damit die Entwickler Stift und Pinsel ändern können, müssen Sie Klassen bereitstellen, die zur Entwurfszeit manipuliert werden können. Diese Klassen werden während der Zeichenoperation in die Zeichenfläche kopiert. Klassen wie diese zusätzlichen Stifte oder Pinsel nennt man *untergeordnete Klassen*, weil die Komponente ihnen übergeordnet ist und die Objekte dieser Klassen selbst erzeugen und freigeben muß.

Wenn Sie untergeordnete Klassen verwenden, müssen Sie

1 die Klassenfelder deklarieren;

2 die Zugriffseigenschaften deklarieren;

3 untergeordnete Klassen initialisieren;

4 Eigenschaften untergeordneter Klassen setzen.

Die Klassenfelder deklarieren

Für jede Klasse, die einer Komponente untergeordnet ist, muß ein Klassenfeld deklariert werden. Dieses Feld garantiert, daß der Komponente immer ein Zeiger zur Verfügung steht, über den sie die untergeordnete Klasse freigeben kann, bevor sie sich selbst freigibt. Im allgemeinen initialisiert eine Komponente ihre untergeordneten Objekte im Konstruktor und gibt sie im Destruktor frei.

Felder für untergeordnete Objekte werden fast immer als **private** deklariert. Wenn Anwendungen (oder andere Komponenten) Zugriff auf solche Objekte benötigen, können Sie zu diesem Zweck Eigenschaften einführen, die Sie als **published** oder **public** deklarieren.

Fügen Sie der Beispielkomponente nun Felder für einen Stift und einen Pinsel hinzu:

```
type
   TSampleShape = class(TGraphicControl)
   private { Felder sind fast immer als private deklariert }
    FPen: TPen; { Ein Feld für das Stift-Objekt }
    FBrush: TBrush;  { Ein Feld für das Pinsel-Objekt }
    ...
   end;
```

Die Zugriffseigenschaften deklarieren

Der Zugriff auf die untergeordneten Objekte einer Komponente wird ermöglicht, indem Sie Eigenschaften vom Typ der Objekte deklarieren. Dadurch können Entwickler sowohl zur Entwurfs- als auch zur Laufzeit auf die Objekte zugreifen. Im Normalfall erfolgt das Lesen der Eigenschaft einfach durch eine Referenz auf das Klassenfeld. Schreibzugriffe werden dagegen meist in Form von Methodenaufrufen durchgeführt. Die Komponente erhält dadurch die Möglichkeit, auf Änderungen am untergeordneten Objekt zu reagieren.

Fügen Sie der Beispielkomponente nun die Eigenschaften hinzu, über die auf die Felder Pen und Brush zugegriffen werden kann. Deklarieren Sie außerdem die Methoden, die auf Änderungen an diesen Eigenschaften reagieren:

```
type
   TSampleShape = class(TGraphicControl)
   ...
   private { Diese Methoden sollten als private deklariert werden }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
   published { Zur Entwurfszeit verfügbar }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
   end;
```

Schreiben Sie im **implementation**-Abschnitt der Unit die Methoden *SetBrush* und *SetPen*:

```
procedure TSampleShape.SetBrush(Value: TBrush);
```

```
begin
   FBrush.Assign(Value); { Bestehenden Pinsel durch Parameter ersetzen }
end;
procedure TSampleShape.SetPen(Value: TPen);
begin
   FPen.Assign(Value); { Bestehenden Stift durch Parameter ersetzen }
end;
```

Die direkte Zuweisung des Inhalts von *Value* an *FBrush* in der folgenden Form:

```
FBrush := Value;
```

würde den internen Zeiger auf *FBrush* überschreiben, zu Speicherverlust führen und eine Reihe von Verwaltungsproblemen verursachen.

Untergeordnete Klassen initialisieren

Wenn Sie Ihrer Komponente Klassen hinzufügen, muß sie der Konstruktor initialisieren, damit der Benutzer zur Laufzeit mit den Objekten interagieren kann. Der Destruktor der Komponente muß dagegen die untergeordneten Objekte freigeben, bevor er die Komponente selbst freigibt.

Da Sie der Beispielkomponente einen Stift und einen Pinsel hinzugefügt haben, müssen diese Objekte im Konstruktor initialisiert und im Destruktor freigegeben werden.

1 Erzeugen Sie Stift und Pinsel im Konstruktor:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
   inherited Create(AOwner); { Immer den geerbten Konstruktor aufrufen }
   Width := 65;
   Height := 65;
   FPen := TPen.Create; { Stift erzeugen }
   FBrush := TBrush.Create; { Pinsel erzeugen }
end;
```

2 Fügen Sie der Deklaration der Komponentenklasse den überschriebenen Destruktor hinzu:

```
type
   TSampleShape = class(TGraphicControl)
   public { Destruktoren sind immer public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override; { override nicht vergessen }
   end;
```

3 Schreiben Sie im **implementation**-Abschnitt der Unit den neuen Destruktor:

```
destructor TSampleShape.Destroy;
begin
   FPen.Free; { Stift freigeben }
   FBrush.Free; { Pinsel freigeben }
   inherited Destroy; { Immer auch den geerbten Destruktor aufrufen }
end;
```

Eigenschaften untergeordneter Klassen setzen

Im letzten Schritt bei der Erzeugung der Stift- und Pinselklassen müssen Sie dafür sorgen, daß sich die Beispielkomponente nach jeder Änderung am Stift oder am Pinsel neu zeichnet. In beiden Klassen gibt es das Ereignis *OnChange*. Sie können also in *TSampleShape* einfach eine Methode schreiben und sie den beiden *OnChange*-Ereignissen zuweisen.

Fügen Sie der Beispielkomponente die folgenden Methoden hinzu und aktualisieren Sie den Konstruktor so, daß er den genannten Ereignissen die neue Methode zuweist:

```
type
   TSampleShape = class(TGraphicControl)
   published
    procedure StyleChanged(Sender: TObject);
   end;
...
implementation
...
constructor TSampleShape.Create(AOwner: TComponent);
begin
   inherited Create(AOwner); { Immer den geerbten Konstruktor aufrufen }
   Width := 65;
   Height := 65;
   FPen := TPen.Create; { Stift erzeugen }
   FPen.OnChange := StyleChanged; { Dem OnChange-Ereignis die Methode zuweisen }
   FBrush := TBrush.Create; { Pinsel erzeugen }
   FBrush.OnChange := StyleChanged; { Dem OnChange-Ereignis die Methode zuweisen }
end;
procedure TSampleShape.StyleChanged(Sender: TObject);
begin
   Invalidate(True); { Komponente löschen und neu zeichnen }
end;
```

Die Komponente zeichnet sich nun nach jeder Änderung an einer der Eigenschaften *Pen* oder *Brush* neu.

Die Komponente zeichnen

Das wichtigste an einer grafischen Komponente ist die Art und Weise, wie sie sich selbst am Bildschirm zeichnet. Der abstrakte Typ *TGraphicControl* definiert die Methode *Paint*, die Sie überschreiben können, um die Komponente nach Ihren eigenen Vorstellungen darzustellen.

Bei der Beispielkomponente muß die Methode *Paint* verschiedene Aktionen durchführen:

- Sie muß den vom Benutzer gewählten Stift und Pinsel verwenden.
- Sie muß die gewählte Figur zeichnen.
- Sie muß die Koordinaten so zuweisen, daß Rechtecke und Kreise dieselbe Breite und Höhe verwenden.

So überschreiben Sie die Methode *Paint*:

1 Fügen Sie *Paint* der Deklaration der Komponente hinzu.

2 Schreiben Sie die *Paint*-Methode im **implementation**-Abschnitt der Unit.

Schritt 1 sieht folgendermaßen aus:

```
type
   TSampleShape = class(TGraphicControl)
   ...
   protected
    procedure Paint; override;
   ...
   end;
Und hier Schritt 2:
procedure TSampleShape.Paint;
begin
   with Canvas do
   begin
    Pen := FPen; { Stift der Komponente kopieren }
    Brush := FBrush; { Pinsel der Komponente kopieren }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height); { Rechtecke und Quadrate zeichnen }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { Figuren mit abgerundeten
                                                                     Ecken }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height); { Runde Figuren }
    end;
   end;
end;
```

Paint wird immer dann aufgerufen, wenn die Komponente ihre Anzeige am Bildschirm aktualisieren muß. Die Aktualisierung wird von Windows veranlaßt, wenn ein Steuerelement zum erstenmal gezeichnet wird oder wenn es von einem anderen Fenster verdeckt war und nun wieder sichtbar ist. Sie können das Neuzeichnen auch explizit durch einen Aufruf von *Invalidate* veranlassen, wie es beispielsweise die Methode *StyleChanged* tut.

Letzte Korrekturen

Die Standardkomponente *TShape* hat der Beispielkomponente nur noch eines voraus: Sie kann neben Rechtecken und Ellipsen auch Quadrate und Kreise zeichnen. Um dies zu realisieren, benötigen Sie einen Quelltext, der die kürzesten Seitenlinien ermittelt und das Bild zentriert.

Hier eine verbesserte Version der Methode *Paint*, die auch Quadrate und Kreise zeichnet:

```
procedure TSampleShape.Paint;
var
   X, Y, W, H, S: Integer;
begin
     with Canvas do
```

```
    begin
     Pen := FPen; { Stift der Komponente kopieren }
     Brush := FBrush; { Pinsel der Komponente kopieren }
     W := Width; { Breite der Komponente verwenden }
     H := Height; { Höhe der Komponente verwenden }
     if W < H then S := W else S := H; { Kleinste Werte speichern }
     case FShape of {  Höhe, Breite und Position anpassen }
       sstRectangle, sstRoundRect, sstEllipse:
         begin
           X := 0; { Ursprung für diese Figuren oben links }
           Y := 0;
         end;
       sstSquare, sstRoundSquare, sstCircle:
         begin
           X := (W - S) div 2; { Diese Figuren horizontal ... }
           Y := (H - S) div 2; { ... und vertikal zentrieren }
           W := S; { Kleinste Werte für Breite ... }
           H := S; { ... und Höhe verwenden }
         end;
     end;
     case FShape of
       sstRectangle, sstSquare:
         Rectangle(X, Y, X + W, Y + H); { Rechtecke und Quadrate zeichnen }
       sstRoundRect, sstRoundSquare:
         RoundRect(X, Y, X + W, Y + H, S div 4, S div 4); { Figuren mit abgerundeten Ecken }
       sstCircle, sstEllipse:
         Ellipse(X, Y, X + W, Y + H); { Runde Figuren }
     end;
    end;
end;
```

Kapitel

41

Gitter anpassen

Delphi stellt abstrakte Komponenten bereit, die Sie als Grundlage für eigene Komponenten verwenden können. Die wichtigsten dieser Komponenten sind Gitter und Listenfelder. Das vorliegende Kapitel zeigt, wie Sie mit Hilfe der grundlegenden Gitterkomponente *TCustomGrid* einen Einmonatskalender entwickeln können.

Die Erstellung des Kalenders umfaßt folgende Schritte:

- Die Komponente erzeugen und registrieren
- Geerbte Eigenschaften als **published** deklarieren
- Initialisierungswerte ändern
- Die Größe der Zellen ändern
- Die Zellen füllen
- Monats- und Jahreswechsel implementieren
- Durch die Tage navigieren

Die resultierende Komponente ähnelt der Komponente *TCalendar* auf der Seite *Beispiele* der Komponentenpalette.

Die Komponente erzeugen und registrieren

Die Erstellung einer Komponente beginnt immer auf die gleiche Weise und erfordert folgende Schritte: Erstellen einer Unit, Ableiten einer Komponentenklasse, Registrieren und Compilieren der Klasse, Installieren der Klasse in der Komponentenpalette. Dieser Vorgang wird im Abschnitt »Eine neue Komponente erzeugen« auf Seite 31-9 ausführlich erläutert.

Im vorliegenden Beispiel gehen Sie folgendermaßen vor:

- Nennen Sie die Unit der Komponente *CalSamp*.

- Leiten Sie von *TCustomGrid* einen neuen Komponententyp mit der Bezeichnung *TSampleCalendar* ab.
- Registrieren Sie *TSampleCalendar* auf der Seite *Beispiele* der Komponentenpalette.

Die resultierende Unit sollte folgendermaßen aussehen:

```
unit CalSamp;
interface
uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
   TSampleCalendar = class(TCustomGrid)
   end;
procedure Register;
implementation
procedure Register;
begin
   RegisterComponents('Beispiele', [TSampleCalendar]);
end;
end.
```

Wenn Sie die Kalenderkomponente jetzt installieren, gehört sie zur Seite *Beispiele*. Momentan sind nur grundlegende Eigenschaften verfügbar. Der nächste Schritt besteht darin, den Benutzern des Kalenders speziellere Eigenschaften zur Verfügung zu stellen.

Hinweis Obwohl die soeben compilierte Kalenderkomponente bereits installiert werden kann, sollten Sie noch nicht versuchen, sie in einem Formular zu plazieren. Zur Komponente *TCustomGrid* gehört nämlich die abstrakte Methode *DrawCell*, die redeklariert werden muß, bevor Instanzen der Klasse erzeugt werden. Das Überschreiben der Methode *DrawCell* wird an späterer Stelle im Abschnitt »Die Zellen füllen« erläutert.

Geerbte Eigenschaften als published deklarieren

Die abstrakte Gitterkomponente *TCustomGrid* stellt eine große Anzahl von Eigenschaften bereit, die als **protected** deklariert sind. Sie können festlegen, welche dieser Eigenschaften für die Benutzer des Kalenders verfügbar sein sollen.

Zu diesem Zweck müssen Sie die betreffenden Eigenschaften im **published**-Abschnitt der Komponentendeklaration redeklarieren.

Für den Kalender werden folgende Eigenschaften und Ereignisse redeklariert:

```
type
   TSampleCalendar = class(TCustomGrid)
   published
    property Align;  { Eigenschaften als published deklarieren }
    property BorderStyle;
    property Color;
    property Ctl3D;
    property Font;
    property GridLineWidth;
    property ParentColor;
```

```
    property ParentFont;
    property OnClick;  { Ereignisse als published deklarieren}
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;
```

Es gibt noch weitere Eigenschaften, die Sie als **published** deklarieren könnten, die aber für einen Kalender keine Bedeutung haben. Dazu gehört z. B. die Eigenschaft *Options*, über die der Benutzer auswählen kann, welche Art von Gitterlinien gezeichnet werden soll.

Wenn Sie die Komponentenpalette neu aufbauen und den Kalender testen, werden Sie feststellen, daß viele weitere, voll funktionsfähige Eigenschaften und Ereignisse verfügbar sind. Sie können nun damit beginnen, dem Kalender neue, selbstdefinierte Fähigkeiten hinzuzufügen.

Die Initialisierungswerte ändern

Ein Kalender besteht hauptsächlich aus einem Gitter mit einer festgelegten Anzahl von Zeilen und Spalten, wobei aber nicht alle Zellen einen Datumswert enthalten müssen. Da es äußerst unwahrscheinlich ist, daß die Benutzer des Kalenders etwas anderes als sieben Tage pro Woche anzeigen wollen, wurden die Gittereigenschaften *ColCount* und *RowCount* nicht als **published** deklariert. Die Initialisierungswerte dieser Eigenschaften müssen aber trotzdem festgelegt werden, damit die Woche immer sieben Tage hat.

Um die Initialisierungswerte der Komponenteneigenschaften zu ändern, müssen Sie den Konstruktor überschreiben. Der Konstruktor muß virtuell sein.

Denken Sie daran, daß der Konstruktor dem **public**-Abschnitt der Objektdeklaration der Komponente hinzugefügt werden muß. Anschließend definieren Sie den neuen Konstruktor im **implementation**-Abschnitt der Komponenten-Unit. Die erste Anweisung im neuen Konstruktor sollte immer ein Aufruf des geerbten Konstruktors sein.

```
type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner: TComponent); override;
  ⋮
  end;
⋮
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { Geerbten Konstruktor aufrufen  }
  ColCount := 7; { Immer sieben Tage pro Woche }
  RowCount := 7; { Immer sechs Wochen plus Überschriften }
  FixedCols := 0; { Keine Zeilenbeschriftungen }
```

```
    FixedRows := 1; { Eine Zeile für die Tagesnamen }
    ScrollBars := ssNone; { Kein Bildlauf erforderlich }
    Options := Options - [goRangeSelect] + [goDrawFocusSelected];  {Bereichsauswahl
                                                                    deaktivieren}
end;
```

Der Kalender verfügt nun über sieben Spalten und über sieben Zeilen. Die oberste Zeile ist nicht beweglich (d. h. sie ist fest verankert und wird beim Bildlauf nicht verschoben).

Die Größe der Zellen ändern

Wenn die Größe eines Fensters oder Steuerelements von einem Benutzer oder einer Anwendung geändert wird, sendet Windows eine WM_SIZE-Botschaft. Das betreffende Fenster oder Steuerelement kann dadurch Einstellungen vornehmen, die für das spätere Zeichnen in der neuen Größe erforderlich sind. Ihre Komponente kann auf die Botschaft mit einer Änderung der Zellgröße reagieren und dafür sorgen, daß weiterhin alle Zellen innerhalb des Steuerelements Platz finden. Zu diesem Zweck benötigt die Komponente eine entsprechende Botschaftsbehandlungsmethode.

Details über Botschaftsbehandlungsmethoden finden Sie in »Neue Routinen zur Botschaftsbehandlung erstellen« auf Seite 37-5.

Im vorliegenden Beispiel muß das Kalender-Steuerelement auf die WM_SIZE-Botschaft reagieren. Dazu fügen Sie dem Steuerelement eine als **protected** deklarierte Methode mit der Bezeichnung *WMSize* hinzu, die mit WM_SIZE indiziert ist. Diese Methode muß die korrekte Zellgröße berechnen, die für eine vollständige Anzeige in der neuen Größe erforderlich ist:

```
type
   TSampleCalendar = class(TCustomGrid)
   protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
   ⋮
   end;
⋮
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
   GridLines: Integer; { Temporäre lokale Variable }
begin
   GridLines := 6 * GridLineWidth; { Gesamtgröße aller Zeilen berechnen }
   DefaultColWidth := (Message.Width - GridLines) div 7; { Neue Standardbreite }
   DefaultRowHeight := (Message.Height - GridLines) div 7; { Neue Standardhöhe }
end;
```

Wenn nun die Größe des Kalenders verändert wird, zeigt er alle Zellen in der maximalen Größe, die in das Steuerelement paßt, an.

Die Zellen füllen

Ein Gitter wird immer zellenweise gefüllt. Für unseren Beispielkalender muß deshalb berechnet werden, welches Datum (falls vorhanden) in welche Zelle gehört. Das standardmäßige Zeichnen von Gitterzellen übernimmt eine virtuelle Methode mit dem Namen *DrawCell*.

Zum Einfügen des Inhalts der Gitterzellen muß die Methode *DrawCell* überschrieben werden.

Am einfachsten lassen sich die Zellen der Überschriften füllen. Die Laufzeitbibliothek deklariert das Array *ShortDayNames*, das die Kurznamen der Wochentage enthält:

```
type
   TSampleCalendar = class(TCustomGrid)
   protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
   end;
⋮
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
   AState: TGridDrawState);
begin
   if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]); { RTL-String verwenden }
end;
```

Das Datum festlegen

Damit der Kalender sinnvoll eingesetzt werden kann, müssen Benutzer oder Anwendungen den Tag, den Monat und das Jahr festlegen können. Delphi speichert Datums- und Zeitangaben in Variablen vom Typ *TDateTime*. Dieser Typ enthält eine kodierte numerische Darstellung des Datums bzw. der Uhrzeit und eignet sich zwar gut für programminterne Manipulationen, ist aber für den Anwender zu unhandlich.

Sie werden das Datum also einerseits in kodierter Form speichern, um den Zugriff zur Laufzeit zu ermöglichen, andererseits aber auch die Eigenschaften *Day, Month* und *Year* verwenden, damit die Benutzer des Kalenders den Tag, den Monat und das Jahr bereits zur Entwurfszeit festlegen können.

Die Datumsaufzeichnung im Kalender setzt sich aus folgenden Vorgängen zusammen:

- Das interne Datum speichern
- Auf den Tag, den Monat und das Jahr zugreifen
- Die Anzahl der Tage im Monat generieren
- Den aktuellen Tag auswählen

Das interne Datum speichern

Zum Speichern des Datums benötigen Sie ein als **private** deklariertes Feld, das den Datumswert enthält, und eine Laufzeiteigenschaft, die den Zugriff auf diesen Wert ermöglicht.

Um dem Kalender das interne Datum hinzuzufügen, sind drei Schritte erforderlich:

1 Deklarieren Sie ein Feld als **private**, das das Datum aufnimmt:

```
type
   TSampleCalendar = class(TCustomGrid)
   private
    FDate: TDateTime;
   ⋮
```

2 Initialisieren Sie dieses Feld im Konstruktor:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
   inherited Create(AOwner); { Existiert bereits }
   ⋮ { hier folgen weitere Initialisierungen }
   FDate := Date; { Aktuelles Datum mit RTL-Funktion ermitteln }
end;
```

3 Deklarieren Sie eine Laufzeiteigenschaft, die den Zugriff auf das kodierte Datum ermöglicht.

Das Datum wird von einer Methode gesetzt, da ja auch die Anzeige auf dem Bildschirm aktualisiert werden muß:

```
type
   TSampleCalendar = class(TCustomGrid)
   private
    procedure SetCalendarDate(Value: TDateTime);
   public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
   ⋮
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
   FDate := Value; { Neuen Datumswert festlegen }
   Refresh; { Anzeige auf dem Bildschirm aktualisieren }
end;
```

Auf den Tag, den Monat und das Jahr zugreifen

Das kodierte Datum ist zwar für den internen Gebrauch sehr nützlich, der Anwender arbeitet aber lieber mit den üblichen Tages-, Monats- und Jahresangaben. Indem Sie Eigenschaften erzeugen, ermöglichen Sie einen alternativen Zugriff auf die codiert gespeicherten Datumswerte.

Da jedes Datumselement (Tag, Monat und Jahr) ein Integer ist und jede Zuweisung die Umwandlung des kodierten Datums erfordert, wäre es sinnvoll, die entsprechenden Methoden für alle drei Eigenschaften gemeinsam zu implementieren. Sie schreiben also jeweils zwei Methoden, eine zum Lesen und eine zum Schreiben. Diese Me-

thoden verwenden Sie dann, um die Werte aller drei Eigenschaften abzurufen bzw. zu setzen.

Gehen Sie folgendermaßen vor, um zur Entwurfszeit den Zugriff auf den Tag, den Monat und das Jahr zu ermöglichen:

1 Deklarieren Sie drei Eigenschaften, und weisen Sie jeder eine eindeutige Indexnummer zu:

```
type
   TSampleCalendar = class(TCustomGrid)
   public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
   ⋮
```

2 Deklarieren und schreiben Sie die Implementierungsmethoden. Weisen Sie dabei je nach Index ein bestimmtes Element zu:

```
type
   TSampleCalendar = class(TCustomGrid)
   private
    function GetDateElement(Index: Integer): Integer; { Beachten Sie den Parameter Index }
    procedure SetDateElement(Index: Integer; Value: Integer);
   ⋮
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
   AYear, AMonth, ADay: Word;
begin
   DecodeDate(FDate, AYear, AMonth, ADay); { Kodiertes Datum in Elemente zerlegen }
   case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
   end;
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
   AYear, AMonth, ADay: Word;
begin
   if Value > 0 then { Alle Elemente müssen positiv sein }
   begin
    DecodeDate(FDate, AYear, AMonth, ADay); { Aktuelle Datumselemente ermitteln }
    case Index of { Neues Element in Abgängigkeit vom Index festlegen }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay); { Geändertes Datum kodieren }
    Refresh; { Anzeige des Kalenders aktualisieren }
   end;
end;
```

Der Tag, der Monat und das Jahr können nun sowohl zur Entwurfszeit mit Hilfe des Objektinspektors als auch zur Laufzeit durch entsprechenden Quelltext festgelegt werden. Zwar existiert immer noch kein Quelltext zur Darstellung der Datumswerte in den Zellen, über die benötigten Daten verfügen Sie jedoch bereits.

Die Anzahl der Tage im Monat generieren

Beim Füllen des Kalenders mit Werten gibt es einiges zu beachten. Die Anzahl der Tage im Monat hängt vom jeweiligen Monat ab. Beim Monat Februar ist wichtig, ob es sich um ein Schaltjahr handelt. Außerdem beginnen die Monate an unterschiedlichen Wochentagen, was vom jeweiligen Monat bzw. Jahr abhängt. Mit der Funktion *IsLeapYear* kann ermittelt werden, ob es sich um ein Schaltjahr handelt. Das Array *MonthDays* in der Unit-Datei SysUtils liefert die Anzahl der Monatstage.

Nachdem nun die Tage pro Monat und die Schaltjahre verfügbar sind, kann berechnet werden, in welche Zellen die einzelnen Datumswerte eingefügt werden müssen. Die Berechnung basiert auf dem Wochentag, mit dem der Monat beginnt.

Wir benötigen nun eine Zahl, die für jede Zelle die Differenz zum Monatsersten angibt. Am sinnvollsten ist es, diese Zahl nur einmal beim Wechsel des Monats bzw. des Jahres zu berechnen und später auf sie Bezug zu nehmen. Der Wert wird in einem Klassenfeld gespeichert, das bei jeder Datumsänderung aktualisiert wird.

Zum Einfügen der Tage in die korrekten Zellen sind folgende Schritte erforderlich:

1 Fügen Sie dem Objekt ein Feld für die Differenz zum Monatsersten hinzu. Außerdem benötigen Sie eine Methode, die den Wert dieses Feldes aktualisiert:

```
type
   TSampleCalendar = class(TCustomGrid)
   private
    FMonthOffset: Integer; { Speichert die Differenz zum Monatsersten }
   ⋮
   protected
    procedure UpdateCalendar; virtual; { Eigenschaft für den Zugriff auf FMonthOffset }
   end;
⋮
procedure TSampleCalendar.UpdateCalendar;
var
   AYear, AMonth, ADay: Word;
   FirstDate: TDateTime; { Datum des ersten Tages im Monat }
begin
   if FDate <> 0 then { Bei gültigem Datum nur die Differenz berechnen }
   begin
    DecodeDate(FDate, AYear, AMonth, ADay); { Elemente des Datums ermitteln }
    FirstDate := EncodeDate(AYear, AMonth, 1); { Daten für den Monatsersten }
    FMonthOffset := 2 - DayOfWeek(FirstDate); { Differenz auf das Gitter übertragen }
   end;
   Refresh; { Anzeige immer aktualisieren }
end;
```

2 Fügen Sie dem Konstruktor und den Methoden *SetCalendarDate* und *SetDateElement* Anweisungen hinzu, die bei jeder Änderung des Datums die neue Aktualisierungsmethode aufrufen:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);                                    { existiert bereits }
   ⋮ { Weitere Initialisierungen }
   UpdateCalendar; { Korrekte Differenz zum Monatsersten festlegen }
end;
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
     FDate := Value; { War bereits vorhanden }
   UpdateCalendar; { Hat vorher Refresh aufgerufen }
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
   ⋮
    FDate := EncodeDate(AYear, AMonth, ADay); { Geändertes Datum kodieren }
    UpdateCalendar; { Hat vorher Refresh aufgerufen }
   end;
end;
```

3 Fügen Sie dem Kalender eine Methode hinzu, die die Anzahl der Tage zurückgibt, wenn die Zeilen- und Spaltenkoordinaten einer Zelle an sie übergeben werden:

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
   Result := FMonthOffset + ACol + (ARow - 1) * 7; { Tag für diese Zelle berechnen }
   if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1; { Wenn ungültig, -1 zurückgeben }
end;
```

Vergessen Sie nicht, der Typdeklaration die Deklaration von *DayNum* hinzuzufügen.

4 Nachdem nun berechnet werden kann, in welche Zellen die Datumswerte eingefügt werden müssen, können Sie die Methode *DrawCell* entsprechend aktualisieren:

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
   TheText: string;
   TempDay: Integer;
begin
   if ARow = 0 then { In der Kopfzeile ...}
    TheText := ShortDayNames[ACol + 1] { nur den Tagesnamen anzeigen }
   else begin
    TheText := ''; { Standardmäßig ist eine Zelle leer }
    TempDay := DayNum(ACol, ARow); { Zahl für diese Zelle ermitteln }
    if TempDay <> -1 then TheText := IntToStr(TempDay); { Wenn gültig, die Zahl anzeigen }
   end;
   with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
```

Wenn der Kalender nun neu installiert und in ein Formular eingefügt wird, zeigt er die korrekten Informationen für den aktuellen Monat an.

Den aktuellen Tag auswählen

Nachdem die Kalenderzellen mit Werten gefüllt sind, wäre es sinnvoll, die Zelle zu markieren, die den aktuellen Tag enthält. Standardmäßig ist immer die linke obere Zelle markiert. Die Eigenschaften *Row* und *Column* müssen daher sowohl bei der ersten Anzeige des Kalenders als auch bei jeder Änderung des Datums gesetzt werden.

Um die Markierung auf den aktuellen Tag zu setzen, ist eine Änderung der Methode *UpdateCalendar* erforderlich. Die Eigenschaften *Row* und *Column* müssen vor dem Aufruf von *Refresh* gesetzt werden:

```
procedure TSampleCalendar.UpdateCalendar;
begin
   if FDate <> 0 then
   begin
    ⋮ { Anweisungen zum Setzen von FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
   end;
   Refresh; { ist bereits vorhanden }
end;
```

Die vorher durch eine Decodierung des Datums festgelegte Variable *ADay* wird hier erneut verwendet.

Durch Monate und Jahre navigieren

Eigenschaften eignen sich besonders zur Entwurfszeit hervorragend für die Manipulation von Komponenten. Für häufig wiederkehrende Aktionen, die untrennbar mit einer Komponente verknüpft sind und zudem mehrere Eigenschaften betreffen, macht es jedoch Sinn, eigene Methoden zu implementieren. Bei einem Kalender ist eine solche Aktion der Wechsel zum nächsten Monat. Eine Funktion, die einen Monats- oder Jahreswechsel durchführt, läßt sich auf einfache Weise in die Komponente integrieren und erleichtert dem Entwickler, der die Komponente einsetzt, die Arbeit.

Einen Nachteil hat die Kapselung gängiger Aktionen in Methoden: Methoden sind nur zur Laufzeit verfügbar. Zur Entwurfszeit kommt es aber ohnehin selten vor, daß sich Manipulationen an Objekten ständig wiederholen.

Fügen Sie nun dem Beispielkalender die folgenden Methoden hinzu, die den Monats- und Jahreswechsel durchführen. Jede dieser Methoden verwendet die Funktion *IncMonth* auf etwas andere Weise, um *CalendarDate* in Schritten von einem Jahr oder einem Monat zu inkrementieren oder zu dekrementieren. Nach der Inkrementierung bzw. Dekrementierung von *CalendarDate* muß der Datumswert decodiert werden, um die Eigenschaften *Year*, *Month* und *Day* mit den entsprechenden neuen Werten zu füllen:

```
procedure TCalendar.NextMonth;
begin
   DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;
procedure TCalendar.PrevMonth;
```

```
begin
   DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;
procedure TCalendar.NextYear;
begin
   DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;
procedure TCalendar.PrevYear;
begin
   DecodeDate(CalendarDate, -12), Year, Month, Day);
end;
```

Vergessen Sie nicht, die Deklarationen der neuen Methoden in die Klassendeklaration aufzunehmen.

Bei der Verwendung der Kalenderkomponente in einer Anwendung läßt sich nun das Navigieren durch die Monate und Jahre sehr einfach implementieren.

Durch die Tage navigieren

Innerhalb eines bestimmten Monats gibt es zwei offensichtliche Möglichkeiten, durch die Tage des Kalenders zu blättern: die Pfeiltasten und die Maus. Die Standard-Gitterkomponente behandelt beide Vorgehensweisen als Mausklicks, d. h. das Drücken einer Pfeiltaste wird als Klick auf eine benachbarte Zelle interpretiert.

Die Bewegung durch die einzelnen Kalendertage besteht aus folgenden Vorgängen:

- Die Markierung bewegen
- Das Ereignis *OnChange* hinzufügen
- Bewegung zu leeren Zellen verhindern

Die Markierung bewegen

Im geerbten Verhalten eines Gitters wird die Markierung bewegt, wenn eine Pfeiltaste gedrückt oder mit der Maus auf eine Zelle geklickt wird. Dieses Standardverhalten muß für den Kalender geändert werden.

Um Bewegungen innerhalb des Kalenders zu bearbeiten, überschreiben Sie die Methode *Click* des Gitters.

Beim Überschreiben einer Methode, die auf Benutzeraktionen reagiert (wie z. B. *Click*), ist fast immer ein Aufruf der geerbten Methode erforderlich. Dadurch ist sichergestellt, daß das geerbte Verhalten nicht verlorengeht.

Nachstehend sehen Sie die überschriebene Methode *Click* für das Kalendergitter. Vergessen Sie nicht, der Deklaration von *TSampleCalendar* die Deklaration von *Click* (einschließlich der Direktive **override**) hinzuzufügen.

```
procedure TSampleCalendar.Click;
var
   TempDay: Integer;
```

```
begin
   inherited Click; { Aufruf der geerbten Methode nicht vergessen! }
   TempDay := DayNum(Col, Row); { Tagesnummer für die Zelle ermitteln }
   if TempDay <> -1 then Day := TempDay; { Wenn gültig, Tag ändern }
end;
```

Das Ereignis OnChange hinzufügen

Nachdem die Kalenderbenutzer jetzt das Datum im Kalender ändern können, müssen Sie der Anwendung die Möglichkeit geben, auf diese Änderung zu reagieren.

Um *TSampleCalendar* das Ereignis *OnChange* hinzuzufügen, gehen Sie folgendermaßen vor:

1 Deklarieren Sie das Ereignis, ein Feld zum Speichern des Ereignisses und eine dynamische Methode, die das Ereignis aufruft:

```
type
   TSampleCalendar = class(TCustomGrid)
   private
    FOnChange: TNotifyEvent;
   protected
    procedure Change; dynamic;
   ⋮
   published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
   ⋮
```

2 Schreiben Sie die Methode *Change*:

```
procedure TSampleCalendar.Change;
begin
   if Assigned(FOnChange) then FOnChange(Self);
end;
```

3 Rufen Sie am Ende der Methoden *SetCalendarDate* und *SetDateElement* die Methode *Change* auf:

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
   FDate := Value;
   UpdateCalendar;
   Change; { Dies ist die einzige neue Anweisung }
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ⋮ { Anweisungen zum Setzen der Elementwerte }
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change; { Neue Anweisung}
   end;
end;
```

Anwendungen, in denen die Kalenderkomponente eingesetzt wird, können nun auf Datumsänderungen reagieren, indem sie dem Ereignis *OnChange* eine entsprechende Behandlungsroutine zuweisen.

Bewegung zu leeren Zellen verhindern

Im gegenwärtigen Zustand des Kalenders kann der Benutzer eine leere Zelle markieren, ohne daß sich das Datum ändert. Es wäre also sinnvoll, die Markierung leerer Zellen zu unterbinden.

Durch Überschreiben der Methode *SelectCell* des Gitters können Sie festlegen, ob eine Zelle markiert werden kann.

Die Funktion *SelectCell* erwartet eine Spalte und eine Zeile als Parameter und gibt einen Booleschen Wert zurück, der angibt, ob die Zelle markiert werden kann.

Sie können die Funktion so überschreiben, daß sie *False* zurückgibt, wenn die Zelle kein gültiges Datum enthält:

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
   if DayNum(ACol, ARow) = -1 then Result := False { -1 weist auf ungültiges Datum hin }
   else Result := inherited SelectCell(ACol, ARow); { Andernfalls geerbte Funktion aufrufen }
end;
```

Wenn der Benutzer nun auf eine leere Zelle klickt oder versucht, mit einer Pfeiltaste in eine leere Zelle zu gelangen, bleibt die aktuelle Zelle markiert.

Kapitel

42

Datensensitive Steuerelemente definieren

Bei der Arbeit mit Datenbankverbindungen ist es häufig von Nutzen, wenn *datensensitive* Steuerelemente verfügbar sind. Das bedeutet, daß die Anwendung eine Verknüpfung zwischen dem Steuerelement und einem Teil der Datenbank einrichten kann. Delphi umfaßt datensensitive Beschriftungen, Eingabefelder, Listenfelder, Kombinationsfelder, Lookup-Steuerelemente und Gitter. Sie können aber auch Ihre eigenen datensensitiven Steuerelemente definieren. Weitere Informationen zur Verwendung von datensensitiven Steuerelementen finden Sie in Kapitel 26, »Datensensitive Steuerelemente«.

Es gibt verschiedene Stufen der Datensensitivität. Die einfachste Stufe ist die Nur-Lesen-Datensensitivität oder das *Bereitstellen* einer Auswahl. Auf dieser Stufe wird der aktuelle Status einer Datenbank angezeigt. Komplizierter ist die zur *Bearbeitung* fähige Datensensitivität. Hier kann der Benutzer die Werte in der Datenbank durch Ändern des Steuerelements bearbeiten. Beachten Sie, daß der Grad der Einflußnahme auf die Datenbank unterschiedlich sein kann. Die Bandbreite reicht vom einfachsten Fall, der Verknüpfung mit einem einzelnen Feld, bis hin zu komplexeren Fällen wie bei Steuerelementen mit mehreren Datensätzen.

In diesem Kapitel wird zuerst der einfachste Fall dargestellt: Es wird ein schreibgeschütztes Steuerelement erzeugt, das eine Verknüpfung zu einem einzigen Feld einer Datenmenge besitzt. Bei diesem Steuerelemente handelt es sich um das in Kapitel 41, »Gitter anpassen«, erstellte Objekt *TSampleCalendar*. Ebenso eignet sich das Steuerelement *TCalendar*, das auf der Registerkarte *Beispiele* der Komponentenpalette zu finden ist.

Anschließend wird in diesem Kapitel erklärt, wie das neue Steuerelement für die Bearbeitung von Daten eingesetzt werden kann.

Ein Steuerelement zur Datensuche erzeugen

Zur Erstellung eines datensensitiven Kalenders müssen Sie die nachstehenden Arbeitsschritte ausführen. Der Kalender kann entweder ein datensensitives Nur-Lesen-Steuerelement sein oder ein Steuerelement, in dem der Benutzer die zugrundeliegenden Daten in der Datenmenge ändern kann.

- Die Komponente erstellen und registrieren.
- Die Datenverknüpfung hinzufügen.
- Auf Datenänderungen reagieren.

Die Komponente erstellen und registrieren

Die Erstellung einer Komponente beginnt auf dieselbe Art und Weise: Sie legen eine Unit an, leiten eine Komponentenklasse ab, registrieren, compilieren und installieren sie in der Komponentenpalette. Dieser Vorgang wird in »Eine neue Komponente erzeugen« auf Seite 31-9 beschrieben.

Für das vorliegende Beispiel führen Sie die allgemeine Prozedur zum Erstellen einer Komponente aus, wobei folgende Schwerpunkte gesetzt werden:

- Benennen Sie die Unit der Komponente *DBCal*.
- Leiten Sie eine neue Komponente namens *TDBCalendar* ab, die eine untergeordnete Klasse von *TSampleCalendar* ist. In Kapitel 41, »Gitter anpassen«, wird die Erstellung der Komponente *TSampleCalendar* erläutert.
- Registrieren Sie *TDBCalendar* auf der Registerkarte *Beispiele* in der Komponentenpalette.

Die resultierende Unit sollte folgendermaßen aussehen:

```
unit DBCal;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
   Forms, Grids, Calendar;
type
   TDBCalendar = class(TSampleCalendar)
   end;
procedure Register;
implementation
procedure Register;
begin
   RegisterComponents('Beispiele', [TDBCalendar]);
end;
end.
```

Nun können Sie den neuen Kalender mit Datensuchfunktionen ausstatten.

Den Kalender als Nur-Lesen-Steuerelement definieren

Da dieser Kalender in bezug auf die Daten das Attribut Nur-Lesen hat, ist es sinnvoll, auch dem Steuerelement selbst dieses Attribut zuzuweisen. Dies verhindert, daß Benutzer Änderungen im Datenelement ausführen und deren Übernahme in die Datenbank erwarten.

Zur Definition des Kalenders als Nur-Lesen-Steuerelement gehören folgende Arbeitsschritte:

- Die Eigenschaft *ReadOnly* hinzufügen.
- Erforderliche Aktualisierungen zulassen.

Beachten Sie, daß die Komponente *TCalendar* bereits die Eigenschaft *ReadOnly* besitzt, wenn Sie auf der Registerkarte *Beispiele* und nicht von *TSampleCalendar* aus mit der Arbeit beginnen. In diesem Fall lassen Sie diese Arbeitsschritte weg.

Die Eigenschaft ReadOnly hinzufügen

Durch das Hinzufügen der Eigenschaft *ReadOnly* kann das Steuerelement zur Entwurfszeit als Nur-Lesen-Element definiert werden. Wenn die Eigenschaft auf *True* gesetzt ist, können Sie alle Zellen im Steuerelement als nicht auswählbar definieren.

1 Fügen Sie die Eigenschaftsdeklaration und ein als **private** deklariertes Feld für das Speichern des Wertes hinzu:

```
type
   TDBCalendar = class(TSampleCalendar)
   private
    FReadOnly: Boolean; { Feld für interne Speicherung }
   public
    constructor Create(AOwner: TComponent); override; { Überschreiben, um den Standard
                                                        zu setzen  }
   published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
   end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
   inherited Create(AOwner); { Immer den geerbten Konstruktor aufrufen! }
   FReadOnly := True; { Standardwert setzen }
end;
```

2 Überschreiben Sie die Methode *SelectCell*, um eine Auswahl zu unterbinden, wenn das Steuerelement das Attribut Nur-Lesen aufweist. Die Verwendung von *SelectCell* wird unter »Bewegung zu leeren Zellen verhindern« auf Seite 41-13 beschrieben.

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
   if FReadOnly then Result := False { Keine Auswahl möglich, wenn schreibgeschützt }
   else Result := inherited SelectCell(ACol, ARow); { Andernfalls geerbte Methode verwenden }
end;
```

Vergessen Sie nicht, die Deklaration von *SelectCell* in die Typdeklaration von *TDBCalendar* aufzunehmen und die Direktive **override** anzufügen.

Wenn Sie nun den Kalender in ein Formular integrieren, werden Sie bemerken, daß die Komponente Mausklicks und Tastenanschläge ignoriert. Außerdem schlägt die Aktualisierung der Auswahlposition fehl, wenn das Datum geändert wird.

Erforderliche Aktualisierungen zulassen

Der Nur-Lesen-Kalender verwendet die Methode *SelectCell* für alle Arten von Änderungen, einschließlich des Setzens der Eigenschaften *Row* und *Col*. Mit der Methode *UpdateCalendar* werden *Row* und *Col* jedesmal dann gesetzt, wenn sich das Datum ändert. Da *SelectCell* aber Änderungen verhindert, bleibt die Auswahl selbst bei einer Datumsänderung in Kraft.

Zur Umgehung dieses absoluten Änderungsverbots können Sie dem Kalender ein internes Boolesches Flag hinzufügen und somit erlauben, daß Änderungen durchgeführt werden, wenn das Flag auf *True* gesetzt ist

```
type
   TDBCalendar = class(TSampleCalendar)
   private
    FUpdating: Boolean; { private-Flag für interne Verwendung }
   protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
   public
    procedure UpdateCalendar; override; { Direktive override nicht vergessen }
   end;
⋮
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
   if (not FUpdating) and FReadOnly then Result := False { auswählen beim Aktualisieren
                                                           möglich }
   else Result := inherited SelectCell(ACol, ARow); { Andernfalls geerbte Methode verwenden }
end;
procedure TDBCalendar.UpdateCalendar;
begin
   FUpdating := True; { Flag setzen, um Aktualisierungen zu ermöglichen }
   try
    inherited UpdateCalendar; { Wie gewohnt aktualisieren }
   finally
    FUpdating := False; { Flag immer löschen }
   end;
end;
```

Der Kalender erlaubt immer noch keine Änderungen durch den Benutzer, spiegelt aber jetzt die ausgeführten Datumsänderungen durch eine entsprechende Änderung der Datumseigenschaften wider. Nun verfügen Sie über einen Nur-Lesen-Kalender, dem Sie die Katalog-Funktion hinzufügen können.

Die Datenverknüpfung hinzufügen

Die Verbindung zwischen einem Steuerelement und einer Datenbank wird von einer Klasse ausgeführt, die *Datenverknüpfung* genannt wird. Diese Klasse, die ein Steuerelement mit einem einzelnen Feld in einer Datenbank verbindet, heißt *TFieldDataLink*. Darüber hinaus gibt es auch Datenverknüpfungen für gesamte Tabellen.

Ein datensensitives Steuerelement besitzt eigene Datenverknüpfungsklassen. Das bedeutet, daß das Steuerelement die Verantwortung für den Aufbau und die Freigabe der Datenverknüpfung übernimmt. Details zur Verwaltung von untergeordneter Klassen finden Sie in Kapitel 40, »Grafische Komponenten erzeugen«.

Zur Einrichtung einer Datenverknüpfung als untergeordnete Klasse führen Sie folgende Arbeitsschritte aus:

1 Deklarieren Sie das Klassenfeld.

2 Deklarieren Sie die Zugriffseigenschaften.

3 Initialisieren Sie die Datenverknüpfung.

Das Klassenfeld deklarieren

Wie unter »Die Klassenfelder deklarieren« auf Seite 40-6 beschrieben, benötigt eine Komponente für jede ihrer Klassen ein Feld. Im vorliegenden Fall benötigt der Kalender für die Datenverknüpfung ein Feld vom Typ *TFieldDataLink*.

Deklarieren Sie ein Feld für die Datenverknüpfung im Kalender:

```
type
   TDBCalendar = class(TSampleCalendar)
   private
    FDataLink: TFieldDataLink;
   ⋮
   end;
```

Bevor Sie die Anwendung compilieren können, müssen Sie *DB* und *DBCtrls* in die **uses**-Klausel der Unit einbinden.

Die Zugriffseigenschaften deklarieren

Alle datensensitiven Steuerelemente besitzen eine Eigenschaft namens *DataSource*, mit der angegeben wird, welche Datenquellenklasse in der Anwendung die Daten für das Steuerelement bereitstellt. Außerdem benötigt ein Steuerelement, das auf ein einzelnes Feld zugreift, die Eigenschaft *DataField*. Mit dieser Eigenschaft wird das Feld in der Datenquelle angegeben.

Im Gegensatz zu den Zugriffseigenschaften für die untergeordnete Klasse in dem Beispiel in Kapitel 40, »Grafische Komponenten erzeugen«, ermöglichen die Zugriffseigenschaften hier keinen Zugriff auf die untergeordneten Klassen selbst, sondern auf die entsprechenden Eigenschaften in diesen Klassen. Sie erstellen somit Eigenschaften, die das Steuerelement und seine Datenverknüpfung dazu veranlassen, dieselbe Datenquelle und dasselbe Feld gemeinsam zu nutzen.

Deklarieren Sie die Eigenschaften *DataSource* und *DataField* und deren Implementierungsmethoden. Schreiben Sie dann die Methoden als »Passthrough-Methoden« für die entsprechenden Eigenschaften der Datenverknüpfungsklasse:

Beispiel für die Deklaration von Zugriffseigenschaften

```
type
   TDBCalendar = class(TSampleCalendar)
   private { Implementierungsmethoden sind als private deklariert }
    ...
    function GetDataField: string; { Namen des Datenfeldes zurückliefern }
    function GetDataSource: TDataSource; { Referenz auf die Datenquelle zurückliefern }
    procedure SetDataField(const Value: string); { Namen des Datenfeldes zuweisen }
    procedure SetDataSource(Value: TDataSource); { Neue Datenquelle zuweisen }
   published { Eigenschaften zur Entwurfszeit bereitstellen }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
   end;
⋮
function TDBCalendar.GetDataField: string;
begin
   Result := FDataLink.FieldName;
end;
function TDBCalendar.GetDataSource: TDataSource;
begin
   Result := FDataLink.DataSource;
end;
procedure TDBCalendar.SetDataField(const Value: string);
begin
   FDataLink.FieldName := Value;
end;
procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
   FDataLink.DataSource := Value;
end;
```

Jetzt haben Sie die Verknüpfungen zwischen dem Kalender und seiner Datenverknüpfung hergestellt. Nun ist ein weiterer wichtiger Schritt auszuführen. Sie müssen die Datenverknüpfungsklasse erstellen, wenn der Kalender aufgebaut wird, und die Datenverknüpfung vor dem Kalender freigeben.

Die Datenverknüpfung initialisieren

Ein datensensitives Steuerelement muß während seiner gesamten Lebensdauer auf seine Datenverbindung zugreifen können. Aus diesem Grund muß das Datenverknüpfungsobjekt als Teil seines untergeordneten Konstruktors aufgebaut werden. Außerdem muß das Datenverknüpfungsobjekt freigegeben werden, bevor das Steuerelement selbst freigegeben wird.

Überschreiben Sie die Methoden *Create* und *Destroy* des Kalenders, um das Datenverknüpfungsobjekt zu erzeugen bzw. freizugeben:

```
type
```

```
    TDBCalendar = class(TSampleCalendar)
    public { Konstruktoren und Destruktoren immer als public deklarieren }
     constructor Create(AOwner: TComponent); override;
     destructor Destroy; override;
     ⋮
    end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
    FDataLink := TFieldDataLink.Create; { Das Datenverknüpfungsobjekt erzeugen }
    inherited Create(AOwner); { Immer zuerst den geerbten Konstruktor aufrufen }
    FReadOnly := True; { Ist bereits vorhanden }
end;
destructor TDBCalendar.Destroy;
begin
    FDataLink.Free; { Immer zuerst die untergeordneten Objekte freigeben }
    inherited Destroy; { und danach den geerbten Konstruktor aufrufen }
end;
```

Jetzt verfügen Sie über eine vollständige Datenverknüpfung. Nun muß dem Steuerelement noch mitgeteilt werden, welche Daten aus dem verknüpften Feld gelesen werden sollen. Im nächsten Abschnitt wird dieser Punkt erläutert.

Auf Datenänderungen antworten

Sobald ein Steuerelement über eine Datenverknüpfung sowie über Eigenschaften zur Angabe der Datenquelle und des Datenfeldes verfügt, muß es auf Datenänderungen in diesem Feld antworten. Derartige Änderungen treten ein, wenn eine Bewegung zu einem anderen Datensatz ausgeführt oder eine Änderung in diesem Feld vorgenommen wurde.

Alle Datenverknüpfungsklassen besitzen Ereignisse mit dem Namen *OnDataChange*. Wenn die Datenquelle eine Änderung ihrer Daten anzeigt, ruft das Datenverknüpfungsobjekt eine mit seinem *OnDataChange*-Ereignis verbundene Behandlungsroutine auf.

Um ein Steuerelement als Antwort auf Datenänderungen aktualisieren zu können, verbinden Sie eine Behandlungsroutine mit dem Ereignis *OnDataChange* der Datenverknüpfung.

In diesem Fall fügen Sie eine Methode in den Kalender ein und kennzeichnen sie dann als die Behandlungsroutine für das Ereignis *OnDataChange* der Datenverknüpfung.

Deklarieren und implementieren Sie die Methode *DataChange*, und weisen Sie sie im Konstruktor der Datenverknüpfung dem Ereignis *OnDataChange* zu. Heben Sie im Destruktor die Verknüpfung der *OnDataChange*-Behandlungsroutine auf, bevor Sie das Objekt freigeben.

```
type
    TDBCalendar = class(TSampleCalendar)
    private { Dieses interne Detail als private deklarieren }
```

```
    procedure DataChange(Sender: TObject); { Es muß der dem Ereignis entsprechende
                                             Parameter vorhanden sein }
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { Immer zuerst den geerbten Konstruktor aufrufen }
  FReadOnly := True; { Ist bereits vorhanden }
  FDataLink := TFieldDataLink.Create; { Datenverknüpfungsobjekt erzeugen }
  FDataLink.OnDataChange := DataChange; { Behandlungsroutine mit dem Ereignis verbinden }
end;
destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil; { vor der Freigabe des Objekts die Verbindung zur Routine
                                   lösen }
  FDataLink.Free; { Immer zuerst untergeordnete Objekte freigeben }
  inherited Destroy; { und danach den geerbten Konstruktor aufrufen }
end;
procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then { Wenn keine Feldzuweisung vorhanden,  }
   CalendarDate := 0 { auf ungültiges Datum setzen,  }
  else CalendarDate := FDataLink.Field.AsDateTime; { ansonsten Kalender auf das Datum
                                                     setzen }
end;
```

Ihr Steuerelement kann nun zur Datensuche verwendet werden.

Ein Bearbeitungselement erstellen

Wenn Sie ein Bearbeitungselement anlegen, erstellen und registrieren Sie die Komponente und fügen die Datenverknüpfung auf dieselbe Weise hinzu wie beim Steuerelement für die Datensuche. Auch die Antwort auf Änderungen in dem zugrundeliegenden Feld erfolgt ähnlich wie bei den Steuerelementen für die Datensuche. Sie müssen für Bearbeitungselemente allerdings noch einige weitere Anforderungen erfüllen.

Wahrscheinlich wollen Sie, daß Ihr Steuerelement sowohl auf Tastatur- als auch auf Mausereignisse antwortet. Wenn der Benutzer den Inhalt des Elements ändert, muß das Steuerelement antworten. Verläßt der Benutzer das Steuerelement, sollen die vorgenommenen Änderungen in die Datenmenge übernommen werden.

Bei dem hier beschriebenen Bearbeitungselement handelt es sich um denselben Kalender, der im ersten Teil des Kapitels beschrieben wurde. Er wird so geändert, daß die Daten in seinem verknüpften Feld bearbeitet und angezeigt werden können.

Sie müssen folgende Schritte ausführen, um das vorhandene Steuerelement mit Datenbearbeitungsfunktionen auszustatten:

- Ändern Sie den Standardwert von *FReadOnly*.
- Bearbeiten Sie Maus- und Tastendruckbotschaften.
- Aktualisieren Sie die Datenverknüpfungsklasse des Feldes.

- Ändern Sie die Methode *Change*.
- Aktualisieren Sie die Datenmenge.

Den Standardwert von FReadOnly ändern

Da es sich hier um ein Bearbeitungselement handelt, muß die Eigenschaft *ReadOnly* per Voreinstellung auf *False* gesetzt werden. Zu diesem Zweck ändern Sie den Wert von *FReadOnly* im Konstruktor:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
   ⋮
   FReadOnly := False;  { Standardwert setzen }
   ⋮
end;
```

Maustasten- und Tastendruckbotschaften behandeln

Wenn der Benutzer die Interaktion mit dem Steuerelement beginnt, empfängt das Steuerelement von Windows entweder Botschaften darüber, daß die Maustaste (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN* oder *WM_RBUTTONDOWN*) oder eine Taste auf der Tastatur (*WM_KEYDOWN*) gedrückt wurde. Damit ein Steuerelement auf diese Botschaften antworten kann, müssen Sie Behandlungsroutinen schreiben, die auf diese Botschaften reagieren.

- Auf Maustastenbotschaften antworten
- Auf Tastendruckbotschaften antworten

Auf Maustastenbotschaften antworten

MouseDown ist eine als **protected** deklarierte Methode für das Ereignis *OnMouseDown* eines Steuerelements. Das Steuerelement selbst ruft *MouseDown* als Antwort auf die Windows-Maustastenbotschaft auf. Beim Überschreiben der geerbten Methode *MouseDown* können Sie Quelltext einfügen, der zusätzlich zum Aufruf des Ereignisses *OnMouseDown* weitere Anworten bereitstellt.

Fügen Sie, zum Überschreiben von *MouseDown*, der Klasse *TDBCalendar* die Methode *MouseDown* hinzu:

```
type
   TDBCalendar = class(TSampleCalendar);
   ⋮
   protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
   ⋮
   end;
procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
   MyMouseDown: TMouseEvent;
```

```
begin
   if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
   else
   begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
   end;
end;
```

Wenn *MouseDown* auf Maustastenbotschaften antwortet, wird die geerbte Methode *MouseDown* nur unter der Voraussetzung aufgerufen, daß das Feld bearbeitet werden kann. Dazu muß die Eigenschaft *ReadOnly* des Steuerelements den Wert *False* haben, und das Datenverknüpfungsobjekt muß sich im Bearbeitungsmodus befinden. Kann das Feld dagegen nicht bearbeitet werden, wird der Quelltext ausgeführt, den der Programmierer in die Ereignisbehandlungsroutine für *OnMouseDown* eingefügt hat.

Auf Tastendruckbotschaften antworten

KeyDown ist eine als **protected** deklarierte Methode für das Ereignis *OnKeyDown* des Steuerelements. Das Steuerelement selbst ruft die Methode *KeyDown* als Antwort auf eine Windows-Tastendruckbotschaft auf. Beim Überschreiben der geerbten Methode *KeyDown* können Sie Quelltext einfügen, der zusätzlich zum Aufruf des Ereignisses *OnKeyDown* weitere Anworten bereitstellt.

Führen Sie zum Überschreiben von *KeyDown* folgende Schritte aus:

1 Fügen Sie die Methode *KeyDown* in die Klasse *TDBCalendar* ein:

```
type
   TDBCalendar = class(TSampleCalendar);
   ⋮
   protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
   ⋮
   end;
```

2 Implementieren Sie die Methode *KeyDown*:

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
   MyKeyDown: TKeyEvent;
begin
   if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
   else
   begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
   end;
end;
```

Wenn *KeyDown* auf Tastendruckbotschaften antwortet, wird die geerbte Methode *KeyDown* nur unter der Voraussetzung aufgerufen, daß die Eigenschaft *ReadOnly* des Steuerelements den Wert *False* hat, die gedrückte Taste eine Cursortaste ist und das Datenverknüpfungsobjekt sich im Bearbeitungsmodus befindet (d. h. das Feld kann bearbeitet werden). Kann das Feld nicht bearbeitet werden oder wurde eine andere Taste gedrückt, wird der Quelltext ausgeführt, den der Programmierer in die Ereignisbehandlungsroutine für *OnKeyDown* eingefügt hat.

Die Datenverknüpfungsklasse des Feldes aktualisieren

Es gibt zwei Arten von Datenänderungen:

- Eine Änderung im Feldwert, die im datensensitiven Steuerelement sichtbar werden muß.
- Eine Änderung im datensensitiven Steuerelement, die im Feldwert der Datenbank berücksichtigt werden muß.

Die Komponente *TDBCalendar* verfügt bereits über die Methode *DataChange*, die eine Änderung des Feldwertes in der Datenmenge bearbeitet, indem sie diesen Wert der Eigenschaft *CalendarDate* zuweist. Die Methode *DataChange* ist die Behandlungsroutine für das Ereignis *OnDataChange*. Folglich kann die Kalenderkomponente die erste Art der Datenänderung bearbeiten.

Parallel dazu verfügt die Datenverknüpfungsklasse des Feldes über ein Ereignis *OnUpdateData*. Dieses Ereignis tritt ein, wenn der Benutzer den Inhalt des datensensitiven Steuerelements ändert. Der Kalender besitzt die Methode *UpdateData*, die zur Behandlungsroutine für das Ereignis *OnUpdateData* wird. *UpdateData* weist dem Feld der Datenverknüpfung den geänderten Wert des datensensitiven Steuerelements zu.

1 Damit die Änderung des Wertes im Kalender im Feldwert angezeigt wird, fügen Sie eine *UpdateData*-Methode in den **private**-Abschnitt der Kalenderkomponente ein:

```
type
   TDBCalendar = class(TSampleCalendar);
   private
    procedure UpdateData(Sender: TObject);
   ⋮
   end;
```

2 Implementieren Sie die Methode *UpdateData*:

```
procedure UpdateData(Sender: TObject);
begin
   FDataLink.Field.AsDateTime := CalendarDate; { Feldverknüpfung zum Kalenderdatum }
end;
```

3 Verknüpfen Sie im Konstruktor für *TDBCalendar* die Methode *UpdateData* mit dem Ereignis *OnUpdateData*:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);
   FReadOnly := True;
```

```
    FDataLink := TFieldDataLink.Create;
    FDataLink.OnDataChange := DataChange;
    FDataLink.OnUpdateData := UpdateData;
end;
```

Die Methode Change ändern

Die Methode *Change* von *TDBCalendar* wird immer dann aufgerufen, wenn ein neues Datum gesetzt wird. *Change* ruft die Ereignisbehandlungsroutine für *OnChange* (falls vorhanden) auf. Der Benutzer der Komponente kann Quelltext in die Ereignisbehandlungsroutine für *OnChange* einfügen, der auf Datumsänderungen antwortet.

Wenn sich das Kalenderdatum ändert, muß die zugrundeliegende Datenmenge von der eingetretenen Änderung benachrichtigt werden. Dazu können Sie die Methode *Change* überschreiben und eine weitere Quelltextzeile hinzufügen. Führen Sie nun folgende Arbeitsschritte aus:

1 Fügen Sie der Komponente *TDBCalendar* eine neue *Change*-Methode hinzu:

```
type
   TDBCalendar = class(TSampleCalendar);
   private
    procedure Change; override;
   ⋮
   end;
```

2 Schreiben Sie die Methode *Change*, und rufen Sie die Methode *Modified* auf, um die Datenmenge über die Datenänderung zu informieren. Danach wird die geerbte Methode *Change* aufgerufen:

```
TDBCalendar.Change;
begin
   FDataLink.Modified; { Methode Modified aufrufen }
   inherited Change; { Geerbte Methode Change aufrufen }
end;
```

Die Datenmenge aktualisieren

Bis jetzt wurden durch eine Änderung im datensensitiven Steuerelement die Werte in der Datenverknüpfungsklasse des Feldes geändert. Im letzten Arbeitsschritt muß die Datenmenge mit dem neuen Wert aktualisiert werden. Die Aktualisierung muß erfolgen, nachdem der Benutzer den Wert im datensensitiven Steuerelement geändert hat und das Steuerelement verläßt, und zwar entweder durch Klicken außerhalb des Steuerelements oder durch Drücken der Taste *TAB*.

In der VCL sind Botschafts-IDs für Operationen mit Steuerelementen vordefiniert. Beispielsweise wird die Botschaft *CM_EXIT* an das Steuerelement gesendet, wenn der Benutzer das Steuerelement verläßt. In diesem Fall antwortet die Methode *CMExit* (die Botschaftsbehandlungsroutine für *CM_EXIT*), indem sie den Datensatz in der Datenmenge mit den geänderten Werten in der Datenverknüpfungsklasse des Feldes aktualisiert. Informationen über die Behandlung von Botschaften finden Sie in Kapitel 37, »Botschaftsbehandlung«.

Zur Aktualisierung der Datenmenge in einer Botschaftsbehandlungsroutine führen Sie folgende Arbeitsschritte aus:

1 Fügen Sie der Komponente *TDBCalendar* die Botschaftsbehandlungsroutine hinzu:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
   procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
  ⋮
  end;
```

2 Implementieren Sie die Methode *CMExit*, damit sie in etwa folgendermaßen aussieht:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
   FDataLink.UpdateRecord; { Datenverknüpfung zur Aktualisierung der Datenbank auffordern }
  except
   on Exception do SetFocus; { Bei Fehlschlag den Fokus nicht weitergeben }
  end;
  inherited;
end;
```

Kapitel

43

Dialogfelder als Komponenten

Sie werden im Verlauf Ihrer Arbeit feststellen, daß es von Nutzen ist, häufig verwendete Dialogfelder als Komponenten in die Komponentenpalette aufzunehmen. Diese Dialogfeldkomponenten funktionieren wie die Komponenten, welche die allgemeinen Standard-Dialogfelder von Windows repräsentieren. Das Ziel liegt in der Erstellung einer einfachen Komponente, die der Benutzer in ein Projekt einfügen und für die er zur Entwurfszeit Eigenschaften festlegen kann.

Die Definition eines Dialogfeldes als Komponente umfaßt folgende Arbeitsschritte:

1. Die Komponentenschnittstelle definieren.
2. Die Komponente erstellen und registrieren.
3. Die Komponentenschnittstelle erstellen.
4. Die Komponente testen.

Die »kapselnde« Komponente, die mit dem Dialogfeld verbunden ist, erstellt und führt das Dialogfeld zur Laufzeit aus und übergibt die vom Benutzer eingegebenen Daten. Die Dialogfeldkomponente kann deshalb wiederverwendet und angepaßt werden.

Im vorliegenden Kapitel erfahren Sie, wie man eine kapselnde Komponente erstellt, in der das generische Formular *Info* aus der Objektablage von Delphi integriert ist.

Hinweis Kopieren Sie die Dateien ABOUT.PAS und ABOUT.DFM in Ihr Arbeitsverzeichnis.

Für den Entwurf eines Dialogfeldes, das als Komponente definiert werden soll, bedarf es nicht vieler Überlegungen. Nahezu jedes Formular kann in diesem Kontext als Dialogfeld eingesetzt werden.

Die Komponentenschnittstelle definieren

Bevor Sie die Komponente für Ihr Dialogfeld erstellen können, müssen Sie festlegen, wie die Entwickler das Dialogfeld nutzen sollen. Sie definieren eine Schnittstelle zwischen dem Dialogfeld und den Anwendungen, in denen es eingesetzt wird.

Sehen Sie sich beispielsweise die Eigenschaften von Standard-Dialogfeldern an. Sie ermöglichen es dem Entwickler, den Anfangsstatus des Dialogfeldes (z. B. den Titel und die Einstellungen für die enthaltenen Steuerelemente) festzulegen und nach dem Schließen des Dialogfeldes deren Status zu überprüfen. Es findet keine direkte Interaktion mit den einzelnen Steuerelementen im Dialogfeld statt, sondern nur mit den Eigenschaften in der kapselnden Komponente.

Die Schnittstelle muß deshalb genügend Informationen enthalten, damit das Dialogfeldformular den Angaben des Entwicklers gemäß angezeigt werden kann. Außerdem müssen alle Informationen zurückgeliefert werden, die für die Anwendung erforderlich sind. Sie können sich die Eigenschaften in der kapselnden Komponente als persistente Daten für ein »kurzlebiges« Dialogfeld vorstellen.

Im Fall des Dialogfeldes *Info* müssen keine Informationen zurückgegeben werden. Aus diesem Grund müssen die Eigenschaften der kapselnden Komponente nur diejenigen Informationen enthalten, die für die korrekte Anzeige des Dialogfeldes erforderlich sind. In diesem Dialogfeld sind vier separate Felder vorhanden, die sich auf die Anwendung beziehen können. Deshalb müssen Sie vier Eigenschaften vom Typ String für diese Felder bereitstellen.

Die Komponente erstellen und registrieren

Die Erstellung aller Komponenten beginnt auf dieselbe Weise: Sie legen eine Unit an, leiten eine Komponentenklasse ab, registrieren, compilieren und installieren die Komponente in der Komponentenpalette. Dieser Vorgang wird in Kapitel 31, »Die Komponentenentwicklung im Überblick«, beschrieben.

Führen Sie für unser Beispiel unter Berücksichtigung der folgenden Besonderheiten den allgemeinen Erstellungsvorgang für Komponenten aus:

- Benennen Sie die Unit der Komponente *AboutDlg*.
- Leiten Sie einen neuen Komponententyp namens *TAboutBoxDlg* ab, der ein Nachkomme von *TComponent* ist.
- Registrieren Sie *TAboutBoxDlg* in der Registerkarte *Beispiele* der Komponentenpalette.

Die sich daraus ergebende Unit sollte folgendermaßen aussehen:

```
unit AboutDlg;
interface
uses
   SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
   TAboutBoxDlg = class(TComponent)
```

```
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Beispiele', [TAboutBoxDlg]);
end;
end.
```

Die neue Komponente besitzt jetzt nur die Fähigkeiten, über die auch *TComponent* verfügt. Sie stellt die einfachste, nicht-visuelle Komponente dar. Im nächsten Abschnitt werden Sie die Schnittstelle zwischen der Komponente und dem Dialogfeld erstellen.

Die Komponentenschnittstelle erstellen

Mit folgenden Arbeitsschritten wird eine Komponentenschnittstelle erstellt:

1. Die Unit-Dateien des Formulars einfügen.
2. Die Schnittstelleneigenschaften hinzufügen.
3. Die Methode *Execute* hinzufügen.

Die Unit des Formulars einfügen

Zur Initialisierung der kapselnden Komponente und zur Anzeige des Dialogfeldes müssen Sie die Unit des Formulars zur **uses**-Klausel der Unit für die kapselnde Komponente hinzufügen.

Fügen Sie den Eintrag *About* in die **uses**-Klausel der Unit *AboutDlg* ein.

Die **uses**-Klausel sieht nun folgendermaßen aus:

```
uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms
    About;
```

In der Unit des Formulars wird immer eine Instanz der Formularklasse deklariert. Im Falle des Dialogfeldes *Info* handelt es sich um die Klasse *TAboutBox*, und die Unit *About* enthält folgende Deklaration:

```
var
    AboutBox: TAboutBox;
```

Durch das Hinzufügen von *About* zur **uses**-Klausel kann die kapselnde Komponente auf *AboutBox* zugreifen.

Die Schnittstelleneigenschaften hinzufügen

Bevor Sie jetzt weiterarbeiten, müssen Sie folgende Frage beantworten: Welche Eigenschaften benötigt die kapselnde Komponente, damit die Entwickler das Dialog-

feld als Komponente in ihren Anwendungen einsetzen können? Danach fügen Sie die Deklarationen für die betreffenden Eigenschaften in die Klassendeklaration der Komponente ein.

Die Eigenschaften in kapselnden Komponenten sind etwas einfacher strukturiert als diejenigen, die Sie bei der Erstellung einer regulären Komponente definieren würden. Sie erstellen hier nur einige persistente Daten, welche die kapselnde Komponente an das Dialogfeld übergibt bzw. daraus abruft. Indem Sie diese Daten als Eigenschaften definieren, ermöglichen Sie den Entwicklern, Daten zur Entwurfszeit festzulegen, die zur Laufzeit von der kapselnden Komponente an das Dialogfeld übergeben werden können.

Zur Deklaration einer Schnittstelleneigenschaft müssen in die Klassendeklaration der Komponente zwei Dinge eingefügt werden:

- Ein als **private** deklariertes Klassenfeld, das von der kapselnden Komponente als Variable zum Speichern des Eigenschaftswertes verwendet wird.
- Die Deklaration der als **published** deklarierten Eigenschaft selbst, die den Namen der Eigenschaft angibt und festlegt, welches Feld für die Speicherung verwendet werden soll.

Derartige Schnittstelleneigenschaften benötigen keine Zugriffsmethoden. Sie greifen direkt auf ihre gespeicherten Daten zu. Per Konvention trägt das Klassenfeld, in dem der Eigenschaftswert gespeichert ist, denselben Namen wie die Eigenschaft, allerdings mit dem Buchstaben F am Namensbeginn. Das Feld und die Eigenschaft müssen denselben Typ aufweisen.

Um beispielsweise eine Schnittstelleneigenschaft namens *Year* vom Typ Integer anzugeben, geben Sie folgende Deklaration an:

```
type
   TMyWrapper = class(TComponent)
   private
    FYear: Integer; { Feld zum Speichern der Daten aus der Eigenschaft Year }
   published
    property Year: Integer read FYear write FYear; { Eigenschaft wird mit Speicher
                                                      verglichen }
   end;
```

Für das Dialogfeld *Info* benötigen Sie vier Eigenschaften vom Typ String — eine für den Produktnamen, eine für die Versionsinformation, eine für die Copyright-Information und eine für Kommentare.

```
type
   TAboutBoxDlg = class(TComponent)
   private
    FProductName, FVersion, FCopyright, FComments: string; { Felder deklarieren }
   published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
   end;
```

Wenn Sie die Komponente in der Komponentenpalette installieren und anschließend in ein Formular einfügen, können Sie die Eigenschaftswerte festlegen. Diese Werte bleiben dann automatisch mit dem Formular verbunden. Die kapselnde Komponente kann die Werte zur Ausführung des enthaltenen Dialogfeldes verwenden.

Die Methode Execute hinzufügen

Im letzten Abschnitt der Komponentenschnittstelle muß eine Möglichkeit zum Öffnen des Dialogfeldes und zur Rückgabe eines Ergebnisses nach dem Schließen implementiert werden. Wie bei anderen Standard-Dialogfeldern verwenden Sie auch hier eine Boolesche Funktion namens *Execute*, die den Wert *True* zurückliefert, wenn der Benutzer auf *OK* klickt. Schließt der Benutzer das Dialogfeld, gibt die Funktion *False* zurück.

Die Deklaration für die Methode *Execute* sieht folgendermaßen aus:

```
type
   TMyWrapper = class(TComponent)
   public
    function Execute: Boolean;
   end;
```

Die Minimalimplementation für *Execute* muß das Dialogfeldformular erzeugen, es als modales Dialogfeld anzeigen und entweder *True* oder *False* zurückliefern (abhängig vom Rückgabewert von *ShowModal*).

Im folgenden sehen Sie die Minimalimplementation der Methode *Execute* für ein Dialogfeldformular vom Typ *TMyDialogBox*:

```
function TMyWrapper.Execute: Boolean;
begin
   DialogBox := TMyDialogBox.Create(Application); { Formular erzeugen }
   try
    Result := (DialogBox.ShowModal = IDOK); { Ausführen; Ergebnis basierend auf der Art
                                              des Schließens setzen }
   finally
    DialogBox.Free; { Formular freigeben }
   end;
end;
```

Beachten Sie die Verwendung des **try..finally**-Blocks. Dieser Block stellt sicher, daß die Anwendung das Dialogfeldobjekt auch dann freigibt, wenn eine Exception auftritt. Generell gilt: Sie sollten bei einer derartigen Konstruktion eines Objekts immer einen **try..finally**-Block einsetzen, um den Quelltextblock zu schützen und sicherzustellen, daß die Anwendung alle zugewiesenen Ressourcen wieder freigibt.

In der Praxis wird sich weitaus mehr Quelltext im **try..finally**-Block befinden. Insbesondere setzt die kapselnde Komponente vor dem Aufruf von *ShowModal* einige Eigenschaften für das Dialogfeld, ausgehend von den Schnittstelleneigenschaften der kapselnden Komponente. Nach der Rückkehr von *ShowModal* setzt die kapselnde Komponente meist einige ihrer Schnittstelleneigenschaften basierend auf dem Ergebnis der Dialogfeldausführung.

Für das Dialogfeld *Info* müssen Sie die vier Schnittstelleneigenschaften der kapselnden Komponente verwenden, um die Beschriftungen im Dialogfeldformular festzulegen. Da das Dialogfeld *Info* keine Informationen an die Anwendung zurückliefert, sind nach dem Aufruf von *ShowModal* keine weiteren Aktionen erforderlich. Der folgende Quelltextausschnitt zeigt, wie die Methode *Execute* für die kapselnde Komponente des Dialogfeldes *Info* geschrieben werden muß.

Fügen Sie die Deklaration für die Methode *Execute* in den als **public** deklarierten Abschnitt der Klasse *TAboutDlg* ein:

```
type
   TAboutDlg = class(TComponent)
public
   function Execute: Boolean;
end;
function TAboutBoxDlg.Execute: Boolean;
begin
   AboutBox := TAboutBox.Create(Application); { Dialogfeld Info erzeugen }
   try
    if ProductName = '' then { Wenn Produktname leer, }
      ProductName := Application.Title; { Titel der Anwendung verwenden. }
    AboutBox.ProductName.Caption := ProductName; { Produktnamen kopieren }
    AboutBox.Version.Caption := Version; { Versionsinformation kopieren }
    AboutBox.Copyright.Caption := Copyright; { Copyright-Informationen kopieren }
    AboutBox.Comments.Caption := Comments; { Kommentare kopieren }
    AboutBox.Caption := 'Info ' + ProductName; { Titel für Info festlegen }
    with AboutBox do begin
      ProgramIcon.Picture.Graphic := Application.Icon; { Das Symbol kopieren }
      Result := (ShowModal = IDOK); { Ausführen und Ergebnis setzen }
    end;
   finally
    AboutBox.Free; { Dialogfeld Info freigeben }
   end;
end;
```

Die Komponente testen

Sobald Sie die Dialogfeldkomponente installiert haben, können Sie diese wie jedes andere Standard-Dialogfeld verwenden, indem Sie die Komponente in ein Formular einfügen und ausführen. Für einen schnellen Test des Dialogfeldes *Info* fügen Sie eine Schaltfläche in ein Formular ein und führen das Dialogfeld aus, sobald der Benutzer auf die Schaltfläche klickt.

Nachdem Sie das Dialogfeld *Info* erstellt, es als Komponente definiert und in die Komponentenpalette eingefügt haben, können Sie es folgendermaßen testen:

1. Erstellen Sie ein neues Projekt.
2. Fügen Sie eine *Info*-Dialogfeldkomponente in das Hauptformular ein.
3. Plazieren Sie eine Schaltfläche im Formular.

4. Doppelklicken Sie auf die Schaltfläche, und erstellen Sie so eine leere Behandlungsroutine für ein Klick-Ereignis.
5. Geben Sie folgende Quelltextzeile in diese Behandlungsroutine ein:

```
AboutBoxDlg1.Execute;
```

6. Führen Sie die Anwendung aus.

Klicken Sie auf die Schaltfläche, wenn das Hauptformular angezeigt wird. Das Dialogfeld *Info* wird mit dem voreingestellten Projekt-Symbol und dem Namen *Project1* eingeblendet. Klicken Sie auf *OK*, um das Dialogfeld zu schließen.

Sie können die Komponente weiter testen, indem Sie ihre verschiedenen Eigenschaften setzen und die Anwendung erneut ausführen.

Teil

V

COM-Anwendungen entwickeln

Die Kapitel in diesem Teil des Entwicklerhandbuchs zeigen Konzepte und Techniken auf, die für die Erzeugung von COM-basierten Anwendungen, einschließlich Automatisierungs-Controllern, Automatisierungs-Servern, ActiveX-Steuerelementen und MTS-Anwendungen, erforderlich sind.

Hinweis Automatisierungs-Controller werden in allen Delphi-Versionen unterstützt. Um jedoch Server zu erstellen, benötigen Sie die Professional- oder die Enterprise-Version von Delphi.

Kapitel

44

COM-Technologien im Überblick

Delphi stellt Experten und Klassen bereit, mit denen sich Anwendungen, die auf dem Component Object Model (COM) von Microsoft beruhen, einfach implementieren lassen. Mit Hilfe dieser Experten können Sie COM-basierte Klassen und Komponenten zur Verwendung in einer einzelnen Anwendung oder voll funktionsfähige COM-Objekte, Automatisierungsserver und -Clients (Controller), ActiveX-Steuerelemente, ASP- oder ActiveForm-Objekte erstellen.

COM ist ein von Microsoft entwickeltes, sprachunabhängiges Software-Komponenten-Modell, das die Interaktion zwischen Software-Komponenten und Anwendungen ermöglichen soll. Microsoft erweitert diese Technologie um ActiveX-Objekte, was eine weitere Verfeinerung des COM-Standards für die Entwicklung von Steuerelementen darstellt (dieser Standard wird primär für die Intranet-Entwicklung eingesetzt). Die Bedeutung von COM besteht im wesentlichen darin, daß dieses Modell durch klar definierte Schnittstellen die Kommunikation zwischen Komponenten, zwischen Anwendungen und zwischen Clients sowie Servern ermöglicht. Schnittstellen bieten Clients die Möglichkeit, von einer COM-Komponente Informationen über die zur Laufzeit unterstützten Funktionen abzurufen. Um eine Komponente mit zusätzlichen Features auszustatten, fügen Sie einfach eine weitere Schnittstelle hinzu.

Eine Anwendung kann auf COM-Komponenten und deren Schnittstellen, die sich auf demselben Computer wie die Anwendung oder auf einem anderen in das Netzwerk eingebundenen Computer befinden, zugreifen. Im zweiten Fall geschieht dies über einen Mechanismus, der als Distributed COM oder DCOM bezeichnet wird. Weitere Informationen über Clients, Server und Schnittstellen finden Sie unter »Elemente einer COM-Anwendung« auf Seite 44-3.

Dieses Kapitel gibt eine konzeptuelle Übersicht über die Technologie, auf der Automatisierung und ActiveX-Steuerelemente basieren. In späteren Kapiteln werden Einzelheiten zum Erstellen von Automatisierungsobjekten und ActiveX-Steuerelementen in Delphi beschrieben.

Die Themen in diesem Abschnitt enthalten eine Übersicht zu den Konzepten der Technologie, auf der Automatisierung und ActiveX-Steuerelemente basieren.

COM als Spezifikation und Implementierung

COM stellt sowohl eine Spezifikation als auch eine Implementierung dar. In der COM-Spezifikation werden die Erstellung von Objekten und die Kommunikation zwischen Objekten definiert. Nach Maßgabe dieser Spezifikation können COM-Objekte in unterschiedlichen Sprachen erstellt sowie in unterschiedlichen Prozeßräumen und auf unterschiedlichen Plattformen ausgeführt werden. Solange die Objekte spezifikationskonform sind, können sie miteinander kommunizieren. Auf diese Weise kann älterer Quelltext (Legacy-Code) als Komponente mit neuen Komponenten integriert werden, die in objektorientierten Sprachen geschrieben wurden.

Die COM-Implementierung liegt in Form der COM-Bibliothek vor (hierzu gehören OLE32.dll und OLEAut32.dll), die eine Reihe von zentralen Diensten zur Verfügung stellt, welche die Grundspezifikation unterstützen. Die COM-Bibliothek enthält eine Reihe von Standard-Schnittstellen, welche die Hauptfunktionen eines COM-Objekts definieren. In der Bibliothek sind einige API-Funktionen enthalten, mit deren Hilfe COM-Objekte erstellt und verwaltet werden.

Hinweis Die Schnittstellenobjekte sowie die Sprache von Delphi sind an der COM-Spezifikation ausgerichtet. Die Implementation der COM-Spezifikation in Delphi wird Delphi ActiveX-Framework (DAX) genannt. Der Großteil der Implementierung ist in der Unit *AxCtrls* enthalten.

Durch die Verwendung von Delphi-Experten und VCL-Objekten in Ihrer Anwendung greifen Sie auf die Delphi-Implementierung der COM-Spezifikation zu. Zusätzlich stellt Delphi noch kapselnde Komponenten für diejenigen COM-Dienste zur Verfügung, die nicht direkt implementiert werden (wie z. B. Active-Dokumente). Diese kapselnden Komponenten sind in der Unit *ComObj* definiert, die API-Definitionen sind in der Unit AxCtrls enthalten.

COM-Erweiterungen

Im Laufe seiner Entwicklung wurde das COM-Modell weit über die grundlegenden COM-Dienste hinaus erweitert. COM dient als Basis für andere Technologien wie etwa Automatisierung, ActiveX-Steuerelemente, Active Server Pages und Active-Dokumente. Einzelheiten hierzu finden Sie im Abschnitt »COM-Erweiterungen« auf Seite 44-9.

Zusätzlich können jetzt COM-Objekte erstellt werden, die innerhalb der MTS-Umgebung (Microsoft Transaction Server) einsetzbar sind. MTS ist ein komponentengestütztes Transaktionsverarbeitungssystem zum Erstellen, Verteilen und Verwalten umfangreicher Server-Anwendungen für Intranet-Netzwerke und das Internet. Obwohl MTS hinsichtlich der Architektur kein Bestandteil von COM ist, erweitert es den Leistungsumfang von COM in umfangreichen verteilten Umgebungen. Informationen zu MTS finden Sie in Kapitel 51, »MTS-Objekte erstellen«.

Delphi stellt Experten zum schnellen und einfachen Implementieren von Anwendungen zur Verfügung, welche die obengenannten Technologien in der Delphi-Umgebung verkörpern. Einzelheiten hierzu finden Sie im Abschnitt »Implementieren von COM-Objekten mit Hilfe der Experten« auf Seite 44-18.

Elemente einer COM-Anwendung

Zur Implementierung einer COM-Anwendung müssen Sie folgendes angeben:

COM-Schnittstelle Die Art und Weise, in der ein Objekt seine Dienste extern für Clients bereitstellt. Ein COM-Objekt bietet eine Schnittstelle für jede Gruppe zugehöriger Methoden (Elementfunktionen) und Eigenschaften (Datenelemente und / oder Inhalt).

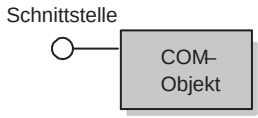
COM-Server Ein Modul (eine EXE-, DLL- oder OCX-Datei), das den Quelltext für ein COM-Objekt enthält. Die Objektimplementierungen sind in Servern enthalten. Ein COM-Objekt implementiert eine oder mehrere Schnittstellen.

COM-Client Den Quelltext, der die Schnittstellen zum Abrufen der angeforderten Server-Dienste aufruft. Die Clients wissen zwar, was sie (über die Schnittstelle) vom Server abrufen möchten, nicht aber, wie der Server die Dienste intern bereitstellt. Der am häufigsten implementierte COM-Client ist ein Automatisierungscontroller. Delphi vereinfacht den Prozeß beim Erstellen eines Clients durch die Möglichkeit, COM-Server (wie etwa ein Word-Dokument oder ein Powerpoint-Dia) als Komponenten in der Komponentenpalette zu installieren. Anschließend können Sie eine Verbindung zum Server herstellen und dessen Ereignisse mit dem Objektinspektor abfangen.

COM-Schnittstellen

Die COM-Clients kommunizieren mit den Objekten über COM-Schnittstellen. Bei den Schnittstellen handelt es sich um Gruppen von logisch oder semantisch zusammengehörenden Routinen, welche die Kommunikation zwischen einem Anbieter eines Dienstes (Server-Objekt) und seinen Clients ermöglichen. Standardmäßig wird eine COM-Schnittstelle wie folgt dargestellt:

Abbildung 44.1 Eine COM-Schnittstelle



Jedes COM-Objekt implementiert beispielsweise die grundlegende Schnittstelle *IUnknow*n, die dem Client mitteilt, welche Schnittstellen im Client zur Verfügung stehen.

Objekte können mehrere Schnittstellen aufweisen, von denen jede eine eigene Funktionalität implementiert. Eine Schnittstelle stellt eine Möglichkeit dar, dem Client mitzuteilen, welchen Dienst er zur Verfügung stellt, ohne daß Implementierungsdetails darüber geliefert werden, wie und wo das Objekt diesen Dienst zur Verfügung stellt.

Im folgenden sind die wichtigsten Merkmale von COM-Schnittstellen beschrieben:

- Sobald Schnittstellen als **published** deklariert wurden, sind sie absolut unveränderlich. Sie können sich darauf verlassen, daß eine Schnittstelle eine bestimmte Gruppe von Funktionen zur Verfügung stellt. Zusätzliche Funktionalität wird durch weitere Schnittstellen bereitgestellt.
- Vereinbarungsgemäß beginnen Schnittstellenbezeichner mit dem Großbuchstaben I, auf den ein symbolischer Name folgt, der die Schnittstelle definiert, wie z. B. *IMalloc* oder *IPersist*.
- Durch Verwendung eines **GUID (Globally Unique Identifier)** ist gewährleistet, daß Schnittstellen immer eindeutig identifizierbar sind. Bei einem GUID handelt es sich um eine zufällig generierte 128-Bit-Zahl. Schnittstellen-GUIDs werden auch **IIDs (Interface Identifiers)** genannt. Dadurch werden Benennungskonflikte zwischen unterschiedlichen Versionen eines Produkts oder zwischen verschiedenen Produkten vermieden.
- Schnittstellen sind sprachunabhängig. Sie können eine COM-Schnittstelle mit Hilfe jeder beliebigen Sprache implementieren, solange diese Sprache eine Struktur von Zeigern unterstützt und mit ihr entweder implizit oder explizit über einen Zeiger eine Funktion aufgerufen werden kann.
- Schnittstellen sind selbst keine Objekte, sondern sie stellen eine Möglichkeit zum Zugriff auf Objekte zur Verfügung. Daher greifen Clients nicht direkt, sondern über einen Schnittstellenzeiger auf Daten zu.
- Schnittstellen entstehen immer durch Vererbung von der grundlegenden Schnittstelle *IUnknown*.
- Schnittstellen können von COM über Proxy-Server so umgeleitet werden, daß Aufrufe der Schnittstellenmethoden zwischen Threads, Prozessen und im Netzwerk laufenden Computern erfolgen können, ohne daß die Client- bzw. Server-Objekte diese Umleitung bemerken. Weitere Informationen finden Sie unter Weitere Informationen finden Sie unter »In-Process-Server, Out-of-Process-Server und Remote-Server« auf Seite 44-7.

Die grundlegende COM-Schnittstelle IUnknown

Alle COM-Objekte müssen die grundlegende Schnittstelle *IUnknown* unterstützen. Sie enthält die folgenden Routinen:

QueryInterface	Stellt Zeiger zu anderen Schnittstellen zur Verfügung, die das Objekt unterstützt.
AddRef und *Release*	Einfache Referenzzählmethoden, die die Lebensdauer eines Objekts verfolgen, so daß ein Objekt sich selbst löschen kann, wenn der Client seinen Dienst nicht mehr benötigt.

Clients erhalten Zeiger auf andere Schnittstellen über die *IUnknown*-Methode *QueryInterface. QueryInterface* kennt alle Schnittstellen im Server-Objekt und kann einem Client einen Zeiger auf die angeforderte Schnittstelle liefern. Der Empfang eines Zei-

gers auf eine Schnittstelle signalisiert dem Client, daß er alle Methoden der Schnittstelle aufrufen kann.

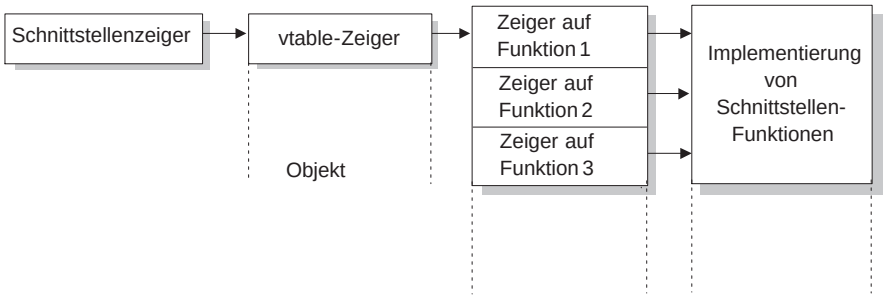
Objekte verfolgen ihre eigene Lebensdauer über die *IUnknown*-Methoden *AddRef* und *Release*, bei denen es sich um einfache Referenzzählmethoden handelt. Solange der Referenzzähler für ein Objekt ungleich Null ist, bleibt das Objekt im Speicher geladen. Sobald der Referenzzähler den Wert Null erreicht, kann die Schnittstellenimplementierung das bzw. die zugrundeliegende(n) Objekt(e) sicher aus dem Speicher entfernen.

COM-Schnittstellenzeiger

Ein Schnittstellenzeiger ist ein 32-Bit-Zeiger auf eine Objektinstanz, die ihrerseits auf die Implementierung der jeweiligen Methode in der Schnittstelle zeigt. Auf die Implementierung wird über ein Array von Zeigern zugegriffen, das **vtable** genannt wird. Diese virtuellen Funktionstabellen ähneln dem Mechanismus zur Unterstützung virtueller Funktionen in ObjectPascal dar.

Die *VTable* wird von allen Instanzen einer Objektklasse gemeinsam benutzt, so daß der Objekt-Quelltext jeder Objektinstanz eine zweite Struktur zuweist, die dessen **private**-Daten enthält. Der Schnittstellenzeiger des Clients fungiert dann als Zeiger auf *den Zeiger* zur *vtable*, wie aus der folgenden Abbildung zu ersehen ist.

Abbildung 44.2 Die Schnittstelle *vtable*



COM-Server

Bei einem COM-Server handelt es sich um eine Anwendung oder eine Bibliothek, die einer Client-Anwendung oder Client-Bibliothek Dienste zur Verfügung stellt. Ein COM-Server besteht aus einem oder mehreren COM-Objekten, wobei ein COM-Objekt eine Gruppe von Eigenschaften (Datenelemente oder Inhalt) und Methoden (oder Elementfunktionen) darstellt.

Den Clients ist nicht bekannt, *wie* ein COM-Objekt seinen Dienst durchführt, die Implementierung des Objekts bleibt verkapselt. Ein Objekt stellt seine Dienste über seine Schnittstellen zur Verfügung, wie weiter oben bereits beschrieben wurde.

Zudem brauchen die Clients nicht zu wissen, *wo* sich ein COM-Objekt befindet. COM bietet transparenten Zugriff unabhängig vom Standort des Objekts.

Wenn ein Client einen Dienst von einem COM-Objekt anfordert, übergibt er einen Klassenbezeichner (CLSID) an COM. Ein CLSID ist nichts weiter als ein GUID, der ein COM-Objekt referenziert. Über diese CLSID sucht COM die entsprechende Server-Implementierung, lädt den Quelltext in den Speicher und veranlaßt den Server, eine Objektinstanz für den Client zu instantiieren. Daher muß ein COM-Server ein Klassengeneratorobjekt (*IClassFactory*) zur Verfügung stellen, das auf Anforderung Instanzen von Objekten erzeugt. (Die CLSID basiert auf der GUID der Schnittstelle.)

Im allgemeinen muß ein COM-Server folgende Schritte ausführen:

- In der Registrierdatenbank Einträge registrieren, die das Server-Modul dem Klassenbezeichner (CLSID) zuordnen.
- Ein Klassengeneratorobjekt implementieren, wobei es sich um einen speziellen Typ von Objekt handelt, der ein anderes Objekt mit einer bestimmten CLSID erstellt.
- Den Klassengenerator für COM bereitstellen.
- Einen Mechanismus zur Verfügung stellen, über den ein Server, der gerade keine Clients bedient, aus dem Speicher entfernt werden kann.

Hinweis Mit den Delphi-Experten kann die Erstellung von COM-Objekten und -Servern automatisiert werden. Dies ist im Abschnitt »Implementieren von COM-Objekten mit Hilfe der Experten« auf Seite 44-18 beschrieben.

Hilfsklassen (CoClasses) und Klassengeneratoren

Ein COM-Objekt ist eine Instanz einer Hilfsklasse (CoClass), wobei es sich um eine Klasse handelt, die eine oder mehrere COM-Schnittstellen implementiert. Das COM-Objekt stellt die Dienste entsprechend der Definition durch seine Hilfsklassen-Schnittstellen zur Verfügung.

Hilfsklassen werden von einem speziellen Objekttyp, dem *Klassengenerator*, instantiiert. Immer wenn die Dienste eines Objekts von einem Client angefordert werden, erzeugt ein Klassengenerator eine Objektinstanz für den betreffenden Client und registriert diese. Wenn dann ein anderer Client die Dienste des Objekts anfordert, erzeugt der Klassengenerator eine weitere Objektinstanz für den zweiten Client.

Eine Hilfsklasse muß über einen Klassengenerator und einen Klassenbezeichner (CLSID) verfügen, so daß das zugehörige COM-Objekt extern, d. h. von einem anderen Modul aus, instantiiert werden kann. Die Verwendung dieser eindeutigen Bezeichner für Hilfsklassen ermöglicht die Aktualisierung der Hilfsklassen bei jeder neuen Implementierung von Schnittstellen in ihrer Klasse. Für eine neue Schnittstelle können Methoden geändert oder hinzugefügt werden, ohne daß ältere Versionen davon betroffen sind, im Gegensatz zur Verwendung von DLLs, wo dies ein häufig auftretendes Problem darstellt.

Die Delphi-Experten übernehmen das Implementieren und Instantiieren von Klassengeneratoren.

In-Process-Server, Out-of-Process-Server und Remote-Server

Bei COM braucht ein Client nicht zu wissen, wo sich ein Objekt befindet, denn es ruft einfach eine Objektschnittstelle auf, wobei die zum Aufrufen notwendigen Schritte von COM durchgeführt werden. Diese Schritte können unterschiedlich sein, je nachdem, ob sich das Objekt im selben Prozeß wie der Client, in einem anderen Prozeß auf dem Client-Computer oder auf einem anderen Computer im Netzwerk befindet. Es gibt die folgenden verschiedenen Server-Typen:

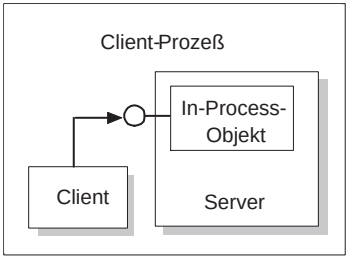
In-Process-Server Eine Bibliothek (DLL), die im *selben Prozeßraum* wie der Client ausgeführt wird. Beispiel: ein ActiveX-Steuerelement, das in eine WWW-Seite eingebettet ist, die wiederum im Browser Internet Explorer oder Netscape angezeigt wird. Das ActiveX-Steuerelement wird hierbei auf den Client-Computer heruntergeladen und innerhalb desselben Prozesses aufgerufen wie der WWW-Browser.
Der Client kommuniziert über direkte Aufrufe der COM-Schnittstelle mit dem In-Process-Server.

Out-of-Process-Server (bzw. lokaler Server) Eine andere Anwendung (EXE), die in einem *anderen Prozeßraum*, aber auf *demselben Computer* wie der Client ausgeführt wird. Beispiel: Eine Excel-Kalkulationstabelle, die in ein Word-Dokument eingebettet ist; diese beiden separaten Anwendungen werden auf demselben Computer ausgeführt.
Der lokale Server kommuniziert über COM mit dem Client.

Remote-Server Eine DLL oder eine andere Anwendung, die auf einem *anderen Computer* als dem ausgeführt wird, auf dem der Client läuft. Beispiel: Eine Delphi-Datenbankanwendung ist mit einem Anwendungsserver auf einem anderen Computer im Netzwerk verbunden.
Der Remote-Server verwendet zur Kommunikation mit dem Anwendungsserver verteilte COM-Schnittstellen (DCOM-Schnittstellen).

Wie in der folgenden Abbildung gezeigt, befinden sich die Zeiger auf Objektschnittstellen für In-Process-Server im selben Prozeßraum wie der Client, so daß COM die Objektimplementierung direkt aufrufen kann.

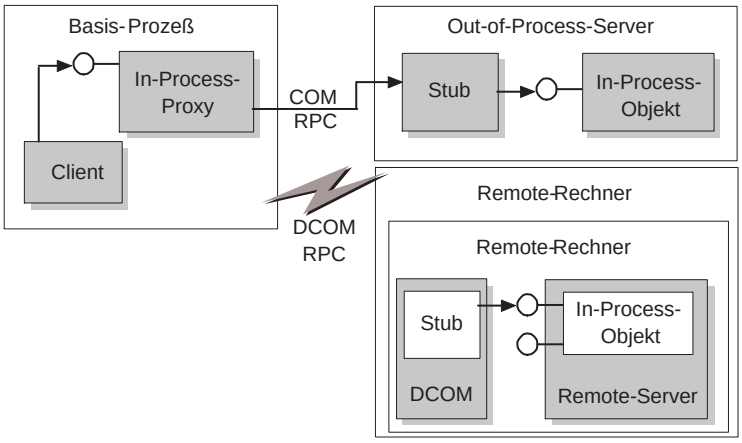
Abbildung 44.3 In-Process-Server



Wie in der folgenden Abbildung gezeigt, verwendet COM zum Einleiten von Remote-Prozedurenaufrufen einen Proxy-Server, wenn der Prozeß entweder in einem anderen Prozeßraum oder gar auf einem ganz anderen Computer ausgeführt wird. Der **Proxy-Server** befindet sich im selben Prozeßraum wie der Client, so daß aus Sicht des Clients alle Schnittstellenaufrufe gleich aussehen. Der Proxy-Server fängt den Aufruf des Clients ab und leitet ihn an die Stelle weiter, an der das tatsächliche Objekt ausgeführt wird. Der Mechanismus, der es dem Client ermöglicht, auf Objekte in einem anderen Prozeßraum oder sogar auf einem anderen Computer genauso zuzugreifen, als würden sie in ihrem eigenen Prozeßraum ausgeführt, wird Sequenzbildung (Marshaling) genannt.

Der Unterschied zwischen Out-of-Process-Servern und Remote-Servern besteht in der verwendeten Art der Kommunikation zwischen den einzelnen Prozessen. Der Proxy-Server verwendet COM zur Kommunikation mit einem Out-of-Process-Server, verteiltes COM (DCOM) zur Kommunikation mit einem Remote-Computer.

Abbildung 44.4 Out-of-Process-Server und Remote-Server



Der Sequenzbildungsmechanismus (Marshaling)

Unter Sequenzbildung (Marshaling) wird der Mechanismus verstanden, der es einem Client ermöglicht, Aufrufe von Schnittstellenfunktionen von Remote-Objekten in einem anderen Prozeßraum oder auf einem anderen Computer durchzuführen. Dieser Mechanismus funktioniert folgendermaßen:

- Der Mechanismus übernimmt einen Schnittstellenzeiger im Prozeß des Servers und stellt dem Quelltext im Client-Prozeß einen Zeiger auf einen Proxy-Server zur Verfügung.
- Der Mechanismus überträgt die Argumente eines Schnittstellenaufrufs in der Form, in der diese übergeben wurden, und stellt diese in den Prozeßraum des Remote-Objekts.

Für jeden Schnittstellenaufruf schiebt der Client Argumente auf einen Stack und führt über den Schnittstellenzeiger einen Funktionsaufruf durch. Wenn der Objektaufruf nicht innerhalb des Prozesses erfolgt, wird er an den Proxy-Server übergeben. Der Proxy-Server stellt die Argumente in ein Sequenzpaket und überträgt diese Struktur an das Remote-Objekt. Der Stub des Objekts »entpackt« das Paket wieder, schiebt die Argumente auf den Stack und ruft die Implementierung des Objekts auf. Das Wesentliche bei diesem Vorgang ist, daß das Objekt den Aufruf des Clients in seinem eigenen Prozeßraum neu erzeugt.

Welche Art von Sequenzbildung (Marshaling) durchgeführt wird, hängt davon ab, was das COM-Objekt implementiert. Die Objekte können einen Standardmechanismus zur Sequenzbildung verwenden, welcher von der Schnittstelle *IDispatch* zur Verfügung gestellt wird. Hierbei handelt es sich um einen generischen Mechanismus zur Sequenzbildung, der die Kommunikation über einen Remote-Prozedurenaufruf (RPC) gemäß dem Systemstandard ermöglicht. Einzelheiten zur Schnittstelle *IDispatch* finden Sie in Kapitel 47, »Automatisierungsserver erstellen«.

Hinweis Zusätzliche Unterstützung für Remote-Objekte bietet Microsoft Transaction Server (MTS). Einzelheiten hierzu finden Sie in Kapitel 51, »MTS-Objekte erstellen«.

COM-Clients

Es ist wichtig, eine COM-Anwendung zu entwickeln, in der die Clients die Schnittstellen von Objekten abfragen können, um die vom Objekt bereitgestellten Merkmale und Funktionen zu ermitteln. Die Server-Objekte sollten keine »feste Vorstellung« darüber haben, wie der Client die Objekte verwendet. Ferner brauchen die Clients nicht zu wissen, wie (oder sogar wo) ein Objekt seine Dienste zur Verfügung stellt. Sie brauchen sich nur darauf zu verlassen, daß das Objekt den Dienst zur Verfügung stellt, den es über die Schnittstelle bekanntgegeben hat.

Ein typischer COM-Client ist der Automatisierungs-Controller. Das ist derjenige Teil der Anwendung, der einen allgemeinen Überblick über deren Verwendungszweck hat. Er weiß, welche Arten von Information er von den einzelnen Objekten auf dem Server benötigt, und fordert bei Bedarf die betreffenden Dienste an.

Delphi erleichtert die Entwicklung eines Automatisierungs-Controllers, da Sie die Typbibliothek eines Automatisierungs-Servers importieren und in der Komponentenpalette installieren können.

Einzelheiten zum Erzeugen eines Automatisierungsservers finden Sie in Kapitel 46, »Automatisierungs-Controller erzeugen«.

COM-Erweiterungen

COM wurde ursprünglich konzipiert, um eine Kernfunktionalität für die Kommunikation zur Verfügung zu stellen und Möglichkeiten zur Erweiterungen zu bieten. Durch die Definition spezialisierter Schnittstellen für ganz bestimmte Zwecke wurde die Kernfunktionalität von COM selbst erweitert.

ActiveX ist eine Technologie, die COM-Komponenten, insbesondere Steuerelemente, kompakter und effizienter macht. Dies kommt vor allem Steuerelementen zugute, die in Intranet-Anwendungen eingesetzt werden sollen, die zunächst von einem Client heruntergeladen werden müssen.

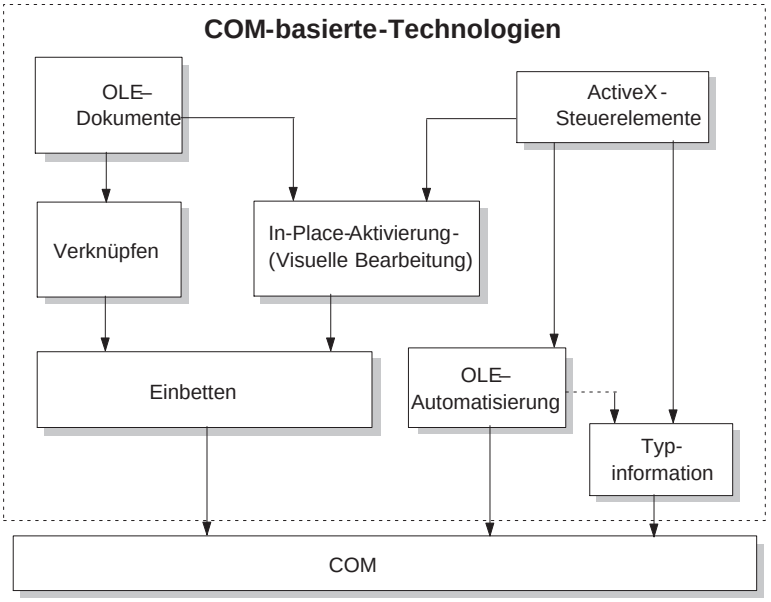
Nun ist Microsoft damit befaßt, einige der MTS-Technologien zur Erstellung komplexer Internet- und Intranet-Anwendungen im Rahmen von COM zu integrieren. Der nächste Schritt in der Entwicklung von COM, gegenwärtig als »COM+« (COM Plus) bezeichnet, sieht eine Integration weiterer neuer Merkmale vor und sollte mit der Veröffentlichung von Windows 2000 verfügbar werden.

Im folgenden ist die umfangreiche Palette von Diensten aufgeführt, die derzeit von den COM-Erweiterungen zur Verfügung gestellt werden. In den weiteren Abschnitten werden diese Dienste eingehender beschrieben.

Automatisierungs-Server	Unter Automatisierung wird hier die Fähigkeit einer Anwendung verstanden, Objekte in einer anderen Anwendung über die Programmierung zu steuern. Automatisierungsserver sind Objekte, die zur Laufzeit von anderen Anwendungen programmiert werden können.
Automatisierungs-Controller (bzw. COM-Clients)	Clients von Automatisierungsservern. Automatisierungs-Controller stellen eine Programmierumgebung zur Verfügung, in der ein Entwickler oder Benutzer Skripte zur Ausführung von Automatisierungsservern schreiben kann.
ActiveX-Steuerelemente	ActiveX-Steuerelemente sind spezialisierte In-Process-COM-Server, die in der Regel in eine Client-Anwendung eingebettet werden. Die Komponenten bieten ein definiertes Entwurfszeit- und Laufzeitverhalten sowie Ereignisse.
Typbibliotheken	Eine Zusammenstellung statischer Datenstrukturen, die oft als Ressourcen gespeichert werden und detaillierte Typinformationen über ein Objekt und seine Schnittstellen liefern. Die Clients von Automatisierungsservern und ActiveX-Steuerelementen erwarten, daß solche Typinformationen zur Verfügung stehen.
Active-Server-Seiten	Active-Server-Seiten sind ActiveX-Komponenten, mit deren Hilfe dynamische Web-Seiten erstellt werden können.
Active-Dokumente	Objekte, welche die Konzepte Verknüpfen und Einbetten, Ziehen und Ablegen, visuelle Bearbeitung und In-Place-Aktivierung unterstützen. Word-Dokumente und Excel-Kalkulationstabellen sind Beispiele für Active-Dokumente.
Prozeßübergreifende visuelle Objekte	Objekte, die über unterschiedliche Prozesse hinweg eingesetzt werden können.

Die folgende Abbildung veranschaulicht die Beziehung zwischen den COM-Erweiterungen und ihrer Erstellung auf COM-Basis.

Abbildung 44.5 COM-basierte Technologien



Die Verwendung von COM-Objekten ist sowohl mit größerer Funktionalität als auch mit Einschränkungen verbunden. ActiveX-Objekte können visuelle oder nicht-visuelle Objekte sein. Einige müssen im selben Prozeßraum ausgeführt werden wie ihre Clients, andere können in anderen Prozessen oder auf anderen Computern laufen, vorausgesetzt, sie stellen Unterstützung für die Sequenzbildung (Marshaling) zur Verfügung.

In der folgenden Tabelle ist zusammengefaßt, welche COM-Objekte Sie erstellen können, ob diese Objekte visuell sind oder nicht, in welchen Prozeßräumen sie ausgeführt werden können, wie sie die Sequenzbildung (Marshaling) Sequenzbildung (Marshaling) zur Verfügung stellen und ob sie eine Typbibliothek benötigen.

Tabelle 44.1 Anforderungen für COM-Objekte

Objekt	Visuelles Objekt?	Prozeßraum	Kommunikation	Typbibliothek
Active-Dokument	In der Regel ja	In-Process- oder Out-of-Process-Ausführung	OLE-Verben	Nein
Automatisierungsobjekt	Gelegentlich	In-Process-, Out-of-Process- oder Remote-Ausführung	Automatische Sequenzbildung mit Hilfe der Schnittstelle *IDispatch* (für Out-of-Process- und Remote-Server)	Für automatische Sequenzbildung erforderlich
ActiveX-Steuerelement	In der Regel ja	In-Process-Ausführung	Automatische Sequenzbildung mit Hilfe der Schnittstelle *IDispatch*	Erforderlich

Tabelle 44.1 Anforderungen für COM-Objekte (Fortsetzung)

Objekt	Visuelles Objekt?	Prozeßraum	Kommunikation	Typbibliothek
Benutzerdefiniertes Schnittstellenobjekt	Optional	In-Process-Ausführung	Für In-Process-Server keine Sequenzbildung erforderlich	Empfohlen
Benutzerdefiniertes Schnittstellenobjekt	Optional	In-Process-, Out-of-Process- oder Remote-Ausführung	Automatische Sequenzbildung über eine Typbibliothek, ansonsten manuelle Sequenzbildung mit Hilfe benutzerdefinierter Schnittstellen	Empfohlen

Automatisierungsserver und -Controller

Unter Automatisierung wird die Fähigkeit verstanden, Objekte in einer anderen Anwendung über die Programmierung so zu steuern, wie dies ein Makro tut, das in mehreren Anwendungen gleichzeitig arbeiten kann. Der Client eines Automatisierungsobjekts wird als Automatisierungs-Controller bezeichnet, das Server-Objekt, welches vom Client bearbeitet wird, heißt Automatisierungsobjekt.

Die Automatisierung kann für In-Process-, Out-of-Process- und Remote-Server verwendet werden.

Die Automatisierung ist durch die folgenden beiden Hauptmerkmale gekennzeichnet:

- Das Automatisierungsobjekt muß in der Lage sein, eine Reihe von Eigenschaften und Befehlen zu definieren und deren Funktionalität über Typbeschreibungen zu beschreiben. Zu diesem Zweck muß eine Möglichkeit vorhanden sein, Informationen über die Schnittstellen des Objekts, die Schnittstellenmethoden und die Argumente dieser Methoden zu liefern. In der Regel stehen diese Informationen in Typbibliotheken zur Verfügung. Der Automatisierungsserver kann Typinformationen auch dynamisch generieren, wenn er abgefragt wird.
- Automatisierungsobjekte müssen den Zugriff auf diese Methoden zur Verfügung stellen, so daß andere Anwendungen diese verwenden können. Hierfür müssen sie die Schnittstelle *IDispatch* implementieren. Über diese Schnittstelle kann ein Objekt seine gesamten Methoden und Eigenschaften bereitstellen. Durch die Primärmethoden dieser Schnittstelle können die Methoden des Objekts aufgerufen werden, nachdem sie zuvor über Typinformationen identifiziert wurden.

Für Entwickler, die nichtvisuelle OLE-Objekte erstellen und verwenden möchten, die in jedem Prozeßraum ausgeführt werden können, ist die Automatisierung gut geeignet. Einer der Gründe hierfür liegt darin, daß die Schnittstelle *IDispatch* den Prozeß der Sequenzbildung (Marshaling) automatisiert. Bei der Automatisierung gelten jedoch Einschränkungen bezüglich der verwendbaren Typen.

Eine Liste der für Typbibliotheken im allgemeinen und Automatisierungsschnittstellen im besonderen gültigen Typen finden Sie in Kapitel 50, »Mit Typbibliotheken arbeiten«

Einzelheiten zum Schreiben eines Automatisierungs-Controllers finden Sie in Kapitel 46, »Automatisierungs-Controller erzeugen«. Informationen zum Schreiben eines Automatisierungsservers sind in Kapitel 47, »Automatisierungsserver erstellen«, enthalten.

ActiveX-Steuerelemente

ActiveX-Steuerelemente sind visuelle Steuerelemente, die nur In-Process-Server ausführen; sie können in eine Container-Anwendung für ActiveX-Steuerelemente gestellt werden. Bei ActiveX-Steuerelementen handelt es sich nicht um vollständige Anwendungen im eigentlichen Sinne, man kann sie sich eher als vordefinierte OLE-Steuerelemente vorstellen, die in verschiedenen Anwendungen wiederverwendet werden können. ActiveX-Steuerelemente verwenden die Automatisierung zur Bereitstellung ihrer Eigenschaften, Methoden und Ereignisse. Die Funktionalität von ActiveX-Steuerelementen umfaßt die Fähigkeit, Ereignisse auszulösen, Bindungen zu Datenquellen herzustellen und die Lizenzierung zu unterstützen.

ActiveX-Steuerelemente werden in zunehmendem Maße als interaktive Objekte auf WWW-Seiten verwendet. In dieser Funktion hat sich ActiveX zu einem Standard insbesondere für interaktive Inhalte für das World Wide Web entwickelt, was auch die Verwendung von ActiveX-Dokumenten zur Anzeige von nicht im HTML-Format vorliegenden Dokumenten in WWW-Browsern mit einschließt. Weitere Informationen über die ActiveX-Technologie finden Sie auf der WWW-Seite von Microsoft über ActiveX.

Mit Hilfe der Delphi-Experten können Sie schnell und einfach ActiveX-Steuerelemente erzeugen. Weitere Informationen zur Erstellung und Verwendung dieser Objekttypen finden Sie in Kapitel 48, »ActiveX-Steuerelemente erstellen«.

Typbibliotheken

Typbibliotheken bieten eine Möglichkeit, weitere Typinformationen zu einem Objekt zu erhalten, als über eine Objektschnittstelle ermittelt werden können. Die in Typbibliotheken enthaltenen Typinformationen liefern benötigte Informationen über Objekte und deren Schnittstellen, z. B. die Auskunft, welche Schnittstellen für welche (anhand der CLSID identifizierten) Objekte vorhanden sind, welche Elementfunktionen für jede Schnittstelle existieren und welche Argumente für diese Funktionen benötigt werden.

Sie können Typinformationen entweder durch Abfragen einer gerade ausgeführten Instanz eines Objekts oder durch Laden und Lesen von Typbibliotheken abrufen. Diese Informationen ermöglichen Ihnen das Implementieren eines Clients, der ein bestimmtes Objekt verwenden soll, wobei Sie im einzelnen wissen, welche Elementfunktionen benötigt werden und was an diese Elementfunktionen übergeben werden muß.

Clients von Automatisierungsservern und ActiveX-Steuerelemente erwarten, daß Typinformationen zur Verfügung stehen. Automatisierungs- und ActiveX-Experten generieren automatisch eine Typbibliothek, wenn Sie mit ihnen Objekte erzeugen. Sie

können diese Art von Informationen mit Hilfe des Typbibliothekseditors anzeigen oder bearbeiten. Dies ist in Kapitel 50, »Mit Typbibliotheken arbeiten«, beschrieben.

Dieser Abschnitt beschreibt, welche Informationen eine Typbibliothek enthält, wie sie erzeugt wird, wann sie verwendet wird und wie auf sie zugegriffen wird. Am Ende dieses Abschnitts sind für Entwickler, die eine sprachübergreifende gemeinsame Benutzung von Schnittstellen vorsehen möchten, Vorschläge zur Verwendung von Typbibliothek-Tools aufgeführt.

Der Inhalt der Typbibliotheken

Typbibliotheken enthalten *Typinformationen*, die folgendes angeben: die Schnittstellen, die in den einzelnen COM-Objekten vorhanden sind, sowie die Typen und die Anzahl von Argumenten für die zugehörigen Schnittstellenmethoden. Diese Beschreibungen enthalten die eindeutigen Bezeichner für die Hilfsklassen (CLSIDs) und für die Schnittstellen (IIDs), so daß ordnungsgemäß auf diese zugegriffen werden kann, sowie die Weiterleitungsbezeichner (dispIDs) für die Methoden und Eigenschaften der Automatisierungsschnittstellen.

Typbibliotheken enthalten ferner die folgenden Informationen:

- Beschreibungen von individuellen Typinformationen, die benutzerdefinierten Schnittstellen zugeordnet sind.
- Routinen, die durch den Automatisierungs- oder ActiveX-Server exportiert werden, aber keine Schnittstellenmethoden sind.
- Informationen über die Datentypen für Aufzählungen, Records (Strukturen), Unions, Aliase und Module.
- Verweise auf Typbeschreibungen aus anderen Typbibliotheken.

Erzeugen von Typbibliotheken

Mit herkömmlichen Entwicklungs-Tools werden zum Erzeugen von Typbibliotheken Skripts in der IDL (Interface Definition Language) oder der ODL (Object Description Language) geschrieben und dann compiliert. Delphi generiert Typbibliotheken jedoch automatisch, wenn Sie entweder mit dem Experten für Automatisierungsserver oder mit dem Experten für ActiveX-Steuerelemente arbeiten. (Sie können eine Typbibliothek auch erzeugen, indem Sie im Hauptmenü *Datei / Neu / ActiveX / Typbibliothek wählen*.) Anschließend können Sie die Typbibliothek mit Hilfe des Typbibliothekseditors von Delphi anzeigen und problemlos bearbeiten. Delphi aktualisiert automatisch die entsprechenden Quelldateien, wenn die Typbibliothek gespeichert wird.

Der Typbibliothekseditor generiert automatisch und in der Regel als Ressource eine Standard-Typbibliothek, zusammen mit einer Delphi-Schnittstellendatei (.PAS-Datei), welche die Typdefinition in der ObjectPascal-Syntax enthält. Weitere Informationen zur Verwendung des Typbibliothekseditors zum Erzeugen von Schnittstellen und Hilfsklassen finden Sie in Kapitel 50, »Mit Typbibliotheken arbeiten«.

Wann werden Typbibliotheken eingesetzt?

Es ist wichtig, für jede Gruppe von Objekten, die für externe Benutzer bereitgestellt werden, eine Typbibliothek zu erzeugen. Beispiele:

- Für ActiveX-Steuerelemente wird eine Typbibliothek benötigt, die als Ressource in derjenigen DLL enthalten sein muß, welche die ActiveX-Steuerelemente enthält.
- Objekte, welche die V-Tabellen-Bindung (*vtable-Bindung*) von benutzerdefinierten Schnittstellen unterstützen, müssen in einer Typbibliothek beschrieben sein, weil V-Tabellen-Referenzen während des Compilierens benötigt werden. Einzelheiten über V-Tabellen und Bindung während des Compilierens finden Sie im Abschnitt »Automatisierungsobjekte für eine Anwendung erstellen« auf Seite 47-1.
- Anwendungen, die Automatisierungsserver implementieren, müssen eine Typbibliothek zur Verfügung stellen, so daß Clients eine frühe Bindung vornehmen können.
- Für Objekte, die aus Klassen instantiiert werden, welche die Schnittstelle *IProviderClassInfo* unterstützen, wie z. B. alle Nachkommen der VCL-Klasse *TTypedComObject*, muß eine Typbibliothek vorhanden sein.
- Typbibliotheken sind zur Identifizierung der zum OLE-Ziehen-und-Ablegen (Drag&Drop) verwendeten Objekte zwar nicht unbedingt erforderlich, aber doch hilfreich.

Wenn Sie Schnittstellen definieren, die nur für die interne Verwendung (innerhalb einer Anwendung) vorgesehen sind, brauchen Sie keine Typbibliothek zu erzeugen.

Zugriff auf Typbibliotheken

Die binäre Typbibliothek ist normalerweise Bestandteil einer Ressourcendatei (.RES) oder eine eigenständige Datei mit der Dateinamenserweiterung TBL. Sobald eine Typbibliothek erzeugt wurde, können Objekt-Browser, Compiler und ähnliche Tools über spezielle Typenschnittstellen darauf zugreifen:

Schnittstelle	Beschreibung
ITypeLib	Stellt Methoden für den Zugriff auf eine Bibliothek mit Typbeschreibungen zur Verfügung.
ITypeInfo	Liefert Beschreibungen individueller Objekte, die in einer Typbibliothek enthalten sind. Beispielsweise verwendet ein Browser diese Schnittstelle zum Extrahieren von Informationen über Objekte aus der Typbibliothek.
ITypeComp	Bietet eine schnelle Möglichkeit zum Zugriff auf Informationen, die Compiler beim Binden an eine Schnittstelle benötigen.

In Delphi können Typbibliotheken verwendet werden, die aus anderen Anwendungen importiert wurden. Die meisten VCL-Klassen, die für COM-Anwendungen verwendet werden, unterstützen diese grundlegenden Schnittstellen, mit denen Informationen aus Typbibliotheken und von gerade ausgeführten Instanzen eines Objekts gespeichert und abgerufen werden. Die VCL-Klasse *TTypedComObject* unterstützt Schnittstellen, welche Typinformationen zur Verfügung stellen, und wird als Grundlage für das ActiveX-Objekt-Framework verwendet.

Die Vorteile von Typbibliotheken

Auch wenn für Ihre Anwendung keine Typbibliothek erforderlich ist, bietet ihr Einsatz folgende Vorteile:

- Die Typprüfung kann bereits während des Compilierens ausgeführt werden.
- Sie können die frühe Bindung für die OLE-Automatisierung (im Gegensatz zum Aufruf über Varianten) und für Controller verwenden, die keine V-Tabellen unterstützen. Außerdem ermöglichen die dualen Schnittstellen die Verschlüsselung der dispIDs während des Compilierens und verbessern so die Ausführungsgeschwindigkeit zur Laufzeit.
- Mit Hilfe von Typ-Browsern kann die Bibliothek durchsucht werden, so daß die Objektmerkmale für die Clients sichtbar werden.
- Mit der Funktion *RegisterTypeLib* können die ausgewiesenen Objekte registriert werden.
- Mit Hilfe der Funktion *UnRegisterTypeLib* können Sie die Typbibliothek einer Anwendung aus der Systemregistrierung entfernen (deinstallieren).
- Der Zugriff auf lokale Server wird verbessert, weil die OLE-Automatisierung Informationen aus der Typbibliothek verwendet, um die Parameter zu packen, die in einem anderen Prozeß an das Objekt übergeben werden.

Tools für Typbibliotheken

Im folgenden sind die Tools zum Arbeiten mit Typbibliotheken aufgeführt.

- Mit TLIBIMP (Type Library Import) werden aus vorhandenen Typbibliotheken Delphi-Schnittstellendateien erzeugt. Dieses Tool ist in den Typbibliothekseditor integriert. TLIBIMP besitzt zusätzliche Konfigurationsoptionen, die nicht im Typbibliothekseditor verfügbar sind.
- Der Microsoft IDL-Compiler (MIDL) compiliert IDL-Skripts und erstellt daraus eine Typbibliothek.
- MKTYPLIB ist ein ODL-Compiler, der ODL-Skripts compiliert und daraus eine Typbibliothek erstellt (Sie finden dieses Tool in MS Win32 SDK).
- OLEView ist ein Typbibliothek-Browser, den Sie auf der WWW-Seite von Microsoft finden.
- TRegSvr ist ein Tool zum Registrieren und zum Aufheben der Registrierung von Servern und Typbibliotheken, das Bestandteil von Delphi ist. Der Quelltext von *TRegSvr* ist im Verzeichnis mit den Beispielen verfügbar.
- RegSvr32.exe (ein Standard-Hilfprogramm von Windows) dient zum Registrieren und zum Aufheben der Registrierung von Servern und Typbibliotheken.

Active-Server-Seiten

Die Technologie Active-Server-Seiten (Active Server Pages = ASP) ermöglicht das dynamische Erstellen von Web-Seiten mit Hilfe von ActiveX-Server-Komponenten. Mit ASP können Sie ActiveX-Steuerelemente in eine Web-Seite einbetten, die bei jedem Laden der Web-Seite durch den Server aufgerufen werden. Sie können beispielsweise ein Object-Pascal-Programm (zum Erstellen eines Bitmaps, zum Herstellen der Verbindung mit einer Datenbank usw.) schreiben. Dieses Steuerelement greift dann auf die Daten zu, die jedesmal aktualisiert werden, wenn der Server die Web-Seite lädt.

ASP benötigt die Umgebung Microsoft Internet Information Server (IIS), um für Web-Seiten genutzt werden zu können.

Auf dem Server sind die Active-Server-Seiten ActiveX-Komponenten, die Sie mit Delphi und den meisten anderen Sprachen wie C++, Java oder Visual Basic entwikkeln können. Auf dem Client sind Active-Server-Seiten HTML-Standarddokumente, die mit jedem Browser und unter jedem System angezeigt werden können.

Mit Hilfe der Delphi-Experten können Sie problemlos Active-Server-Seiten erstellen. Weitere Informationen zum Erstellen und Verwenden der Objekte dieses Typs finden Sie in Kapitel 49, »Eine Active-Server-Seite erstellen«

Active-Dokumente

Bei Active-Dokumenten (die früher als OLE-Dokumente bezeichnet wurden) handelt es sich um eine Reihe von COM-Diensten, welche die Konzepte »Verknüpfen und Einbinden« (Linking and Embedding), »Ziehen und Ablegen« (Drag&Drop) sowie visuelles Bearbeiten unterstützen. Active-Dokumente können nahtlos Daten oder Objekte in verschiedenen Formaten aufnehmen, beispielsweise Klangdateien, Kalkulationstabellen, Text und Bitmap-Grafiken.

Anders als ActiveX-Steuerelemente sind Active-Dokumente nicht auf In-Process-Server beschränkt, sondern können in prozeßübergreifenden Anwendungen verwendet werden.

Im Gegensatz zu Automatisierungsobjekten, die fast nie visuellen Charakter haben, können Active-Dokumentobjekte in anderen Anwendungen visuell aktiv sein. Sie gehören daher zu zwei Datenkategorien: zu den Präsentationsdaten, die zur visuellen Darstellung eines Objekts am Bildschirm oder über ein Ausgabegerät verwendet werden, und zu den nativen Daten, die zur Bearbeitung eines Objekts dienen.

Objekte, die als Active-Dokumente vorliegen, können Dokument-Container oder Dokument-Server sein. Delphi stellt zwar keinen Experten zum automatischen Erstellen von Active-Dokumenten zur Verfügung, aber Sie können mit der VCL-Klasse *TOleContainer* das Verknüpfen und Einbetten in vorhandenen Active-Dokumenten unterstützen.

Ferner lassen sich *TOleContainer* als Basis für einen Active-Dokument-Container verwenden. Zum Erstellen von Objekten für Active-Dokument-Server verwenden Sie eine der VCL-COM-Basisklassen und implementieren die entsprechenden Schnittstellen für diesen Objekttyp, je nachdem, welche Dienste das Objekt unterstützen muß.

Weitere Informationen zur Erzeugung und Verwendung von Active-Dokument-Servern finden Sie auf der WWW-Seite von Microsoft über ActiveX.

Hinweis Während die Spezifikation für Active-Dokumente über eine integrierte Unterstützung für die Sequenzbildung (Marshaling) bei prozeßübergreifenden Anwendungen verfügt, können Active-Dokumente nicht auf Remote-Servern ausgeführt werden, da sie mit spezifischen Typen für ein System auf einem bestimmten Computer arbeiten, beispielsweise mit Fenster-Handles, Menü-Handles usw.

Visuelle prozeßübergreifende Objekte

Automatisierungsobjekten, Active-Dokumenten und ActiveX-Steuerelementen begegnet man häufig. Weniger weit verbreitet sind OLE- bzw. ActiveX-Objekte, die in prozeßübergreifenden Anwendungen angezeigt und bearbeitet werden. Die Erstellung dieser Objektarten ist schwieriger, da das Kommunikationsprotokoll, das bei der visuellen Manipulation von Objekten in prozeßübergreifenden Anwendungen benutzt wird, nur für Objekte standardisiert ist, die Active-Dokumentschnittstellen benutzen. Sie müssen aus diesem Grund eine oder mehrere benutzerdefinierte Schnittstellen für Ihr Objekt implementieren und sind für die Sequenzbildung der Schnittstellen selbst verantwortlich.

Hier bieten sich zwei Vorgehensweisen an:

- Die Verwendung einer dualen *IDispatch*-Schnittstelle, für die eine automatische Sequenzbildung möglich ist (die einfachste und empfohlene Vorgehensweise). Der Automatisierungs-Experte erzeugt standardmäßig duale Schnittstellen, wenn Sie ein Automatisierungsobjekt erzeugen.
- Die manuelle Erstellung der Sequenzbildungsklassen durch Implementierung von *IMarshal* oder zugehöriger Schnittstellen.

Implementieren von COM-Objekten mit Hilfe der Experten

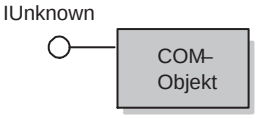
Delphi erleichtert das Schreiben von COM-Anwendungen durch die Bereitstellung von Experten. Die von diesen erstellten Delphi-Anwendungen können in der COM-Umgebung ausgeführt werden. Ferner bietet Delphi gesonderte Experten zum Erstellen folgender Objekte:

- Einfaches COM-Objekt
- Automatisierungsobjekt
- ActiveX-Steuerelement
- Active-Server-Seite
- ActiveX-Formular
- ActiveX-Bibliothek
- Eigenschaftsseite
- Typbibliothek

- MTS-Objekt

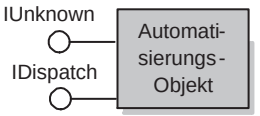
Die Experten dienen zur Automatisierung der Aufgaben, die zum Erstellen der einzelnen Typen von COM-Objekten durchgeführt werden müssen. Sie stellen die für jeden Objekttyp erforderlichen COM-Schnittstellen zur Verfügung, wie in Abbildung 44.6 gezeigt ist. So implementiert der Experte bei einem einfachen COM-Objekt die einzige erforderliche COM-Schnittstelle, *IUnknown*, die einen Schnittstellenzeiger auf das Objekt liefert.

Abbildung 44.6 Eine einfache COM-Objektschnittstelle



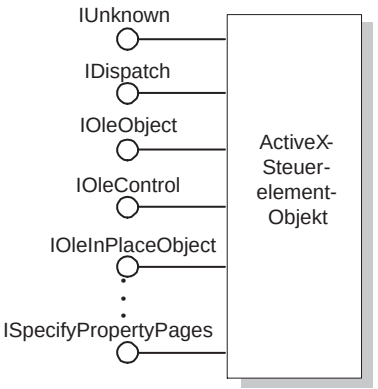
Für Automatisierungsobjekte implementiert der Experte die Schnittstellen *IUnknown* und *IDispatch*, wobei letztere eine automatische Sequenzbildung (Marshaling) zur Verfügung stellt.

Abbildung 44.7 Automatisierungs-Objektschnittstelle



Für ActiveX-Steuerelemente implementiert der Experte alle erforderlichen Schnittstellen für ActiveX-Steuerelemente ausgehend von *IUnknown, IDispatch, IOleObject, IOLEControl* usw. Eine vollständige Liste der Schnittstellen finden Sie auf der Referenzseite für das Objekt *TActiveXControl*.

Abbildung 44.8 AxtiveX-Steuerelement-Schnittstelle



Sie können sich die Experten als "Lieferanten" einer erweiterten Implementierung vorstellen. Wie in Tabelle 44.2 gezeigt, implementieren die einzelnen Experten die folgenden COM-Schnittstellen:

Tabelle 44.2 Die Delphi-Experten für die Implementierung von COM-, Automatisierungs- und ActiveX-Objekten

Experte	Implementierte Schnittstellen	Vom Experten durchgeführte Aufgaben
COM-Server	*Iunknown*	Exportieren der notwendigen Routinen für die Server-Registrierung, die Klassenregistrierung, das Laden und das Entfernen des Servers in den bzw. aus dem Speicher sowie die Objektinstantiierung. Erzeugen und Verwalten der Klassengeneratoren für auf dem Server implementierte Objekte. Anweisung an COM, die Objektschnittstellen auf Basis eines angegebenen Threading-Modells aufzurufen. Bereitstellung einer Typbibliothek, wenn angefordert.
Automatisierungsserver	*IUnkown, IDispatch*	Alle oben genannten und zusätzlich: Implementieren der von Ihnen angegebenen Schnittstelle, entweder als duale oder als Dispatch-Schnittstelle. Automatische Bereitstellung einer Typbibliothek.
ActiveX-Steuerelement	*IUnknown, IDispatch, IPersistStreamInit, IOleInPlaceActiveObject, IPersistStorage, IViewObject, IOleObject, IViewObject2, IOleControl, IPerPropertyBrowsing, IOleInPlaceObject, ISpecifyPropertyPages*	Alle oben genannten und zusätzlich: Implementieren der Eigenschaften, Methoden und Ereignisse für alle Schnittstellen in der Klasse *TActiveXControl*. Zeigt den Quelltexteditor an, so daß Sie das Objekt ändern können.
ActiveForm	Gleiche Schnittstellen wie ActiveX-Steuerelement	Alle oben genannten und zusätzlich: Implementieren der Eigenschaften, Methoden und Ereignisse für alle Schnittstellen in der Klasse *TActiveXControl*. Zeigt ein Formular an, so daß Sie eine Anwendung entwerfen können.
Active-Server-Objekt	*IUnknown, IDispatch*	Führt die Aufgaben eines Automatisierungsobjekt-Experten (siehe oben) durch und generiert eine ASP-Seite, die in einen Web-Browser geladen werden kann. Sie gelangen in den Typbibliothekseditor, in dem Sie die Eigenschaften und Methoden des Objekts nach Bedarf ändern können. Wenn Sie als Active-Server-Typ Ereignismethoden auf Seitenebene einstellen, werden *OnStartPage* und *OnEndPage* automatisch implementiert.

Tabelle 44.2 Die Delphi-Experten für die Implementierung von COM-, Automatisierungs- und ActiveX-Objekten (Fortsetzung)

Experte	Implementierte Schnittstellen	Vom Experten durchgeführte Aufgaben
ActiveX-Bibliothek	Standardmäßig keine	Erzeugen einer neuen ActiveX- oder COM-Server-DLL und Bereitstellung der notwendigen Exportfunktionen.
Eigenschaftenseite	*IUnknown, IPropertyPage*	Erzeugen einer neuen Eigenschaftsseite, die Sie im Formular-Designer bearbeiten können.
Typbibliothek	Standardmäßig keine	Erzeugen einer neuen Typbibliothek und deren Zuordnung zum aktiven Projekt.
MTS-Objekt	Die Methoden der Schnittstelle *IObjectControl, Activate, Deactivate* und *CanBePooled.*	Fügt eine neue Unit mit der MTS-Objektdefinition in das aktuelle Projekt ein, damit Clients in der MTS-Laufzeitumgebung auf den Server zugreifen können. Sie gelangen in den Typbibliothekseditor, in dem Sie die Eigenschaften und Methoden des Objekts nach Bedarf ändern können.

Wenn Sie zusätzliche COM-Objekte hinzufügen (oder vorhandene Implementierungen erneut implementieren) möchten, steht dem nichts im Wege. Um eine neue Schnittstelle zur Verfügung zu stellen, erzeugen Sie einen Nachkommen der Schnittstelle *IDispatch* und implementieren die erforderlichen Methoden in diesem Nachkommen. Zum Neuimplementieren einer Schnittstelle erzeugen Sie einen Nachkommen der betreffenden Schnittstelle und ändern diesen entsprechend ab.

Kapitel

45

Ein einfaches COM-Objekt erstellen

Zur Erstellung verschiedener COM-Objekte stehen in Delphi Experten zur Verfügung. Dieses Kapitel gibt einen Überblick über die Erstellung eines einfachen COM-Objekts wie etwa einer Shell-Erweiterung in der Delphi-Umgebung. Informationen zur Erstellung von Automatisierungs-Clients und Automatisierungs-Servern finden Sie in Kapitel 46, »Automatisierungs-Controller erzeugen«, und in Kapitel 47, »Automatisierungsserver erstellen«. Die Erstellung einer Active-Server-Seite wird in Kapitel 49, »Eine Active-Server-Seite erstellen« beschrieben. Ziel dieses Kapitels ist es nicht, detailliert die Erstellung von COM-Anwendungen zu beschreiben. Entsprechende Informationen finden Sie in der Dokumentation zu Microsoft Developer's Network (MSDN). Auch auf der Microsoft-WWW-Seite im Internet sind aktuelle Informationen zu diesem Thema zu finden.

Das Erstellen eines COM-Objekts im Überblick

Um ein einfaches, unkompliziertes COM-Objekt, wie beispielsweise eine Shell-Erweiterung, zu erstellen, arbeiten Sie mit dem COM-Objekt-Experten.Ein COM-Objekt kann als In-Process-Server, als Out-of-Process-Server oder als Remote-Server implementiert werden.

Der Experte für COM-Objekte führt die folgenden Aufgaben durch:

- Erstellen einer neuen Unit.
- Definition einer neuen, von *TCOMObject* abgeleiteten Klasse und Einrichtung des Konstruktors für die Klassengenerierung.

Zum Erstellen eines COM-Objekts sind die folgenden Schritte notwendig:

1. Entwerfen eines COM-Objekts.
2. Erstellen des COM-Objekts mit dem COM-Objekt-Experten.
3. Registrieren des COM-Objekts.

4 Testen des COM-Objekts.

Ein COM-Objekt entwerfen

Beim Entwerfen des COM-Objekts müssen Sie entscheiden, welche COM-Schnittstellen implementiert werden sollen. Der Experte stellt die Schnittstelle *IUnknown* zur Verfügung. Wie Sie zum Implementieren anderer COM-Schnittstellen vorgehen müssen, entnehmen Sie der MSDN-Dokumentation.

Ferner müssen Sie festlegen, ob es sich bei dem betreffenden Objekt um einen In-Process-Server, einen Out-of-Process-Server oder einen Remote-Server handeln soll. Bei In-Process- und Out-of-Process-Servern sowie bei Remote-Servern, die eine Typbibliothek verwenden, übernimmt COM die Übertragung der Daten (Marshaling) für Sie. In den übrigen Fällen müssen Sie für das Übertragen der Daten zu Out-of-Process-Servern selbst sorgen.

Weitere Informationen zu Server-Typen finden Sie im Abschnitt »In-Process-Server, Out-of-Process-Server und Remote-Server« auf Seite 44-7.

Ein COM-Objekt mit dem COM-Objekt-Experten entwerfen

Vor dem Erstellen eines COM-Objekts müssen Sie das Projekt für eine Anwendung erstellen oder öffnen, welche die zu implementierenden Funktionen enthält. Dabei kann es sich je nach Bedarf entweder um eine Anwendung oder eine ActiveX-Bibliothek handeln.

So rufen Sie den COM-Objekt-Experten auf:

1 Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.

2 Wählen Sie die Registerkarte *ActiveX*.

3 Doppelklicken Sie auf das Symbol für ein COM-Objekt.

Geben Sie im Experten folgendes an:

Klassenname	Geben Sie den Namen des zu implementierenden Objekts an.
Instantiierung	Geben Sie über einen Modus zum Instantiieren an, wie Ihr COM-Objekt gestartet werden soll. Einzelheiten hierzu finden Sie im Abschnitt »Instantiierungstypen für COM-Objekte« auf Seite 45-3. Hinweis: Die Angabe zur Instantiierung wird nicht berücksichtigt, wenn Ihr COM-Objekt nur als In-Process-Server verwendet wird.
Threading-Modell	Wählen Sie das gewünschte Threading-Modell , um anzugeben, wie Client-Anwendungen die COM-Objektschnittstelle aufrufen sollen. Weitere Informationen hierzu finden Sie unter »Ein Threading-Modell auswählen« auf Seite 45-3 **Hinweis**: Je nach gewähltem Threading-Modell erfolgt die Registrierung des Objekts auf unterschiedliche Weise. Achten Sie darauf, daß Ihre Objektimplementierung mit dem gewählten Modell kompatibel ist.
Implementierte Interfaces	Geben Sie die Namen der COM-Schnittstellen an, die von diesem COM-Objekt implementiert werden sollen.

Beschreibung	Geben Sie eine Beschreibung für das zu erstellende COM-Objekt an.
Typbibliothek einschließen	Aktivieren Sie dieses Feld, um eine Typbibliothek für dieses Objekt zu generieren. In einer Typbibliothek sind Typinformationen enthalten, die es Ihnen ermöglichen, jede Objektschnittstelle und ihre Methoden und Eigenschaften für Client-Anwendungen zur Verfügung zu stellen. Bei Aktivierung dieses Feldes wird automatisch auch *Schnitstelle als OleAutomation* aktiviert.
Schnittstelle als OleAutomation	Aktivieren Sie dieses Feld, um den Sequenzbildungscode verfügbar zu machen, der bei der Erstellung einer Typbibliothek generiert wird. COM weiß, wie die Sequenzbildung (Marshaling) für die in der Typbibliothek enthaltenen *automatisierungskompatiblen* Typen zu erfolgen hat und kann für Sie die Proxies und Stubs einrichten, so daß Sie Parameter an Out-of-Process-Server (.EXE) übergeben können. Weitere Informationen hierzu finden Sie unter »Der Sequenzbildungsmechanismus (Marshaling)« auf Seite 44-8.

Instantiierungstypen für COM-Objekte

Hinweis Wenn das COM-Objekt nur als In-Process-Server verwendet wird, werden die Angaben zur Instantiierung nicht berücksichtigt.

Wenn Ihre COM-Anwendung ein neues Objekt erstellt, kann dieses einen der folgenden Instantiierungstypen aufweisen:

Typ	Bedeutung
Intern	Das betreffende Objekt kann nur intern erstellt werden; eine externe Anwendung kann keine Instanz des Objekts direkt erstellen. Beispiel: Eine Textverarbeitungsanwendung kann ein Dokumentobjekt aufweisen, das nur durch Aufruf einer Methode dieser Anwendung erstellt werden kann.
Einfache Instanz	Gibt an, daß ein Objekt, nachdem es mit einer Anwendung verbunden wurde, aus der Öffentlichkeit entfernt wird, so daß keine andere Anwendung damit verbunden werden kann. Diese Option wird in der Regel für MDI-Anwendungen (Multiple Document Interface) verwendet. Wenn ein Client Dienste eines als einfache Instanz vorhandenen Objekts anfordert, werden alle Anforderungen von demselben Server verarbeitet. So wird jedesmal, wenn ein Benutzer das Öffnen eines neuen Dokuments in einer Textverarbeitungsanwendung anfordert, derselbe Anwendungsprozeß verwendet.
Mehrfache Instanz	Gibt an, daß mehrere Anwendungen mit einem Objekt verbunden werden können. Jedesmal, wenn ein Client einen Dienst anfordert, wird eine neue, gesonderte Instanz des Servers aufgerufen. So wird beispielsweise jedesmal, wenn ein Benutzer den Windows-Explorer öffnet, ein separater Explorer erstellt.

Ein Threading-Modell auswählen

Beim expertengestützten Erstellen eines Objekts wählen Sie ein Threading-Modell aus, das von Ihrem Objekt unterstützt wird. Durch Hinzufügen von Threading-Unterstützung zu COM-Objekten können Sie deren Leistungsfähigkeit erhöhen.

In Tabelle 45.1 sind die verschiedenen verfügbaren Threading-Modelle aufgelistet.

Tabelle 45.1 Threading-Modelle für COM-Objekte

Threading-Modell	Beschreibung	Vor- und Nachteile für die Implementierung
Einzelner Thread	Keine Threading-Unterstützung. Client-Anforderungen werden durch den Aufrufmechanismus serialisiert.	Clients werden nacheinander behandelt, so daß keine Threading-Unterstützung benötigt wird. Keine Leistungsvorteile.
Apartment-Modell (oder Einzel-Thread-Apartment)	Clients können die Methoden eines Objekts nur von dem Thread aus aufrufen, mit dem das Objekt erstellt wurde. Es können zwar verschiedene Objekte desselben Servers auf unterschiedlichen Threads aufgerufen werden, aber jedes Objekt wird nur von diesem einen Thread aufgerufen.	Instanzdaten sind sicher, globale Daten müssen geschützt werden. Hierzu müssen kritische Abschnitte oder eine andere Form der Serialisierung verwendet werden. Die lokalen Variablen des Threads sind auch bei mehreren Aufrufen zuverlässig. Geringfügige Leistungsvorteile. Objekte lassen sich einfach schreiben, die Erstellung von Clients ist u. U. problematisch. Hauptsächlich für Steuerelemente von Web-Browsern verwendet.
Frei (oder Multi-Threading-Apartment)	Clients können die Methoden jedes Objekts von jedem Thread aus jederzeit aufrufen. Objekte können eine beliebige Anzahl von Threads gleichzeitig bearbeiten.	Objekte müssen alle Instanzendaten und globalen Daten schützen. Hierzu müssen kritische Abschnitte oder eine andere Form der Serialisierung verwendet werden. Die lokalen Variablen des Threads sind bei mehreren Aufrufen *nicht* zuverlässig. Clients lassen sich einfach schreiben, das Erstellen von Objekten ist u. U. problematisch. Hauptsächlich für verteilte DCOM-Umgebungen verwendet.
Beides	Objekte können Clients unterstützen, die entweder das Apartment- oder das freie Modell verwenden.	Maximale Leistung und Flexibilität. Unterstützung beider Threading-Modelle, wenn Clients zur Leistungsverbesserung entweder das Single-Threading oder das freie Modell verwenden.

Sowohl die Client- als auch die Server-Seite der Anwendung teilt COM die Regeln mit, die für die Verwendung von Threads befolgt werden sollen, wenn COM initialisiert wird. COM vergleicht die Threading-Regeln der beiden Seiten und stellt bei Übereinstimmung eine direkte Verbindung zwischen beiden her, wobei davon ausgegangen wird, daß sich beide Seiten auch an die Regeln halten. Wenn die beiden Seiten mit unterschiedlichen Regeln arbeiten wollen, richtet COM eine Formatübertragung (Marshaling) zwischen beiden Seiten ein, so daß für jede Seite nur die Threading-Re-

geln sichtbar sind, die sie entsprechend ihren eigenen Angaben auch verarbeiten kann. Natürlich führt dies zu Leistungseinbußen, ermöglicht aber die Kommunikation bei Verwendung unterschiedlicher Threading-Modelle.

Hinweis Über das im Experten gewählte Threading-Modell wird festgelegt, wie das Objekt in der Registrierung eingetragen wird. Sie müssen dann dafür sorgen, daß die Objektimplementierung mit dem gewählten Threading-Modell kompatibel ist.

Dieses Threading-Modell gilt nur für In-Process-Server. Das Festlegen des Threading-Modells im Experten bewirkt, daß der Threading-Modell-Schlüssel im CLSID-Registrierungseintrag *InProcessServer32* gesetzt wird.

Out-of-Process-Server werden als EXE-Dateien registriert, und Sie müssen für die Implementierung des Threading-Modells selbst sorgen. Wenn die EXE-Datei einen Server enthält, der mit dem freien Threading-Modell arbeitet, wird COM von Delphi für freies Threading initialisiert. Das bedeutet, daß COM die erwartete Unterstützung für alle in der EXE-Datei enthaltenen Objekte bereitstellen kann, die mit dem freien oder mit dem Apartment-Modell arbeiten. Informationen über das manuelle Überschreiben des Threading-Verhaltens in EXE-Dateien finden Sie in der Online-Hilfe unter *CoInitFlags*.

Hinweis Lokale Variablen sind immer sicher, unabhängig davon, welches Threading-Modell verwendet wird. Dies liegt daran, daß lokale Variablen im Stack gespeichert werden und jeder Thread über einen eigenen Stack verfügt.

Ein Objekt schreiben, das das freie Threading-Modell unterstützt

Verwenden Sie das freie Threading-Modell und nicht das Apartment-Modell, wenn der Zugriff auf ein Objekt von mehr als einem Thread aus nötig ist. Ein häufiges Beispiel hierfür ist eine Client-Anwendung, die mit einem Objekt auf einem Remote-Computer verbunden ist. Wenn der Remote-Client eine Methode für das Objekt aufruft, empfängt der Server den Aufruf auf einem Thread aus dem Thread-Pool auf dem Server-Computer. Dieser Empfangs-Thread nimmt den Aufruf lokal für das tatsächliche Objekt vor. Da das Objekt das freie Threading-Modell unterstützt, kann der Thread die Methode direkt im Objekt aufrufen.

Wenn das Objekt in einem solchen Fall das Apartment-Threading-Modell unterstützt, hätte der Aufruf zu dem Thread zurückgeführt werden müssen, mit dem das Objekt erstellt worden ist, und das Ergebnis hätte an den Empfangs-Thread geliefert werden müssen, bevor es zum Client zurückgegeben worden wäre. Für diesen Ansatz ist also eine gesonderte Formatübertragung (Marshaling) erforderlich.

Wenn das freie Threading-Modell unterstützt werden soll, müssen Sie in Betracht ziehen, wie für die *einzelnen* Methoden auf die Instanzdaten zugegriffen werden kann. Wenn eine Methode in Instanzdaten schreibt, müssen Sie zum Schutz der Instanzdaten kritische Abschnitte oder eine andere Art der Serialisierung verwenden. Wahrscheinlich ist der Systemaufwand zum Serialisieren kritischer Aufrufe geringer als das Ausführen von COM-Marshaling-Code.

Wenn es sich bei den Instanzdaten um Nur-Lese-Daten handelt, wird keine Serialisierung benötigt.

Ein Objekt schreiben, das das Apartment-Threading-Modell unterstützt

Beim Implementieren des Apartment-Threading-Modells (mit Einzel-Thread) müssen Sie einige Regeln beachten:

- Der erste Thread in der Anwendung, der erstellt wird, ist der COM-Haupt-Thread. Dies ist in der Regel der Thread, mit dem WinMain aufgerufen wurde. Dies muß auch der letzte Thread sein, der die COM-Initialisierung wieder aufhebt.
- Für jeden Thread im Apartment-Threading-Modell muß eine Nachrichtenschleife vorhanden sein, und die Nachrichtenwarteschlange muß häufig abgefragt werden.
- Wenn ein Thread einen Zeiger zu einer COM-Schnittstelle erhält, können die Methoden der betreffenden Schnittstelle nur von diesem Thread aus aufgerufen werden.

Das Einzel-Thread-Apartment-Modell ist ein Kompromiß zwischen der Bereitstellung der kompletten Multi-Threading-Unterstützung beim freien Threading-Modell und keiner Threading-Unterstützung. Ein Server, der nach dem Apartment-Modell arbeitet, gewährleistet, daß ein serialisierter Zugriff auf alle globalen Daten (wie z. B. den Objektzähler) möglich ist. Der Grund dafür ist, daß verschiedene Objekte unter Umständen versuchen, von verschiedenen Threads aus auf die globalen Daten zuzugreifen. Die Instanzdaten der einzelnen Objekte sind jedoch sicher, weil die Methoden jeweils über denselben Thread aufgerufen werden.

In der Regel wird für die Steuerelemente in WWW-Browsern das Apartment-Threading-Modell verwendet, weil Browser-Anwendungen ihre Threads immer als Apartments initialisieren.

Allgemeine Informationen über Threads finden Sie in Kapitel 8, »Multithread-Anwendungen entwickeln«.

Ein COM-Objekt registrieren

Nach dem Erstellen eines COM-Objekts müssen Sie dieses registrieren, so daß es von den übrigen Anwendungen gefunden und verwendet werden kann.

Hinweis Vor dem Entfernen eines COM-Objekts aus dem System müssen Sie seine Registrierung aufheben.

So registrieren Sie ein COM-Objekt:

- Wählen Sie *Start / ActiveX-Server eintragen*.

So heben Sie die Registrierung eines COM-Objekts auf:

- Wählen Sie *Start / ActiveX-Server austragen*.

Alternativ können Sie den Befehl **tregsvr** in die Kommandozeile eingeben oder die Datei regsvr32.exe vom Betriebssystem aus ausführen.

Ein COM-Objekt testen

Das Testen eines COM-Objekts erfolgt je nach Art des erstellten Objekts auf unterschiedliche Weise. Nachdem Sie das Objekt erstellt haben, können Sie es testen, indem Sie die implementierten Schnittstellen verwenden, mit denen auf die Methoden der Schnittstellen zugegriffen wird.

So testen und debuggen Sie COM-Objekte:

1. Aktivieren Sie, falls erforderlich, auf der Registerseite *Compiler* des Dialogfelds *Projekt / Optionen* die Debugger-Informationen. Aktivieren Sie außerdem *Integrierte Fehlersuche* im Dialogfeld *Tools / Debugger-Optionen*.
2. Wählen Sie für einen In-Process-Server *Start / Parameter*.
3. Geben Sie in das Feld *Host-Anwendung* den Namen der Client-Anwendung ein, die die Dienste des COM-Objekts anfordern wird, und wählen Sie dann *OK*.
4. Wählen Sie *Start / Start*.
5. Setzen Sie Haltepunkte im COM-Objekt.
6. Lassen Sie die Client-Anwendung mit dem COM-Objekt in Interaktion treten.

Wenn die Haltepunkte erreicht werden, wird die Ausführung des COM-Objekts unterbrochen.

Kapitel

46

Automatisierungs-Controller erzeugen

Unter Automatisierung versteht man ein COM-Protokoll, über welches definiert wird, wie eine Anwendung auf ein Objekt zugreift, das sich innerhalb einer anderen Anwendung oder DLL befindet. Ein *Automatisierungs-Controller* ist eine Client-Anwendung, die einen Automatisierungsserver steuert. Dies geschieht über ein oder mehrere vom Server zur Verfgung gestellte Objekte, die die *IDispatch*-Schnittstelle implementieren.

Automatisierungs-Controller können in jeder Sprache geschrieben sein, die eine Implementierung von COM und Automatisierungsfunktionen vorsieht. Die meisten Automatisierungs-Controller sind derzeit in C++, Object Pascal (Delphi) oder Visual Basic geschrieben.

Beispiele für Automatisierungsserver sind Microsoft Word, Microsoft Excel und Internet Explorer. Diese Anwendungen können über Delphi-Anwendungen oder über andere Automatisierungs-Controller gesteuert werden.

Aufgrund der Flexibilität von Delphi ist es möglich, Anwendungen und DLLs als Automatisierungsserver oder -Controller mit einer Vielzahl von Anwendungen zu integrieren.

Dieses Kapitel bietet eine Übersicht zum Erstellen eines Automatisierungs-Controllers in der Delphi-Umgebung. Es kann jedoch keine vollständige Anleitung zum Schreiben der Controller-Anwendung für jeden Server-Typ enthalten. Wenn Sie spezielle Informationen hierzu benötigen, schlagen Sie in der Dokumentation zu Ihrer Server-Anwendung nach.

Informationen zur Erstellung eines Automatisierungsservers finden Sie in Kapitel 47, »Automatisierungsserver erstellen«.

In Delphi können Sie einen Automatisierungs-Controller erstellen, indem Sie die Typbibliothek eines Automatisierungsservers importieren und in der Komponentenpalette installieren.

Einen Automatisierungs-Controller durch Importieren einer Typbibliothek erzeugen

Sie können einen Automatisierungs-Controller erzeugen, indem Sie die Typbibliothek eines vorhandenen Automatisierungsservers importieren und die automatisch generierten Klassen zur Steuerung des Servers verwenden. Mit dem Dialogfeld *Typbibliothek importieren* installieren Sie den Server, der die Typbibliothek für die Komponentenpalette beschreibt und die Herstellung einer Verbindung sowie das Abfangen der Ereignisse mit dem Objektinspektor ermöglicht. Anschließend können Sie die Server-Eigenschaften im Quelltext ändern.

Zum Import der Typbibliothek eines Automatisierungs-Controllers führen Sie folgende Schritte aus:

1. Wählen Sie *Projekt / Typbibliothek importieren*.
2. Wählen Sie die Typbibliothek aus der angezeigten Liste.

   Das Dialogfeld enthält alle im System registrierten Bibliotheken. Wenn die gewünschte Typbibliothek nicht in der Liste aufgeführt ist, klicken Sie auf die Schaltfläche *Hinzufügen* und suchen nach der Typbibliotheksdatei. Markieren Sie die Datei, klicken Sie auf die Schaltfläche *OK*, und wiederholen Sie Schritt 2. Beachten Sie, daß es sich bei der Typbibliothek um eine einzelne Bibliotheksdatei (TLB, OLB) oder um einen Server handeln kann, der eine Typbibliothek zur Verfügung stellt (DLL, OCX, EXE).
3. Wählen Sie die Registerkarte in der Palette, auf der Sie den Server einfügen wollen.
4. Markieren Sie *Komponenten-Wrapper generieren*, um einen *TComponent*-Wrapper zu erstellen, mit dessen Hilfe Sie den von der Bibliothek beschriebenen Server in der Komponentenpalette installieren können.
5. Wählen Sie *Installieren*.

   Geben Sie die Zielposition für die Bibliothek an (vorhandenes oder neues Package). Diese Schaltfläche ist deaktiviert, wenn für die Typbibliothek keine Komponenten erstellt werden können.

Ein von der Typbibliothek beschriebener Server befindet sich jetzt in der Komponentenpalette. Mit dem Objektinspektor können Sie für den Server eine Ereignisbehandlungsroutine schreiben.

Damit der Automatisierungsserver über diesen Controller gesteuert werden kann, müssen Sie Quelltext in die implementierende Unit einfügen, der die Unterstützung des frühen (VTable) und des späten (Dispatch) Bindens bereitstellt. Die soeben erstellte Unit enthält Schnittstellen-Wrapper für VTable-Bindung für alle verfügbar gemachten Schnittstellen und Dispatch-Bindung für duale und Dispatch-Schnittstellen.

Ereignisse in einem Automatisierungs-Controller verarbeiten

Nachdem Sie einen Server in der Komponentenpalette installiert haben, können Sie mit Hilfe des Objektinspektors eine Ereignisbehandlungsroutine schreiben.

So werden Ereignisse unterstützt:

1. Ziehen Sie die gewünschte Serverkomponente aus der Komponentenpalette auf das Formular.
2. Wählen Sie die Komponente, und klicken Sie im Objektinspektor auf das Register *Ereignis*. Eine Liste der Ereignisse des Objekts wird angezeigt.
3. Klicken Sie doppelt auf das leere Feld neben dem Ereignisnamen. Delphi öffnet den Quelltexteditor mit dem Rumpf der Ereignisbehandlungsroutine, die Sie dann fertigstellen können.

Nach dem Implementieren der Ereignisbehandlungsroutine können Sie eine Verbindung mit einem Server herstellen.

Die Verbindung zu einem Server herstellen und trennen

Normalerweise wird die Verbindung zu einem Server über seine Hauptschnittstelle hergestellt. So wird die Verbindung zu Microsoft Word über die Komponente *WordApplication* hergestellt. Nachdem die Verbindung zur Hauptschnittstelle besteht, können Sie Verbindungen zu jeder anderen Anwendungskomponente (beispielsweise WordDocument oder WordParagraphFormat) mit Hilfe der Methode *ConnectTo* herstellen.

Nachstehend finden Sie die Ereignisbehandlungsroutine für das Ereignis *OnNewWorkBook* der Komponente *ExcelApplication*. In dieser Routine wird der Komponente *ExcelWorkbook* eine Arbeitsmappe zugewiesen.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
      ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkBook));
end;
```

Nach dem Importieren der Typbibliothek fügen Sie Quelltext in die Implementierungs-Unit ein, um den Server entweder über duale Schnittstellen (das gebräuchlichste Vorgehen) oder eine Dispatch- Schnittstelle zu steuern.

Hinweis Die beim Import einer Typbibliothek angelegten Importdateien (Bibname_TLB.CPP und *Bibname_TLB.H*) sollten als Read-only-Dateien betrachtet werden. Daran sind keinerlei Änderungen vorgesehen. Diese Dateien werden neu generiert, sobald die Typbibliothek aktualisiert wird. Änderungen werden dann überschrieben.

Wenn Server eine *Quit*-Methode verfügbar machen (beispielsweise *WordApplication* und *ExcelApplication*), wird der Quelltext zum Aufrufen dieser Methode in der Methode *Disconnect* generiert. Quit stellt eine Funktionalität bereit, die dem Klicken auf das Menü *Datei* zum Beenden der Anwendung entspricht. Wurde *AutoConnect* der Wert *True* zugewiesen, ruft der Anwendungsserver *Quit* auf, sobald der Client been-

det wird. Aus diesem Grund hat das Drücken der Taste *F1* - während im Objektinspektor *AutoQuit* markiert ist - keinerlei Auswirkungen.

Einen Automatisierungsserver über eine duale Schnittstelle steuern

Wenn Sie einen Automatisierungs-Controller durch Auswahl eines Objekts in der Komponentenpalette erstellen, stellt letzteres automatisch eine duale Schnittstelle bereit, da Delphi der Typbibliothek das VTable-Layout entnehmen kann. Sobald Sie eine Methode der Klasse aufrufen, wird automatisch die Verbindung zu einer Instanz von Word hergestellt.

Für einen Word-Server können Sie z. B. folgendermaßen eine Methode der Klasse *TWordApplication aufrufen:*

```
foo := TWordapplication
foo.DoSomething;
```

Natürlich kann auch die vorherige Möglichkeit zum Steuern eines Automatisierungsservers über eine duale Schnittstelle genutzt werden. Dieses Vorgehen ist jedoch aufwendiger. Sie müssen zunächst eine Schnittstelle deklarieren und dann mit der Methode *Create* der Proxy-Klasse eines CoClass-Clients initialisieren. Anschließend können Sie die Methoden des Schnittstellenobjekts aufrufen. Ein Beispiel:

```
foo : _Application
foo := CoWordApplication.Create
foo.DoSomething;
```

Die Schnittstelle sowie die Proxy-Klasse des CoClass-Clients werden in der Unit definiert, die automatisch beim Importieren einer Typbibliothek erzeugt wird. Die Namen der Schnittstellen beginnen mit »I«. So heißt z. B. die Hauptschnittstelle für Microsoft Word *IWordBasic*. Die Namen der Proxy-Klassen für *CoClass*-Clients beginnen mit Co. So hat beispielsweise die Haupt-Proxy-Klasse des *CoClass*-Clients für Microsoft Word den Namen *CoApplication*.

Informationen zu dualen Schnittstellen finden Sie unter »Automatisierungsschnittstellen« auf Seite 47-7.

Einen Automatisierungsserver über eine Dispatch-Schnittstelle steuern

Normalerweise verwenden Sie die duale Schnittstelle zum Steuern des Automatisierungsservers (siehe oben). Unter Umständen kann es aber erforderlich werden, den Automatisierungsserver über eine Dispatch-Schnittstellezu steuern. Gehen Sie folgendermaßen vor:

1. Deklarieren Sie in der Implementierungs-Unit des Automatisierungs-Controllers eine Dispatch-Schnittstelle.
2. Setzen Sie zur Steuerung des Automatisierungsservers Methoden des Dispatch-Schnittstellenobjekts ein.

Informationen zu Dispatch-Schnittstellen finden Sie unter »Automatisierungsschnittstellen« auf Seite 47-7.

Beispiel: Ein Dokument mit Microsoft Word drucken

In den folgenden Arbeitsschritten wird die Erzeugung eines Automatisierungs-Controllers demonstriert, der ein Dokument unter Verwendung von Microsoft Word 8 aus Office 97 ausdruckt.

Erzeugen Sie zunächst ein neues Projekt, das aus einem Formular, einer Schaltfläche und einem Dialogfeld zum Öffnen von Dateien (*TOpenDialog*) besteht. Diese Steuerelemente bilden den Automatisierungs-Controller.

Schritt 1: Delphi für dieses Beispiel vorbereiten

Delphi wird mit zahlreichen gebräuchlichen Servern wie Word, Excel und Powerpoint in der Komponentenpalette ausgeliefert, um Ihnen die Arbeit zu erleichtern. Das Importieren eines Servers wird am Beispiel Word beschrieben. Da sich der Server bereits in der Komponentenpalette befindet, müssen Sie in diesem ersten Schritt das Package mit dem Server Word entfernen, damit Sie es anschließend installieren können. Schritt 4 zeigt, wie Sie den ursprünglichen Zustand der Komponentenpalette wiederherstellen können.

So entfernen Sie Word aus der Komponentenpalette:

1. Wählen Sie *Komponente / Packages installieren*.
2. Klicken Sie auf *Borland Sample Automation Server*, und wählen Sie *Entfernen*.

   Die Registerkarte *Server* der Komponentenpalette enthält jetzt keinen der mit Delphi gelieferten Server mehr. (Wenn keine anderen Server importiert wurden, wird die Registerkarte ebenfalls aus der Komponentenpalette entfernt.)

Schritt 2: Die Typbibliothek von Word importieren

Führen Sie zum Importieren der Word-Typbibliothek folgende Schritte aus:

1. Wählen Sie *Projekt / Typbibliothek importieren*.
2. Gehen Sie im Dialogfeld *Typbibliothek importieren* folgendermaßen vor:
   1. Wählen Sie *Microsoft Office 8.0 Object Library*.

      Wenn *Word (Version 8)* nicht in der Liste aufgeführt ist, klicken Sie auf die Schaltfläche *Hinzufügen*, und wählen Sie in der Typbibliotheksdatei von Word die Datei MSWord8.OLB. Markieren Sie die Datei, klicken Sie auf die Schaltfläche *Hinzufügen*, und wählen Sie in der Liste *Word (Version 8*).
   2. Wählen Sie unter *Palettenseite* den Eintrag *Server*.
   3. Wählen Sie *Installieren*.

      Klicken Sie auf die Schaltfläche *Hinzufügen*, wenn *Word (Version 8)* nicht in der Liste angezeigt wird, wechseln Sie in das Verzeichnis PROGRAMME\MICRO-

SOFT OFFICE\OFFICE, wählen Sie die Word-Typbibliothek MSWORD8.OLB, klicken Sie erneut auf *Hinzufügen*, und wählen Sie anschließend *Word (Version 8)* in der Liste.

3 Aktivieren Sie die Registerkarte *Server*, wählen Sie *WordApplication*, und fügen Sie die Komponente in ein Formular ein.

4 Schreiben Sie eine Ereignisbehandlungsroutine für jedes Objekt, indem Sie die Registerkarte *Ereignisse* im Objektinspektor aktivieren und auf das Feld neben dem jeweiligen Ereignisnamen doppelklicken. Anschließend können Sie den Rumpf der Ereignisbehandlungsroutine im Quelltexteditor ergänzen. Geben Sie die folgenden Informationen an:

Schritt 3: Microsoft Word mit einer VTable- oder einer Dispatch-Schnittstelle steuern

Zur Steuerung von Microsoft Word können Sie eine VTable- oder eine Dispatch-Schnittstelle verwenden.

Ein VTable-Schnittstellenobjekt verwenden

Wenn Microsoft Word über ein VTable-Schnittstellenobjekt gesteuert werden soll, fügen Sie Quelltext in die Behandlungsroutine für das Ereignis *OnClick* der Schaltfläche des Automatisierungs-Controllers ein. Zu diesem Zweck rufen Sie einfach Methoden der soeben erstellten Klasse auf. Für Word handelt es sich um die Klasse *TWordApplication*.

Hinweis Die Word-Typbibliothek Word_TLB muß in die uses-Klausel des Automatisierungs-Controllers aufgenommen werden.

1 Erstellen und initialisieren Sie folgendermaßen ein Anwendungsobjekt mit Hilfe der VTable-Schnittstelle *_Application:*

```
var
FileName: OleVariant;
begin
      MyWord: _Application;
      WordApplication1.Create;
```

2 Rufen Sie die Methode *PrintOut* des Objekts WordBasic mit dem Punktoperator (.) auf, und geben Sie das Objekt mit Quit frei:

```
if OpenDialog1.Execute then
  begin
    FileName := OpenDialog1.Filename;
    WordApplication1.Documents.Open(FileName,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam);
   WordApplication1.ActiveDocument.PrintOut(EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam);
   WordApplication1.Quit(EmptyParam,EmptyParam,EmptyParam);
```

```
end;
```

Ein Dispatch-Schnittstellenobjekt verwenden

Wenn Sie zur Steuerung von Microsoft Word eine Dispatch-Schnittstelle einsetzen wollen, gehen Sie folgendermaßen vor, um der Ereignisbehandlungsroutine für *OnClick* der Schaltfläche des Automatisierungs-Controllers Quelltext zuzuordnen:

1 Erstellen und initialisieren Sie folgendermaßen ein Anwendungsobjekt mit Hilfe der Dispatch-Wrapper-Klasse *_ApplicationDisp* und einer ihrer Bindemethoden:

```
var
      MyWord: _ApplicationDisp;
      FileName: OleVariant;
begin
  MyWord:= CoApplication_.Create;
```

Hinweis Fügen Sie in die **uses**-Klausel der Implementierungsdatei des Automatisierungs-Controllers Word_TLB ein, um die Word-Typbibliothek aufzunehmen.

2 Rufen Sie die Methoden *Open*, *PrintOut* und *Quit* auf.

```
if OpenDialog.Execute then
begin
   FileName := OpenDialog1.Filename;
    MyWord.Documents.Open(FileName,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam);
    MyWord.ActiveDocument.PrintOut(EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam);
    MyWord.Quit(EmptyParam,EmptyParam,EmptyParam);
  end;
```

Schritt 4: Bereinigungsarbeiten

Nachdem Sie dieses Beispiel durchgeführt haben, können Sie den ursprünglichen Zustand von Delphi wiederherstellen.

1 Löschen Sie die Objekte auf der Registerkarte *Server*:

   1 Wählen Sie *Komponente / Palette konfigurieren* und anschließend die Registerkarte *Server*.

   2 Wählen Sie auf der Registerkarte alle zuvor hinzugefügten Objekte aus, und wählen Sie dann *Schließen*.

   3 Wählen Sie *Löschen*. Die Registerkarte *Server* wird nicht mehr angezeigt.

2 Installieren Sie das Package *Borland Sample Automation Server Components* neu:

   1 Wählen Sie *Komponente / Packages installieren*.

2 Klicken Sie auf *Hinzufügen*, wechseln Sie in das Delphi-Verzeichnis BIN, und wählen Sie das Package *Borland Sample Automation Server Components*. Klicken Sie auf *Öffnen*.

Weitere Informationen

Die neuesten Informationen über Dispatch- und duale Schnittstellen sowie über Proxy-Klassen von CoClass-Clients können Sie den Kommentaren der automatisch erzeugten Quelltextdatei entnehmen.

Kapitel

47

Automatisierungsserver erstellen

Ein *Automatisierungsserver* ist eine Anwendung, die ihre Funktionen anderen Client-Anwendungen, sogenannten Automatisierungs-Controllern, zur Verfügung stellt. Bei einem Controller kann es sich um jede Anwendung handeln, welche die Automatisierung unterstützt, wie Delphi, Visual Basic oder C++ Builder. Bei einem Automatisierungsserver kann es sich um eine Anwendung oder um eine Bibliothek handeln.

Dieses Kapitel zeigt, wie Automatisierungsserver mit dem Experten für Automatisierungsserver von Delphi erstellt werden. Damit sind Sie in der Lage, einer vorhandenen Anwendung für die Steuerung der Automatisierung Eigenschaften und Methoden bereitzustellen.

Zur Erstellung eines Automatisierungsservers für eine bestehende Anwendung sind folgende Schritte erforderlich:

- Erzeugen Sie ein Automatisierungsobjekt für die Anwendung.
- Stellen Sie die Eigenschaften und Methoden der Anwendung für die Automatisierung bereit.
- Registrieren Sie die Anwendung als Automatisierungsserver.
- Testen Sie die Anwendung, und entfernen Sie die Fehler.

Weitere Informationen zum Erstellen eines Automatisierungs-Controllers finden Sie in Kapitel 46, »Automatisierungs-Controller erzeugen«. Allgemeine Informationen zu COM-Technologien finden Sie in Kapitel 44, »COM-Technologien im Überblick«.

Automatisierungsobjekte für eine Anwendung erstellen

Ein Automatisierungsobjekt ist eine von *TAutoObject* abgeleitete Object Pascal-Klasse, welche die OLE-Automatisierung unterstützt, indem sie sich selbst für andere Anwendungen verfügbar macht. Da *TAutoObject* die OLE-Automatisierung unterstützt, enthalten auch alle von dieser abgeleiteten Klassen automatisch die Unterstüt-

zung für die OLE-Automatisierung. Zur Erstellung eines Automatisierungsobjekts verwenden Sie den Automatisierungsobjekt-Experten.

Vor der eigentlichen Erstellung des Automatisierungsobjekts öffnen oder erzeugen Sie das Projekt für die Anwendung, in der die bereitzustellenden Funktionen enthalten sind. Bei dem Projekt kann es sich je nach Bedarf um eine Anwendung oder eine ActiveX-Bibliothek handeln.

So starten Sie den Automatisierungsexperten:

1. Wählen Sie *Datei / Neu*.
2. Aktivieren Sie die Registerkarte *ActiveX*.
3. Doppelklicken Sie auf das Symbol *Automatisierungsobjekt*.

Geben Sie im Experten folgendes an:

CoClass-Name	Geben Sie die Klasse an, deren Eigenschaften und Methoden Sie den Client-Anwendungen zur Verfügung stellen wollen. (Delphi stellt diesem Namen ein T voran.)
Instantiierung	Geben Sie über einen Modus zum Instantiieren instancingan, der bestimmt, wie Ihr Automatisierungsserver gestartet werden soll. Einzelheiten hierzu finden Sie im Abschnitt »Instantiierungstypen für COM-Objekte« auf Seite 45-3. **Hinweis**: Die Angabe zur Instantiierung wird nicht berücksichtigt, wenn Ihr Automatisierungsobjekt nur als In-Process-Server verwendet wird.
Threading-Modell	Wählen Sie das gewünschte Threading-Modell, um anzugeben, wie Client-Anwendungen die COM-Objektschnittstelle aufrufen sollen. Hierbei handelt es sich um das Threading-Modell, das Sie bei der Implementierung des Automatisierungsobjekts verwenden. Weitere Informationen zu den Threading-Modellen finden Sie unter »Ein Threading-Modell auswählen« auf Seite 45-3. **Hinweis**: Je nach gewähltem Threading-Modell erfolgt die Registrierung des Objekts auf unterschiedliche Weise. Achten Sie darauf, daß Ihre Objektimplementierung mit dem gewählten Modell kompatibel ist.
Ereignisunterstützung generieren	Aktivieren Sie dieses Kontrollkästchen, um den Experten anzuweisen, eine separate Schnittstelle zur Ereignisverwaltung für das Automatisierungsobjekt zu implementieren.

Nachdem Sie diese Angaben gemacht haben, wird zum aktuellen Projekt eine neue Unit hinzugefügt, welche die Definition für das Automatisierungsobjekt enthält. Außerdem fügt der Experte ein Typbibliothek-Projekt hinzu und öffnet die Typbibliothek. Nun können Sie über die Typbibliothek die Eigenschaften und Methoden für die Schnittstelle bereitstellen, wie im folgenden beschrieben.

Das Automatisierungsobjekt implementiert eine *duale Schnittstelle*, welche sowohl frühes Binden (zum Zeitpunkt der Compilierung) über die VTable als auch spätes

Binden (zur Laufzeit) über die Schnittstelle *IDispatch* unterstützt. Weitere Informationen finden Sie im Abschnitt »Duale Schnittstellen« auf Seite 47-7.

Ereignisse in Ihrem Automatisierungsobjekt verwalten

Der Automatisierungsexperte generiert automatisch Ereigniscode, wenn Sie im Dialogfeld *Automatisierungsobjekt-Experte* die Option zum Generieren von Unterstützungscode aktiviert haben.

Damit ein Server Ereignisse unterstützen kann, stellt er eine Definition einer nach außen gerichteten Schnittstelle bereit, die von einem Client implementiert wird. Der Client ermittelt die verfügbaren nach außen gerichteten Schnittstellen, indem die Serverschnittstelle *IConnectionPointContainer* abgefragt wird. Mit Hilfe der Methoden der Serverschnittstelle *IConnectionPoint* übergibt der Client dem Server einen Zeiger auf die Client-Implementierung der Ereignisse. Der Server muß eine Liste dieser Ereignisse verwalten und die entsprechenden Methoden aufrufen, sobald eines dieser Ereignisse auftritt. Wenn Sie Ereignisunterstützung generieren wählen, generiert Delphi den Quelltext zur Unterstützung von *IConnectPoint* und *IConnectPointContainer* automatisch.

Eigenschaften, Methoden und Ereignisse einer Anwendung für die Automatisierung bereitstellen

Wenn Sie einen Automatisierungsserver mit dem Automatisierungsexperten erstellen, wird automatisch eine Typbibliothek generiert, mit deren Hilfe eine Host-Anwendung ermitteln kann, welche Operationen ein Objekt ausführen kann. Um die Eigenschaften, Methoden und Ereignisse einer Anwendung verfügbar zu machen, müssen Sie die Typbibliothek des Automatisierungsservers wie unten beschrieben bearbeiten.

Eine Eigenschaft für die Automatisierung bereitstellen

Eine Eigenschaft ist eine Elementfunktion, mit der Informationen über den Zustand des Objekts, wie z. B. Farbe oder Schriftart, festgelegt oder ermittelt werden. Für ein Schaltflächen-Steuerelement könnte eine Eigenschaft z. B. folgendermaßen deklariert sein:

```
property Caption: WideString;
```

Eine Eigenschaft wird folgendermaßen für die Automatisierung bereitgestellt:

1 Wählen Sie im Typbibliothekseditor die Dispatch-Schnittstelle für das Automatisierungsobjekt.

  Die Dispatch-Schnittstelle hat denselben Namen wie das Automatisierungsobjekt, aber mit vorangestelltem *I*. Sie können den Standardwert ermitteln, indem Sie im Typbibliothekseditor auf den Registerkarten *CoClass* und Implementierung die Li-

ste der implementierten Schnittstellen nach derjenigen durchsuchen, die mit Vorgabe markiert ist.

2. Um die Eigenschaft für das Lesen / Schreiben bereitzustellen, klicken Sie in der Werkzeugleiste auf die Schaltfläche *Eigenschaft*. Wenn Sie auf den Pfeil neben dieser Schaltfläche klicken, wird eine Liste mit den verfügbaren Eigenschaften angezeigt, aus der Sie dann die gewünschte auswählen können.
3. Geben Sie in der Registerkarte *Attribute* den Namen der Eigenschaft an.
4. Bestimmen Sie in der Registerkarte *Parameter* den Rückgabetyp der Eigenschaft, und fügen Sie die entsprechenden Parameter hinzu.
5. Klicken Sie in der Werkzeugleiste auf die Schaltfläche *Aktualisieren*.

   In die Unit-Dateien des Automatisierungsobjekts werden daraufhin eine Definition und eine vorgefertigte Implementierung für die Eigenschaft eingefügt.
6. Fügen Sie in der vorgefertigten Implementierung für die Eigenschaft zwischen der **try**- und der **finally**-Anweisung den Quelltext für die gewünschte Funktionalität ein. In vielen Fällen muß nur eine vorhandene Funktion innerhalb der Anwendung aufgerufen werden.

Eine Methode für die Automatisierung bereitstellen

Bei einer Methode kann es sich um eine Prozedur oder um eine Funktion handeln. So stellen Sie eine Methode für die Automatisierung bereit:

1. Wählen Sie im Typbibliothekseditor die Dispatch-Schnittstelle für das Automatisierungsobjekt.

   Die Dispatch-Schnittstelle hat denselben Namen wie das Automatisierungsobjekt, aber mit vorangestelltem *I*. Sie können den Standardwert ermitteln, indem Sie im Typbibliothekseditor auf den Registerkarten *CoClass* und Implementierung die Liste der implementierten Schnittstellen nach derjenigen durchsuchen, die mit Vorgabe markiert ist.
2. Klicken Sie auf die Schaltfläche *Methode*.
3. Geben Sie im Bereich *Attribute* den Namen der Methode an.
4. Bestimmen Sie im Bereich *Parameter* den Rückgabetyp der Methode, und fügen Sie die entsprechenden Parameter hinzu.
5. Klicken Sie in der Werkzeugleiste auf die Schaltfläche *Aktualisieren*.

   In die Unit-Dateien des Automatisierungsobjekts werden daraufhin eine Definition und eine vorgefertigte Implementierung für die Methode eingefügt.
6. Fügen Sie in der vorgefertigten Implementierung für die Methode zwischen der **try**- und der **finally**-Anweisung den Quelltext für die gewünschte Funktionalität ein. In vielen Fällen muß nur eine vorhandene Funktion innerhalb der Anwendung aufgerufen werden.

Ein Ereignis für die Automatisierung bereitstellen

So stellen Sie ein Ereignis für die Automatisierung bereit:

1. Aktivieren Sie im Automatisierungsexperten das Kontrollfeld *Ereignisunterstützung generieren*.

   Der Experte erzeugt ein Automatisierungsobjekt, das eine Ereignisschnittstelle enthält.

2. Wählen Sie im Typbibliothekseditor die Ereignisschnittstelle für das Automatisierungsobjekt.

   Die Ereignisschnittstelle weist den Namen des Automatisierungsobjekts mit vorangestelltem Buchstaben »I« und nachgestelltem Wort »Events« auf.

3. Klicken Sie in der Werkzeugleiste der Typbibliothek auf die Schaltfläche *Methode*.

4. Geben Sie im Bereich *Attribute* den Namen der Methode an, wie z. B. *MyEvent*.

5. Klicken Sie in der Werkzeugleiste auf die Schaltfläche *Aktualisieren*.

   In die Unit-Dateien des Automatisierungsobjekts werden eine Definition und eine Dummy-Implementierung für das Ereignis eingefügt.

6. Erzeugen Sie im Quelltext-Editor eine Ereignisbehandlungsroutine innerhalb des Nachkommens von *TAutoObject* in der Automatisierungsobjektklasse. Beispiel:

```
unit ev;
interface
uses
      ComObj, AxCtrls, ActiveX, Project1_TLB;
type
      TMyAutoObject = class (TAutoObject,IConnectionPointContainer, IMyAutoObject)
private
      ...
public
      procedure Initialize; override;
      procedure EventHandler;          { Ereignisbehandlungsroutine hinzufügen }
```

7. Weisen Sie am Ende der Methode *Initialize* der soeben erzeugten Ereignisbehandlungsroutine ein Ereignis zu. Beispiel:

```
procedure TMyAutoObject.Initialize;
begin
      inherited Initialize;
      FConnectionPoints:= TConnectionPoints.Create(Self);
      if AutoFactory.EventTypeInfo <> nil then
         FConnectionPoints.CreateConnectionPoint (AUtoFactory.EventIID,
            ckSingle, EventConnect);
      OnEvent = EventHandler; { Der Ereignisbehandlungsroutine ein Ereignis zuweisen }
      end;
```

8. Fügen Sie den erforderlichen Quelltext zum Aufrufen der implementierten Methode hinzu. Ersetzen Sie z. B. mit dem folgenden Quelltext »MyEvent« durch den Namen Ihres Ereignisses.

```
procedure TMyAutoObject.EventHandler;
```

```
begin
      if FEvents <> nil then FEvents.MyEvent;      { Implementierte Methode aufrufen.}
end;
```

Weitere Informationen

Wenn Sie weitere Informationen zum Typbibliothekseditor benötigen, drücken Sie an einer beliebigen Stelle im Editor *F1*.

Eine Anwendung als Automatisierungsserver registrieren

Ein Automatisierungsserver kann als In-Process- oder als Out-of-Process-Server registriert werden. Informationen über diese Server-Typen finden Sie unter »In-Process-Server, Out-of-Process-Server und Remote-Server« auf Seite 44-7.

Hinweis Bevor Sie einen Automatisierungsserver aus dem System entfernen, sollten Sie zuerst die Registrierung entfernen, indem Sie die zugehörigen Einträge in der Windows-Registrierung löschen.

Einen In-Process-Server registrieren

Zur Registrierung eines In-Process-Servers (DLL oder OCX) gehen Sie folgendermaßen vor:

- Wählen Sie *Start / ActiveX-Server eintragen*.

Mit dem folgenden Schritt heben Sie die Registrierung eines In-Process-Servers auf:

- Wählen Sie *Start / ActiveX-Server austragen*.

Einen Out-of-Process-Server registrieren

Zur Registrierung eines Out-of-Process-Servers gehen Sie folgendermaßen vor:

- Starten Sie den Server mit der Kommandozeilenoption /regserver. (Die Kommandozeilenoptionen können Sie in dem Dialogfeld festlegen, das nach Auswahl von *Start / Parameter* angezeigt wird.)

  Sie können den Server auch während der Ausführung registrieren.

Mit dem folgenden Schritt heben Sie die Registrierung eines Out-of-Process-Servers auf:

- Starten Sie den Server mit der Kommandozeilenoption /unregserver.

Die Anwendung testen und Fehler entfernen

Um einen Automatisierungsserver zu testen und eventuelle Fehler zu entfernen, gehen Sie folgendermaßen vor:

1. Aktivieren Sie, falls nötig, den Debugger. Die entsprechende Option befindet sich auf der Registerkarte *Compiler* des Dialogfeldes *Projektoptionen*. Schalten Sie ferner die Option *Integrierter Debugger* in dem über *Tools / Debugger-Optionen* aufgerufenen Dialogfeld ein.
2. Bei einem In-Process-Server wählen Sie *Start / Parameter*, geben den Namen des Automatisierungs-Controllers in das Feld *Host-Anwendung* ein und bestätigen mit *OK*.
3. Wählen Sie *Start / Start*.
4. Setzen Sie Haltepunkte im Automatisierungsserver.
5. Verwenden Sie den Automatisierungs-Controller zur Interaktion mit dem Automatisierungsserver.

Die Ausführung des Automatisierungsservers wird unterbrochen, wenn dieser einen Haltepunkt erreicht.

Automatisierungsschnittstellen

Die Delphi-Experten implementieren standardmäßig eine duale Schnittstelle. Dies bedeutet, daß das Automatisierungsobjekt folgende Bindevorgänge unterstützt:

- Spätes Binden zur Laufzeit über die Schnittstelle *IDispatch*. Hierbei wird eine Dispatch-Schnittstelle oder *dispinterface* implementiert.
- Frühes Binden beim Compilieren. Dies erfolgt durch direkten Aufruf einer der Elementfunktionen aus der Tabelle der virtuellen Funktionen (VTable) des Objekts. Dies wird als benutzerdefinierte Schnittstelle bezeichnet.

Duale Schnittstellen

Unter einer dualen Schnittstelle versteht man eine Schnittstelle, die gleichzeitig benutzerdefinierte Schnittstelle und Dispatch-Schnittstelle (dispinterface) ist. Sie wird als COM-VTable-Schnittstelle implementiert, die von *IDispatch* abgeleitet ist. Für Controller, die nur zur Laufzeit auf das Objekt zugreifen, steht die Dispatch-Schnittstelle zur Verfügung; für Objekte, welche die Vorteile des frühen Bindens beim Compilieren nützen können, wird die effektivere benutzerdefinierte VTable-Schnittstelle verwendet.

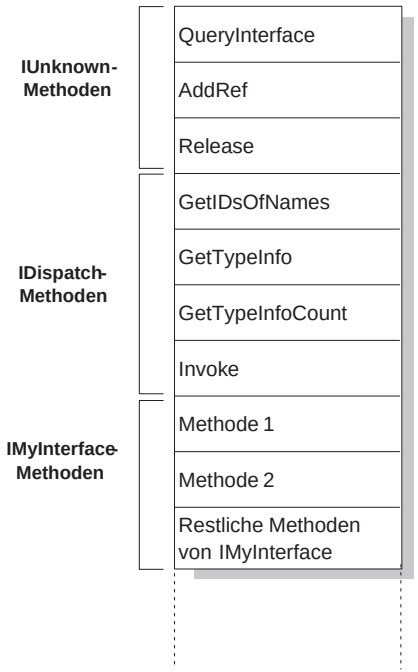
Duale Schnittstellen bieten die Vorteile von benutzerdefinierten VTable-Schnittstellen und Dispatch-Schnittstellen:

- Bei VTable-Schnittstellen führt der Compiler eine Typprüfung durch und stellt informativere Fehlermeldungen zur Verfügung.

- Bei Automatisierungs-Controllern, die keine Typinformationen erhalten können, stellt die Dispatch-Schnittstelle zur Laufzeit Zugriff auf das Objekt zur Verfügung.
- Bei In-Process-Servern können Sie den schnellen Zugriff über die VTable-Schnittstellen nutzen.
- Bei Out-of-Process-Servern führt COM die Sequenzbildung der Daten (Marshaling) sowohl für die *VTable*-Schnittstellen als auch für die Dispatch-Schnittstellen durch. COM stellt eine generische Proxy-/ Stub-Implementierung zur Verfügung, welche eine Sequenzbildung (Marshaling) für die Schnittstelle anhand der in einer Typbibliothek enthaltenen Informationen vornehmen kann. Weitere Informationen zu diesem Thema finden Sie im Abschnitt »Sequenzbildung für Daten (Marshaling)« auf Seite 47-10.

Die folgende Abbildung zeigt die Schnittstelle *IMyInterface* in einem Objekt, das eine duale Schnittstelle mit der Bezeichnung *IMyInterface* unterstützt. Die ersten drei Einträge der Vtable beziehen sich auf die Schnittstelle *IUnknown*, die folgenden vier auf die Schnittstelle *IDispatch*, und bei den übrigen Einträgen handelt es sich um COM-Einträge für den direkten Zugriff auf Elemente der benutzerdefinierten Schnittstelle.

Abbildung 47.1 Vtable für duale Schnittstelle



Dispatch-Schnittstellen

Automatisierungs-Controller sind Clients, die die COM-Schnittstelle *IDispatch* für den Zugriff auf COM-Server-Objekte verwenden. Der Controller muß das betreffende Objekt zunächst erzeugen, dann die Schnittstelle *IUnknown* des Objekts nach ei-

nem Zeiger auf seine Schnittstelle *IDispatch* abfragen. *IDispatch* verfolgt die Methoden und Eigenschaften intern über einen Dispatch-Bezeichner (dispID). Dabei handelt es sich um eine eindeutige Identifikationsnummer für Schnittstellenelemente. Über *IDispatch* ruft ein Controller die Typinformationen des Objekts für die Dispatch-Schnittstelle ab und bildet die Schnittstellennamen auf spezielle dispIDs ab. Diese stehen zur Laufzeit zur Verfügung und werden von den Controllern durch Aufruf der *IDispatch*-Methode *GetIDsOfNames* abgerufen.

Sobald der Controller die dispID ermittelt hat, kann er die *IDispatch*-Methode *Invoke* aufrufen, um den zugehörigen Quelltext (Eigenschaft oder Methode) ausführen zu lassen, wobei die Parameter für die Eigenschaft oder Methode in einen der *Invoke*-Parameter verpackt werden. Invoke weist bei der Compilierung eine feste Signatur auf, die es diesem Parameter ermöglicht, beim Aufrufen einer Schnittstellenmethode eine beliebige Anzahl von Argumenten zu akzeptieren.

Die Implementierung der Methode *Invoke* für das Automatisierungsobjekt muß dann die Parameter wieder »entpacken«, die Eigenschaft oder Methode aufrufen und auf die Behandlung jeglicher auftretenden Fehler eingerichtet sein. Wenn die Eigenschaft oder Methode zurückgegeben wird, liefert das Objekt den Rückgabewert an den Controller.

Dieser Vorgang wird spätes Binden genannt, weil die Bindung des Controllers an die Eigenschaft bzw. Methode zur Laufzeit und nicht schon bei der Compilierung erfolgt.

Benutzerdefinierte Schnittstellen

Benutzerdefinierte Schnittstellen ermöglichen Clients das Aufrufen von Schnittstellenmethoden anhand ihrer Reihenfolge in der VTable und anhand der Kenntnis der benötigten Argumenttypen. In der VTable sind die Adressen aller Eigenschaften und Methoden aufgeführt, die Elemente des Objekts sind, einschließlich der Elementfunktionen der vom Objekt unterstützten Schnittstellen. Wenn das Objekt *IDispatch* nicht unterstützt, folgen die Einträge für die Elemente der benutzerdefinierten Schnittstelle des Objekts direkt nach den Elementen von *IUnknown*.

Wenn das Objekt eine Typbibliothek besitzt, können Sie über das Layout seiner virtuellen Funktionstabelle (VTable), die mit dem Typbibliothekseditor ermittelt werden kann, auf die benutzerdefinierte Schnittstelle zugreifen. Besitzt das Objekt eine Typbibliothek und unterstützt außerdem *IDispatch*, kann ein Client auch die dispIDs der *IDispatch*-Schnittstelle ermitteln und direkt an einen VTable-Offset binden. Die Importroutine für Typbibliotheken von Delphi (TLIBIMP) ruft dispIDs beim Importieren ab. Wenn Clients die Dispatch-Schnittstellen kapseln, sind keine Aufrufe von *GetIDsOfNames* erforderlich, da sich die Informationen bereits in der _TLB-Datei befindet. Clients müssen jedoch weiterhin Invoke aufrufen.

Sequenzbildung für Daten (Marshaling)

Bei Out-of-Process-Servern und Remote-Servern müssen Sie berücksichtigen, auf welche Weise COM die Sequenzbildung und Übertragung der Daten aus dem aktuellen Prozeß heraus (Marshaling) vornimmt. Sie können diese Funktionalität folgendermaßen zur Verfügung stellen:

- Automatisch durch Verwendung der Schnittstelle *IDispatch*.
- Automatisch durch Erstellen einer Typbibliothek zusammen mit dem Server und Markieren der Schnittstelle mit dem Ole-Automatisierungs-Flag. COM weiß, wie die Sequenzbildung (Marshaling) aller automatisierungskompatiblen Typen in der Typbibliothek erfolgen muß und kann die Proxy-Server und Stubs entsprechend einrichten. Für die automatische Sequenzbildung gelten einige Einschränkungen bezüglich der verwendbaren Typen.
- Manuell durch Implementierung aller Methoden der Schnittstelle *IMarshal*. Diese Sequenzbildung wird benutzerdefinierte Sequenzbildung genannt.

Automatisierungskompatible Typen

Funktionsergebnisse und Parametertypen der in dualen und Dispatch-Schnittstellen deklarierten Methoden müssen automatisierungskompatibel sein. Die folgenden Typen sind OLE-automatisierungskompatibel:

- Die vordefinierten gültigen Typen wie *Smallint*, *Integer*, *Single*, *Double*, *WideString*. Eine vollständige Liste finden Sie unter »Gültige Typen« auf Seite 50-25.
- In einer Typbibliothek definierte Aufzählungstypen. OLE-automatisierungskompatible Aufzählungstypen werden als 32-Bit-Werte gespeichert und bei der Parameterübergabe als Werte vom Typ Integer behandelt.
- In einer Typbibliothek definierte Schnittstellentypen, die OLE-automatisierungssicher sind, d. h. von der Schnittstelle *IDispatch* abgeleitet wurden und nur OLE-automatisierungskompatible Typen enthalten.
- In einer Typbibliothek definierte dispinterface-Typen.
- *IFont*, *IStrings* und *IPicture*. Für die folgenden Zuordnungen müssen Hilfsobjekte instantiiert werden:
  - *IFont* zu *TFont*
  - *IStrings* zu *TStrings*
  - *IPicture* zu *TPicture*

    Der Experte für ActiveX-Steuerelemente und der ActiveForm-Experte erstellen diese Hilfsobjekte bei Bedarf automatisch. Wenn Sie diese Hilfsobjekte verwenden möchten, rufen Sie die betreffende globale Routine auf: *GetOleFont*, *GetOleStrings*, *GetOlePicture*.

Typeinschränkungen bei der automatischen Sequenzbildung

Damit eine Schnittstelle die automatische Sequenzbildung unterstützt, müssen bestimmte Einschränkungen beachtet werden. Wenn Sie ein Automatisierungsobjekt mit dem Typbibliothekseditor bearbeiten, erzwingt dieser die folgenden Einschränkungen:

- Damit eine plattformübergreifende Kommunikation möglich ist, müssen die Typen entsprechend kompatibel sein. So ist es z. B. nicht möglich, Datenstrukturen (außer zum Implementieren eines weiteren Eigenschaftenobjekts), Argumente ohne Vorzeichen, Wide-Strings usw. zu verwenden.
- String-Datentypen müssen als BSTR übertragen werden. Werte vom Typ PChar bzw. AnsiString können nicht sicher übertragen werden.
- Alle Elemente einer dualen Schnittstelle müssen HRESULT als Rückgabewert der Funktion übergeben.
- Elemente einer dualen Schnittstelle, die andere Werte zurückgeben müssen, sollten diese Parameter als var oder out zurückgeben, wodurch ein Ausgabeparameter angegeben wird, der den Funktionswert zurückgibt.

Hinweis Ein Verfahren, um die Typeinschränkungen zu umgehen, besteht darin, eine separate *IDispatch*-Schnittstelle und eine benutzerdefinierte Schnittstelle zu implementieren. Auf diese Weise können Sie die volle Bandbreite möglicher Argumenttypen nutzen. Dies bedeutet, daß die COM-Clients wahlweise die benutzerdefinierte Schnittstelle verwenden können, auf welche die Automatisierungs-Controller dennoch Zugriff haben. In diesem Fall müssen Sie allerdings den Quelltext für die Sequenzbildung (Marshaling) manuell implementieren.

Benutzerdefinierte Sequenzbildung

In der Regel werden Sie für Ihre Out-of-Process-Server und Remote-Server automatische Sequenzbildung verwenden, weil dies einfacher ist; COM erledigt hierbei die Arbeit für Sie. Es kann jedoch Fälle geben, in denen Sie benutzerdefinierte Sequenzbildung zur Verfügung stellen möchten, wenn Sie davon ausgehen können, daß die manuelle Sequenzbildung im betreffenden Fall schneller abläuft.

Kapitel

48

ActiveX-Steuerelemente erstellen

Ein ActiveX-Steuerelement ist eine Softwarekomponente, die auf jedem Host integriert werden kann, der ActiveX-Steuerelemente unterstützt. Dazu gehören beispielsweise C++Builder, Delphi, Visual dBASE, Visual Basic, Internet Explorer und Netscape Navigator.

Delphi wird mit mehreren ActiveX-Steuerelementen ausgeliefert, mit denen sich z. B. Diagramm-, Tabellenkalkulations- und Grafikfähigkeiten implementieren lassen. Sie können diese Steuerelemente in die Komponentenpalette der Entwicklungsumgebung aufnehmen und sie wie alle Standard-VCL-Komponenten einsetzen, sie auf Formularen ablegen und ihre Eigenschaften im Objektinspektor ändern.

ActiveX-Steuerelemente lassen sich außerdem im Web weitergeben. Sie können dann in HTML-Dokumenten verwendet und mit ActiveX-fähigen Web-Browsern angezeigt werden.

Dieses Kapitel bietet einen Überblick über die Erstellung von ActiveX-Steuerelementen in der Delphi-Umgebung. Die im Rahmen der Erstellung von ActiveX-Steuerelementen erforderlichen Einzelheiten zur Implementierung sollen hier nicht beschrieben werden. Diese Informationen sind in der Dokumentation zu Microsoft Developer's Network (MSDN) enthalten. Sie können auch in der Web-Site von Microsoft nach Informationen zu ActiveX suchen.

ActiveX-Steuerelemente erzeugen – Übersicht

Delphi stellt zwei Experten für die ActiveX-Entwicklung bereit:

- Der ActiveX-Steuerelement-Experte ermöglicht Ihnen die Konvertierung eines vorhandenen bzw. eines benutzerdefinierten VCL-Steuerelements in ein ActiveX-Steuerelement durch Kapseln des VCL-Steuerelements mit einer ActiveX-Klasse.
- Mit Hilfe des ActiveForm-Experten können Sie auch von Grund auf neue ActiveX-Anwendungen erstellen. Der Experte richtet hierbei das Projekt ein und fügt ein

leeres Formular hinzu, in das Sie die Steuerelemente zum Entwerfen des Formulars einbauen können.

Der ActiveX-Steuerelement-Experte erzeugt eine Implementierungs-Unit, die einerseits vom Objekt *TActiveXControl* (für die spezifischen Komponenten von ActiveX-Steuerelementen) und andererseits vom VCL-Objekt des Steuerelements, das eingekapselt werden soll, abstammt. Die vom ActiveForm-Experten erzeugte Implementierungs-Unit ist ein Nachkomme von *TActiveForm*.

Führen Sie zum Erstellen eines neuen ActiveX-Steuerelements die folgenden Schritte durch:

1. Entwerfen und erstellen Sie das benutzerdefinierte VCL-Steuerelement, das die Basis für Ihr ActiveX-Steuerelement bilden soll.
2. Erstellen Sie mit Hilfe des ActiveX-Steuerelement-Experten aus dem einen VCL-Steuerelement ein ActiveX-Steuerelement

   oder:

   Erstellen Sie mit Hilfe desActiveForm-Experten auf der Basis eines VCL-Formulars ein ActiveX-Steuerelement für die Weitergabe im Web.
3. Verwenden Sie den ActiveX-Experten für Eigenschaftenseiten, um bei Bedarf eine oder mehrere Eigenschaftenseiten für das Steuerelement zu erstellen.
4. Verknüpfen Sie die Eigenschaftenseite mit dem ActiveX-Steuerelement (optional).
5. Registrieren Sie das Steuerelement.
6. Testen Sie das Steuerelement mit allen potentiellen Zielanwendungen.
7. Nun können Sie das ActiveX-Steuerelement über das Web weitergeben.

Ein ActiveX-Steuerelement besteht aus dem VCL-Steuerelement, ausgehend von dem es erstellt wurde, sowie aus Eigenschaften, Methoden und Ereignissen, die in der Typbibliothek des Steuerelements aufgelistet sind.

Elemente eines ActiveX-Steuerelements

Ein ActiveX-Steuerelement enthält eine Vielzahl von Elementen, von denen jedes eine bestimmte Funktion erfüllt. Bei diesen Elementen handelt es sich um ein VCL-Steuerelement, um Eigenschaften, Methoden und Ereignisse sowie um eine oder mehrere verknüpfte Typbibliotheken.

VCL-Steuerelement

Ein ActiveX-Steuerelement in Delphi ist nichts anderes als ein VCL-Steuerelement, das so angepaßt wurde, daß Anwendungen und Objekte, die ActiveX-Steuerelemente unterstützen, darauf zugreifen können. Zum Erstellen eines ActiveX-Steuerelements müssen Sie zunächst das VCL-Steuerelement, das als Basis für das ActiveX-Steuerelement dienen soll, erstellen oder auswählen.

Hinweis Die in der Liste des Experten zur Verfügung stehenden Steuerelemente sind vom Objekt *TWinControl* abgeleitet. Manche Steuerelemente, wie z. B. EditControl, sind als Nicht-ActiveX-Steuerelemente (NonActiveXControl) registriert und daher in der Liste nicht enthalten.

Typbibliothek

Eine Typenbibliothek enthält die Typdefinitionen für das Steuerelement und wird automatisch vom ActiveX-Steuerelement-Experten erzeugt. Diese Typinformationen, die detaillierter sind als die von der Schnittstelle gelieferten Informationen, stellen für Steuerelemente eine Möglichkeit dar, den Host-Anwendungen mitzuteilen, welche Dienste sie zur Verfügung stellen. Beim Entwerfen eines Steuerelements werden die Typbibliotheksinformationen in einer Datei mit der Erweiterung TLB und in einer entsprechenden Pascal-Datei gespeichert, die die Umsetzungen in Pascal enthält. Beim Erstellungsvorgang für das ActiveX-Steuerelement werden die Typbibliotheksinformationen automatisch compiliert und als Ressource in die DLL des ActiveX-Steuerelements geschrieben.

Eigenschaften, Methoden und Ereignisse

Das ActiveX-Steuerelement übernimmt die Eigenschaften, Methoden und Ereignisse des VCL-Steuerelements.

- Eine Eigenschaft ist ein Attribut, wie z. B. eine Farbe oder ein Label.
- Eine Methode ist eine Anforderung an ein Steuerelement, eine bestimmte Aktion durchzuführen.
- Ein Ereignis ist eine Benachrichtigung von einem Steuerelement an den Container, die diesen darüber informiert, daß ein bestimmter Vorgang erfolgt ist.

Eigenschaftenseite

Die Eigenschaftenseite ermöglicht dem Benutzer eines Steuerelements, dessen Eigenschaften anzuzeigen und zu bearbeiten. Sie können mehrere Eigenschaften auf einer Seite gruppieren oder für eine einzelne Eigenschaft ein Benutzeroberflächenelement in der Art eines Dialogfelds zur Verfügung stellen. Informationen zum Erstellen von Eigenschaftenseiten finden Sie unter »Eine Eigenschaftenseite für ein ActiveX-Steuerelement erstellen« auf Seite 48-16.

Ein ActiveX-Steuerelement entwerfen

Am Anfang des Entwurfsvorgangs für ein ActiveX-Steuerelement steht die Erstellung eines benutzerdefinierten VCL-Steuerelements, ausgehend von dem das neue ActiveX-Steuerelement erstellt werden soll. Informationen hierzu finden Sie in Teil IV, »Benutzerdefinierte Komponenten erzeugen«.

Beim Entwerfen eines ActiveX-Steuerelements entweder ausgehend von einem vorhandenen VCL-Steuerelement oder mit Hilfe eines neuen Active-Formulars (Active-

Form) müssen Sie beachten, daß Sie ein ActiveX-Steuerelement implementieren, das in eine andere Anwendung eingebettet wird und nicht selbst eine Anwendung ist.

Aus diesem Grund sollten Sie keine komplexen Dialogfelder oder andere Hauptkomponenten einer Benutzeroberfläche verwenden. In der Regel geht es darum, ein einfaches Steuerelement zu erstellen, das in der Hauptanwendung nach deren Regeln arbeitet.

Ob Sie Ihr ActiveX-Steuerelement aus einem VCL-Steuerelement oder mit Hilfe eines Active-Formulars (ActiveForm) erstellen sollten, hängt davon ab, ob Sie bereits über ein entsprechendes Steuerelement verfügen, das einfach in einem ActiveX-Steuerelement verkapselt werden soll. In diesem Fall empfiehlt sich die Verwendung des ActiveX-Steuerelement-Experten, wie dies im Abschnitt »ActiveX-Steuerelemente aus VCL-Steuerelementen erstellen« auf Seite 47-5.

Wenn Sie eine vollständige ActiveX-Anwendung erstellen möchten, sollten Sie den ActiveForm-Experten verwenden; siehe hierzu die Beschreibung im Abschnitt »ActiveX-Steuerelemente auf der Basis eines VCL-Formulars erstellen« auf Seite 47-8.. Normalerweise werden mit einem Active-Formular (ActiveForm) ActiveX-Anwendungen erstellt, die dann über das Web weitergegeben werden sollen.

Die Experten implementieren mit Hilfe der VCL-Objekte *TActiveXControl* bzw. *TActiveForm* alle erforderlichen ActiveX-Schnittstellen. Sie brauchen dann nur die zusätzlichen Schnittstellen zu implementieren, die Sie gegebenenfalls in Ihr Steuerelement eingebettet haben.

Sobald das Steuerelement mit Hilfe eines der Experten erzeugt wurde, können Sie dessen Eigenschaften, Methoden und Ereignisse über den Typbibliothekseditor ändern.

ActiveX-Steuerelemente aus VCL-Steuerelementen erstellen

Die Erstellung eines ActiveX-Steuerelements aus einem VCL-Steuerelement erfolgt mit Hilfe des ActiveX-Element-Experten. Alle Eigenschaften, Methoden und Ereignisse des VCL-Steuerelements werden für das ActiveX-Steuerelement übernommen.

Informationen über die Erstellung von VCL-Steuerelementen finden Sie in Teil IV, »Benutzerdefinierte Komponenten entwickeln«.

Der ActiveX-Steuerelement-Experte kapselt das VCL-Steuerelement in einer ActiveX-Klasse und erstellt auf diese Weise ein ActiveX-Steuerelement, das dieses Objekt enthält. Die Funktionalität des VCL-Steuerelements wird über diese ActiveX-Kapselung anderen Objekten und Servern zur Verfügung gestellt.

Bevor Sie den ActiveX-Steuerelement-Experten verwenden können, müssen Sie das VCL-Steuerelement auswählen, ausgehend von dem das ActiveX-Steuerelement erstellt werden soll.

So rufen Sie den ActiveX-Steuerelement-Experten auf:

1. Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.
2. Wählen Sie die Registerkarte *ActiveX aus.*

3 Doppelklicken Sie auf das Symbol *ActiveX-Element.*

Geben Sie im Experten folgendes an:

VCL-Klassenname	Wählen Sie das VCL-Steuerelement in der Dropdown-Liste aus. Um beispielsweise ein ActiveX-Steuerelement zu erzeugen, das Client-Anwendungen die Verwendung des TButton-Objekts ermöglicht, wählen Sie *TButton* aus. Für ActiveForms ist die Option VCL-Klassenname nicht verfügbar, da Active-Formulare immer auf *TactiveForm* basieren.
Neuer ActiveX-Name	Der Experte stellt einen Standardnamen bereit, der von Clients zur Identifizierung Ihres ActiveX-Steuerelements verwendet wird. Ändern Sie diesen Namen, um einen anderen OLE-Klassennamen bereitzustellen.
Implementierungs-Unit	Der Experte bietet einen Standardnamen für die Unit an, die den Quelltext zur Implementierung des Verhaltens Ihres ActiveX-Steuerelements enthält. Sie können entweder diesen Standardnamen akzeptieren oder einen neuen eingeben.
Projekt-Name	Der Experte bietet einen Standardnamen für das ActiveX-Bibliotheksprojekt für Ihr ActiveX-Steuerelement an, sofern aktuell kein Projekt geöffnet ist. Wenn bereits eine ActiveX-Bibliothek geöffnet wurde, ist diese Option deaktiviert.
Threading-Modell	Wählen Sie das gewünschte Threading-Modell , um anzugeben, wie Client-Anwendungen die Schnittstelle Ihres Steuerelements aufrufen können. Hierbei legen Sie das Threading-Modell für die Implementierung im Steuerelement fest. Weitere Informationen zu Threading-Modellen finden Sie im Ab schnitt »Ein Threading-Modell auswählen« auf Seite 45-5. **Hinweis**: Über die Wahl des Threading-Modells legen Sie fest, wie das Objekt registriert wird. Beachten Sie, daß Ihre Objektimplementierung mit dem gewählten Modell kompati bel ist.

Legen Sie die folgenden ActiveX-Steuerelementoptionen fest:

Element lizensieren	Markieren Sie dieses Kontrollkästchen, um die Lizensierung für die ActiveX-Steuerelemente zu aktivieren (siehe Lizensierung aktivieren), die Sie entwickeln und weitergeben (sofern die Weitergabe des Steuerelements nicht gebührenfrei erfolgen soll). Der Experte generiert dann eine Entwurfszeitlizenz für die Entwickler des Steuerelements sowie eine Laufzeitlizenz für die Benutzer des Steuerelements.
Versionsinformationen hinzufügen	Markieren Sie dieses Kontrollkästchen, um Versionsinformationen wie das Copyright oder eine Dateibeschreibung in das ActiveX-Steuerelement einzufügen. Diese Informationen können in einem Browser angezeigt werden. Wählen Sie *Projekt / Optionen* und dann die Registerkarte *Versionsinfo*, um die Versionsinformationen einzugeben. Klicken Sie gegebenenfalls auf die Schaltfläche *Hilfe*, um Informationen zu den Einstellungen anzuzeigen. **Hinweis**: Versionsinformationen werden für die Registrierung des Steuerelements in Visual Basic 4.0 benötigt.
Info-Fenster hinzufügen	Wenn diese Option aktiviert ist, enthält das Projekt ein Info-Fenster. Der Benutzer des Steuerelements kann das Info-Fenster in einer Entwicklungsumgebung anzeigen. Bei dem Fenster handelt es sich um ein separates Formular, das Sie ändern können. Das Info-Fenster enthält per Voreinstellung den Namen des ActiveX-Steuerelements, ein Bild, Copyright-Informationen und eine OK-Schaltfläche.

Der Experte erzeugt die folgenden Dateien:

- Ein ActiveX-Bibliotheksprojekt mit dem Quelltext, der für den Start von ActiveX-Steuerelementen benötigt wird. Diese Datei braucht normalerweise nicht geändert zu werden.
- Eine Typbibliothek (mit der Dateinamenserweiterung TLB), in der die Eigenschaften, Methoden und Ereignisse definiert und implementiert sind, die das ActiveX-Steuerelement für die Automatisierung bereitstellt. Einzelheiten zu Typbibliotheken finden Sie in Kapitel 50, »Mit Typbibliotheken arbeiten«.
- Eine ActiveX-Implementierungs-Unit, die das ActiveX-Steuerelement mit Hilfe des Delphi ActiveX-Frameworks (DAX) definiert und implementiert. DAX stellt die Delphi-Implementierung der COM-Spezifikation für ActiveX-Steuerelemente dar. Sie können die Implementierung entsprechend Ihren Erfordernissen ändern.
- Ein Formular und eine Unit für ein Info-Fenster (wenn Sie die Option *Info-Fenster hinzufügen* aktiviert haben).

ActiveX-Steuerelemente lizensieren

Das Lizensieren eines ActiveX-Steuerelements erfolgt durch Bereitstellen eines Lizenzschlüssels zur Entwurfszeit und der Unterstützung zur dynamischen Generierung von Lizenzen, wenn das Steuerelement zur Laufzeit erstellt wird.

Zur Bereitstellung von Entwurfszeitlizenzen erstellt der ActiveX-Experte einen Schlüssel für das Steuerelement, der in einer Datei mit dem Namen des Projekts und der Dateinamenerweiterung LIC gespeichert wird. Der Benutzer des Steuerelements muß eine Kopie der LIC-Datei besitzen, um das Steuerelement in einer Entwicklungsumgebung öffnen zu können. Jedes Steuerelement im Projekt, für das die Option *Element lizensieren* aktiviert ist, besitzt einen eigenen Schlüsseleintrag in der LIC-Datei.

Zur Unterstützung der Erstellung von Laufzeitlizenzen wird die Lizenz durch Abfragen des Steuerelements (zur Entwurfszeit) generiert, dann gespeichert und später an das Steuerelement übergeben, wenn dieses im Kontext einer ausführbaren Datei (EXE) erstellt wird. Sobald die Laufzeitlizenz an das Steuerelement übergeben wurde, wird die Entwurfszeitlizenz nicht mehr benötigt.

Laufzeitlizenzen für Internet Explorer setzen einen weiteren Umweg voraus, da Benutzer den HTML-Quelltext jeder Web-Seite anzeigen können und das ActiveX-Steuerelement vor dem Anzeigen auf den Computer des Benutzers kopiert wird. Wenn Sie also Lizenzen für Steuerelemente erstellen, die im Internet Explorer genutzt werden, müssen Sie zunächst eine Lizenzpaketdatei (LPK-Datei) generieren und diese in die HTML-Seite mit dem Steuerelement einbetten.

Die LPK-Datei besteht aus einem Array mit CLSIDs und Lizenzschlüsseln für ActiveX-Steuerelemente.

Hinweis Die LPK-Datei wird mit dem Dienstprogramm LPK_TOOL.EXE generiert, das Sie aus der Web-Site von Microsoft (www.microsoft.com) herunterladen können.

Verwenden Sie die HTML-Objekte <OBJECT> und <PARAM> folgendermaßen, um die LPK-Datei in eine Web-Seite einzubetten:

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
    <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

Die CLSID identifiziert das Objekt als Lizenzpaket, während PARAM die relative Position der Lizenzpaketdatei im Verhältnis zur HTML-Seite angibt.

Sobald die Web-Seite mit dem Steuerelement in Internet Explorer angezeigt werden soll, wird die LPK-Datei untersucht, der Lizenzschlüssel extrahiert und - sofern der Schlüssel der Lizenz des Steuerelements entspricht - das Steuerelement auf der Seite angezeigt. Enthält eine Web-Seite mehrere LPKs, berücksichtigt Internet Explorer nur die erste.

Weitere Informationen finden Sie in der Web-Site von Microsoft unter dem Stichwort »ActiveX-Steuerelemente lizensieren«.

ActiveX-Steuerelemente auf der Basis eines VCL-Formulars erstellen

Der ActiveForm-Experte erzeugt ein ActiveX-Steuerelement auf der Basis eines VCL-Formulars, das Sie entwerfen können, nachdem der Experte den Formular-Designer aufgerufen hat. Mit Hilfe eines Active-Formulars (ActiveForm) lassen sich Anwendungen erstellen, die anschließend über das Web weitergegeben werden können.

Bei der Weitergabe eines Active-Formulars (ActiveForm) wird eine HTML-Seite erstellt, die einen Verweis auf das Active-Formular enthält und dessen Position auf der Seite angibt. Das Active-Formular wird dann in einem Web-Browser angezeigt und ausgeführt. Innerhalb des Web-Browsers verhält sich das Formular genauso wie ein eigenständiges Formular von Delphi. Das Formular kann VCL- oder ActiveX-Komponenten, einschließlich benutzerdefinierter VCL-Steuerelemente, enthalten.

So starten Sie den ActiveForm-Experten:

1. Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.
2. Wählen Sie die Registerkarte *ActiveX aus.*
3. Doppelklicken Sie auf das Symbol *ActiveForm.*

Geben Sie im Experten folgendes an:

VCL-Klassenname	Die Option *VCL-Klassenname* ist nicht verfügbar, da Active-Formulare immer auf *TactiveForm* basieren.
Neuer ActiveX-Name	Der Experte stellt einen Standardnamen bereit, der von Clients zur Identifizierung Ihres ActiveX-Steuerelements verwendet wird. Ändern Sie diesen Namen, um einen anderen OLE-Klassennamen bereitzustellen.
Implementierungs-Unit	Der Experte bietet einen Standardnamen für die Unit an, die den Quelltext zur Implementierung des Verhaltens Ihres ActiveX-Steuerelements enthält. Sie können entweder diesen Standardnamen akzeptieren oder einen neuen Namen eingeben.
Projekt-Name	Der Experte bietet einen Standardnamen für das ActiveX-Bibliotheksprojekt für Ihr ActiveX-Steuerelement an, sofern aktuell kein Projekt geöffnet ist. Wenn bereits eine ActiveX-Bibliothek geöffnet wurde, ist diese Option deaktiviert.
Threading-Modell	Wählen Sie das gewünschte Threading-Modell , um anzugeben, wie Client-Anwendungen die Schnittstelle Ihres Steuerelements aufrufen können. Hierbei legen Sie das Threading-Modell für die Implementierung im Steuerelement fest. Weitere Informationen zu Threading-Modellen finden Sie im Abschnitt »Ein Threading-Modell auswählen« auf Seite 45-5. **Hinweis**: Über die Wahl des Threading-Modells legen Sie fest, wie das Objekt registriert wird. Beachten Sie, daß Ihre Objektimplementierung mit dem gewählten Modell kompatibel ist.

Legen Sie die folgenden ActiveX-Steuerelementoptionen fest:

Element lizensieren	Markieren Sie dieses Kontrollkästchen, um die Lizensierung für die ActiveX-Steuerelemente zu aktivieren (siehe Lizensierung aktivieren), die Sie entwickeln und weitergeben (sofern die Weitergabe des Steuerelements nicht gebührenfrei erfolgen soll). Der Experte generiert dann eine Entwurfszeitlizenz für die Entwickler sowie eine Laufzeitlizenz für die Benutzer des Steuerelements.
Versionsinformationen hinzufügen	Markieren Sie dieses Kontrollkästchen, um Versionsinformationen wie das Copyright oder eine Dateibeschreibung in das ActiveX-Steuerelement einzufügen. Diese Informationen können in einem Browser angezeigt werden. Wählen Sie *Projekt / Optionen* und dann die Registerkarte *Versionsinfo*, um die Versionsinformationen einzugeben. Klicken Sie gegebenenfalls auf die Schaltfläche *Hilfe*, um Informationen zu den Einstellungen anzuzeigen. **Hinweis**: Versionsinformationen werden für die Registrierung des Steuerelements in Visual Basic 4.0 benötigt.
Info-Fenster hinzufügen	Wenn diese Option aktiviert ist, enthält das Projekt ein Info-Fenster. Der Benutzer des Steuerelements kann das Info-Fenster in einer Entwicklungsumgebung anzeigen. Bei dem Fenster handelt es sich um ein separates Formular, das Sie ändern können. Das Info-Fenster enthält per Voreinstellung den Namen des ActiveX-Steuerelements, ein Bild, Copyright-Informationen und eine *OK*-Schaltfläche.

Der Experte erzeugt die folgenden Dateien:

- Ein ActiveX-Bibliotheksprojekt mit dem Quelltext, der für den Start von ActiveX-Steuerelementen benötigt wird. Diese Datei braucht normalerweise nicht geändert zu werden.
- Ein Formular.
- Eine Typbibliothek (mit der Dateinamenserweiterung TLB), in der die Eigenschaften, Methoden und Ereignisse definiert und implementiert sind, die das ActiveX-Steuerelement für die Automatisierung bereitstellt. Einzelheiten zu Typbibliotheken finden Sie in Kapitel 50, »Mit Typbibliotheken arbeiten«.
- Eine ActiveX-Implementierungs-Unit, die die Eigenschaften, Methoden und Ereignisse des Objekts *TActiveForm* mit Hilfe des Delphi ActiveX-Frameworks (DAX) definiert und implementiert. DAX stellt die Delphi-Implementierung der COM-Spezifikation für ActiveX-Steuerelemente dar. Sie können die Implementierung entsprechend Ihren Erfordernissen ändern.
- Ein Formular und eine Unit für ein Info-Fenster (wenn Sie die Option *Info-Fenster hinzufügen* aktiviert haben).

An dieser Stelle fügt der Experte ein leeres Formular zu Ihrem ActiveX-Bibliotheksprojekt hinzu. Nun können Sie Steuerelemente hinzufügen und das Formular nach Ihren Wünschen gestalten.

Nachdem Sie das ActiveForm-Projekt entworfen und compiliert haben, wobei eine ActiveX-Bibliothek (mit der Erweiterung OCX) erzeugt wurde, können Sie das Projekt an Ihren Web-Server weitergeben. Delphi erzeugt dann eine HTML-Testseite mit einem Verweis auf das Active-Formular (ActiveForm).

Eigenschaften, Methoden und Ereignisse von ActiveX-Steuerelementen

Der ActiveX-Element-und der ActiveForm-Experte legen eine Typbibliothek an, in der die Eigenschaften, Methoden und Ereignisse des ursprünglichen VCL-Steuerelements bzw. VCL-Formulars definiert und implementiert sind. Dies gilt allerdings nicht für folgende Elemente:

- Eigenschaften, Methoden und Ereignisse, die einen Nicht-OLE-Typ verwenden
- Datensensitive Eigenschaften

Eventuell müssen deshalb bestimmte Eigenschaften, Methoden und Ereignisse manuell hinzugefügt werden.

Hinweis Da ActiveX-Steuerelemente über einen anderen Mechanismus als VCL-Steuerelemente verfügen, wenn es darum geht, die Datensensitivität des Steuerelements festzulegen, führen die Experten keine Konvertierung der datenbezogenen Eigenschaften durch. Sie können Ihr Steuerelement mit einigen der ActiveX-Eigenschaften für Datensensitivität versehen, wie im Abschnitt »Einfache Datenbindung mit der Typbibliothek ermöglichen« auf Seite 48-15 beschrieben ist.

Sie können Eigenschaften, Methoden und Ereignisse in einem ActiveX-Steuerelement durch Bearbeitung der Typbibliothek hinzufügen, ändern und entfernen. Hierbei haben Sie mehrere Möglichkeiten:

- Wählen Sie *Bearbeiten / Zur Schnittstelle hinzufügen* in der Entwicklungsumgebung. Einzelheiten hierzu finden Sie im Abschnitt »Weitere Eigenschaften, Methoden und Ereignisse hinzufügen« auf Seite 48-12.
- Verwenden Sie den Typbibliothekseditor, wie in Kapitel 50, »Mit Typbibliotheken arbeiten«beschrieben.

Hinweis Die in der Typbibliothek vorgenommen Änderungen gehen verloren, wenn Sie das ActiveX-Steuerelement aus dem ursprünglichen VCL-Steuerelement bzw. -Formular neu generieren.

Hinweis Eigenschaften, die nicht als published deklariert und die von Hand in die Typbibliothek eingefügt wurden, erscheinen zwar in der Entwicklungsumgebung, doch sind die daran vorgenommenen Änderungen nicht von Dauer. Das bedeutet: Wenn der Benutzer den Wert einer Eigenschaft ändert, wird dies bei der Ausführung des betreffenden Steuerelements ignoriert. Wenn bei einem aus einem VCL-Objekt abgeleiteten Steuerelement eine Eigenschaft noch nicht als **published** deklariert wurde, müssen

Sie zuerst einen Nachkommen des VCL-Objekts erstellen und dann die betreffende Eigenschaft dieses Nachkommens als published deklarieren.

Weitere Eigenschaften, Methoden und Ereignisse hinzufügen

Sie können dem Steuerelement folgendermaßen Eigenschaften, Methoden und Ereignisse hinzufügen:

1 Wählen Sie *Bearbeiten / Zur Schnittstelle hinzufügen*. Die Implementierungs-Unit für das ActiveX-Steuerelement muß geöffnet und ausgewählt sein, damit der Menübefehl zur Verfügung steht.

   Damit wird das Dialogfeld *Zur Schnittstelle hinzufügen* aufgerufen.

2 Wählen Sie in der Dropdown-Liste für die Schnittstelle einen der Einträge *Methode* oder *Ereignis* aus. Neben den Eigenschaften, Methoden oder Ereignissen steht der Name der Schnittstelle, zu dem das Element hinzugefügt wird.

3 Geben Sie die Deklaration für die Eigenschaft, die Methode oder das Ereignis ein. Wenn Sie beispielsweise ein ActiveX-Steuerelement aus dem VCL-Steuerelement *TButton* erstellen, können Sie die folgende Eigenschaft hinzufügen:

```
property Top:integer;
```

   Wenn Sie das Kontrollfeld *Syntaxhilfe* aktivieren, werden während der Eingabe Popup-Fenster eingeblendet, die Ihnen mitteilen, was Sie an der jeweiligen Stelle eingeben müssen.

4 Wählen Sie *OK*.

   Die Deklaration wird automatisch zur Implementierungs-Unit des Steuerelements, zur Typbibliotheksdatei (TLB) und zur Typbibliotheks-Unit hinzugefügt. Welche spezifische Deklaration Delphi hinzufügt, hängt davon ab, ob Sie eine Eigenschaft, eine Methode oder ein Ereignis hinzugefügt haben.

So fügt Delphi Eigenschaften hinzu

Da es sich bei der Schnittstelle um eine duale Schnittstelle handelt, werden die Eigenschaften über die Zugriffsmethoden *Read* und *Write* (Lese- und den Schreibzugriff) in der Pascal-Datei (mit der Erweiterung PAS) implementiert. Wenn Sie eine *Caption*-Eigenschaft spezifizieren, fügt der Experte die folgende Deklaration zur Implementierungs-Unit hinzu:

```
property Caption: Integer read Get_Caption write Set_Caption;
```

Die Deklarationen und Implementierungen der Zugriffsmethoden *Read* und *Write* für die Eigenschaft lauten dann folgendermaßen:

```
function Get_Caption: Integer; safecall;
procedure Set_Caption(Value: Integer); safecall;
function TButtonX.Get_Caption: Integer;
begin
  Result := FDelphiControl.Caption;
end;
procedure TButtonX.Set_Caption(Value: Integer);
```

```
begin
  FDelphiControl.Caption := Value;
end;
```

Hinweis Da die Automatisierungsschnittstellenmethoden als safecall deklariert sind, brauchen Sie keinen COM-Exception-Quelltext für diese Methoden zu implementieren; der Delphi-Compiler erledigt dies für Sie, indem er um den Hauptteil der safecall–Methoden Quelltext generiert, über den Delphi-Exceptions abgefangen und in COM-Fehlerinformationsstrukturen und Rückgabecodes konvertiert werden.

Die Methoden der Schnittstellenimplementierung geben das Verhalten einfach an das VCL-Steuerelement weiter. In manchen Fällen müssen Sie Quelltext hinzufügen, um die COM-Datentypen in native Delphi-Typen zu konvertieren.

So fügt Delphi Methoden hinzu

Wenn Sie eine Methode hinzufügen, wird eine leere Implementierung erstellt, die Sie dann mit der gewünschten Funktionalität versehen können. Beispiel: Sie geben folgendes ein:

```
procedure Move;
```

Daraufhin werden die folgende Deklaration und der folgende Quelltext zur Unit hinzugefügt:

```
procedure Move; safecall;
procedure TButtonX.Move;
begin
end;
```

Hinweis Da die Automatisierungsschnittstellenmethoden als safecall deklariert sind, brauchen sie keinen COM-Exception-Quelltext für diese Methoden zu implementieren; der Delphi-Compiler erledigt dies für Sie, indem er um den Hauptteil der safecall–Methoden Quelltext generiert, über den Delphi-Exceptions abgefangen und in COM-Fehlerinformationsstrukturen und Rückgabecodes konvertiert werden.

So fügt Delphi Ereignisse hinzu

Ein ActiveX-Steuerelement kann Ereignisse an seinen Container weitergeben. Durch das Hinzufügen von Ereignissen legen Sie fest, welche Ereignisse durch Steuerelement ausgelöst werden können. Für jedes mit dem Steuerelement verknüpfte Ereignis werden die folgenden Elemente erzeugt:

- Eine Dispatch-Schnittstellendeklaration und Implementierungseinträge in der Implementierungs-Unit.

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);

procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
```

```
end;
```

- Ein Typbibliothekseintrag für die Ereignisschnittstelle, wobei auf der Attributeseite *dispinterface* gewählt ist.
- Eine Object Pascal-Version der Schnittstellenquelldatei der Typbibliothek.

Im Gegensatz zur Automatisierungsschnittstelle wird die Ereignisschnittstelle nicht zur Schnittstellenliste des Steuerelements hinzugefügt. Nicht das Steuerelement empfängt diese Ereignisse, sondern der Container.

Einfache Datenbindung mit der Typbibliothek ermöglichen

Durch einfache Datenbindung können Sie eine Eigenschaft Ihres ActiveX-Steuerelements mit einem speziellen Feld in einer Datenbank verbinden. Hierzu müssen Sie mit Hilfe des Typbibliothekseditors die Binde-Flags der Eigenschaft entsprechend setzen.

Im folgenden wird die Bindung datensensitiver Eigenschaften in einem ActiveX-Steuerelement beschrieben. Informationen über die Bindung datensensitiver Eigenschaften in einem Delphi-Container finden Sie unter »Einfache Datenbindung von ActiveX-Steuerelementen im Delphi-Container ermöglichen« auf Seite 48-16.

Wenn Sie eine Eigenschaft als *bindable* markieren, benachrichtigt das Steuerelement die Datenbank bei einer Änderung einer Eigenschaft (z. B. eines Feldes in einer Datenbank) durch den Benutzer darüber, daß der Wert geändert wurde; gleichzeitig fordert es die Aktualisierung des Datensatzes an. Die Datenbank benachrichtigt daraufhin das Steuerelement, ob die Aktualisierung des Datensatzes erfolgreich durchgeführt werden konnte oder gescheitert ist.

Mit Hilfe der Typbibliothek können Sie die einfache Datenbindung aktivieren. Gehen Sie folgendermaßen vor:

1. Klicken Sie auf der Werkzeugleiste die zu bindende Eigenschaft an.
2. Wählen Sie die *Flags*-Seite.
3. Wählen Sie die folgenden Bindeattribute aus:

Bindeattribut	Bedeutung
Bindable	Gibt an, daß die Eigenschaft Datenbindung unterstützt. Wenn eine Eigenschaft als *bindable* markiert ist, teilt sie ihrem Container eine Änderung ihres Wertes mit.
Request Edit	Gibt an, daß die Eigenschaft die *OnRequestEdit*-Benachrichtigung unterstützt. Dies ermöglicht es dem Steuerelement, den Container zu fragen, ob der Wert der Eigenschaft durch den Benutzer geändert werden darf oder nicht.
Display Bindable	Gibt an, daß der Container dem Benutzer anzeigen kann, daß die Eigenschaft das Attribut *bindable* aufweist.

Bindeattribut	Bedeutung
Default Bindable	Gibt an, daß die Eigenschaft mit dem einzigen Attribut *bindable* am besten zur Darstellung des Objekts geeignet ist. Eigenschaften, für die das Attribut *Default Bindable* gesetzt ist, müssen auch das Attribut *bindable* aufweisen. *Default Bindable* kann in einer Dispatch-Schnittstelle nur für eine Eigenschaft angegeben werden.
Immediate Bindable	Ermöglicht einzelnen Eigenschaften in einem Formular, die das Attribut *bindable* aufweisen, dieses Verhalten anzugeben. Wenn dieses Attribut gesetzt ist, erfolgen bei allen Änderungen Benachrichtigungen. Damit dieses neue Attribut wirksam wird, müssen die Attribute *bindable* und *request edit* gesetzt sein.

4 Klicken Sie zum Aktualisieren der Typbibliothek die Schaltfläche *Aktual.* auf der Werkzeugleiste an.

Bevor Sie eine Schaltfläche mit Datenbindung testen können, müssen Sie sie registrieren.

Um beispielsweise aus einem *TEdit*-Steuerelement ein ActiveX-Steuerelement mit Datenbindung zu machen, leiten Sie zunächst das ActiveX-Steuerelement aus dem *TEdit*-Element ab und setzen dann die Flags der Eigenschaft *Text* auf *Bindable, Display Bindable, Default Bindable* und *Immediate Bindable*. Nachdem das Steuerelement registriert und importiert wurde, kann es zur Datenanzeige verwendet werden.

Einfache Datenbindung von ActiveX-Steuerelementen im Delphi-Container ermöglichen

Um eine Optionsliste für datensensitive ActiveX-Steuerelemente anzuzeigen, klicken Sie mit der rechten Maustaste ein ActiveX-Steuerelement an, nachdem Sie es auf der Seite *ActiveX* der Komponentenpalette installiert und in einem Formular plaziert haben. Zusätzlich zu den Standardoptionen des lokalen Menüs für Formulare wird die Option *Datenbindung* angezeigt.

Hinweis: Setzen Sie die Eigenschaft *DataSource* auf eine *DataSource*-Komponente im Formular, bevor Sie den Editor für ActiveX-Datenbindung aufrufen. Im Editor werden dann die Felder *Feldname* und *Eigenschaft* der *DataSource*-Komponente angezeigt. Der Editor enthält nur diejenigen Eigenschaften der *DataSource*-Komponente, die für die Datenbindung des ActiveX-Steuerelements in Frage kommen.

So binden Sie ein Feld an eine Eigenschaft:

1 Markieren Sie im Dialogfeld *Editor* für ActiveX-Datenbindung den Namen eines Feldes und einer Eigenschaft.

   Unter *Feldname* sind die Felder der Datenbank, unter *Name der Eigenschaft* sind die Eigenschaften des ActiveX-Steuerelements aufgeführt, die an ein Datenbankfeld gebunden werden können. Die DispID der Eigenschaft ist eingeklammert, z. B. Value(12).

2 Klicken Sie auf *Bindung* und anschließend auf *OK*.

Hinweis: Wenn im Dialogfeld keine Eigenschaften angezeigt werden, verfügt das ActiveX-Steuerelement über keine datensensitiven Eigenschaften. Um eine einfache Datenbindung für eine Eigenschaft eines ActiveX-Steuerelements einzurichten, verwenden Sie die Typbibliothek, wie beschrieben im Abschnitt »Einfache Datenbindung mit der Typbibliothek ermöglichen« auf Seite 48-15.

Im folgenden Beispiel werden die Schritte erläutert, die Sie für den Einsatz eines datensensitiven ActiveX-Steuerelements im Delphi-Container ausführen müssen. Dabei wird das Steuerelement *Calendar* von Microsoft benutzt. Sie haben darauf Zugriff, wenn auf Ihrem System Microsoft Office 97 installiert ist.

1. Wählen Sie im Hauptmenü von Delphi *Komponente / ActiveX importieren.*
2. Markieren Sie ein datensensitive ActiveX-Steuerelement, etwa Microsoft Calendar 8.0. Ändern Sie dann dessen Klassennamen in *TCalendarAXControl* um, und klikken Sie auf *Installieren*.
3. Klicken Sie im Dialogfeld *Installieren* auf OK, um das Steuerelement zum Standard-Package hinzuzufügen. Damit steht es nun in der Palette zur Verfügung.
4. Wählen Sie *Datei / Alle schließen* und *Datei / Neue Anwendung,* um eine neue Anwendung zu beginnen
5. Fügen Sie in das Formular aus der Seite *ActiveX* der Komponentenpalette das soeben zur Palette hinzugefügte *TCalendarAXControl*-Objekt ein.
6. Fügen Sie in das Formular aus der Seite *Datenzugriff* der Komponentenpalette ein *DataSource*- und ein *Table*-Objekt ein.
7. Markieren Sie das *DataSource*-Objekt, und setzen Sie seine Eigenschaft *DataSet* auf *Table1*.
8. Markieren Sie das Table-Objekt, und führen Sie folgende Schritte aus:
   - Setzen Sie die Eigenschaft *DataBaseName* auf DBDEMOS.
   - Setzen Sie die Eigenschaft *TableName* auf EMPLOYEE.DB.
   - Setzen Sie die Eigenschaft *Active* auf *True.*
9. Markieren Sie das *TCalendarAXControl*-Objekt, und setzen Sie seine Eigenschaft *DataSource* auf *DataSource1*.
10. Markieren Sie das *TCalendarAXControl*-Objekt, klicken Sie mit der rechten Maustaste, und wählen Sie *Datenbindungen*. Dadurch wird der Editor für ActiveX-Datenbindungen aufgerufen.

    Unter *Feldname* sind alle Felder der aktiven Datenbank aufgeführt. Unter Name der Eigenschaft sind diejenigen Eigenschaften des ActiveX-Steuerelements aufgeführt, die an ein Datenbankfeld gebunden werden können. Die DispID der Eigenschaft ist eingeklammert.
11. Markieren Sie das Feld *HireDate,* und wählen Sie dann *Binden* und *OK*.

    Damit sind nun der Feldname und die Eigenschaft aneinander gebunden.
12. Fügen Sie aus der Seite *Datensteuerung* ein *DBGrid*-Objekt in das Formular ein, und setzen Sie seine Eigenschaft *DataSource* auf *DataSource1*.

13 Fügen Sie aus der Seite *Datensteuerung* ein *DBNavigator*-Objekt in das Formular ein, und setzen Sie seine Eigenschaft *DataSource* auf *DataSource1*.

14 Führen Sie die Anwendung aus.

15 Testen Sie die Anwendung folgendermaßen:

Zeigen Sie im *DBGrid*-Objekt das Feld *HireDate* an, und nehmen Sie mit Hilfe des *DBNavigator*-Objekts in der Datenbank beliebige Positionsänderungen vor. Sie werden feststellen, daß sich die im ActiveX-Steuerelement angezeigten Daten entsprechend ändern.

Eine Eigenschaftenseite für ein ActiveX-Steuerelement erstellen

Eine Eigenschaftenseite ist ein Dialogfeld, das dem Objektinspektor von Delphi ähnelt. Die Benutzer können in der Eigenschaftenseite die Eigenschaften eines ActiveX-Steuerelements ändern. In einem Eigenschaftenseiten-Dialogfeld können Sie zahlreiche Eigenschaften für ein Steuerelement gruppieren, so daß diese gemeinsam geändert werden können. Ferner können Sie ein Dialogfeld für komplexere Eigenschaften vorsehen.

In der Regel wird die Eigenschaftenseite angezeigt, wenn der Benutzer mit der rechten Maustaste auf das Steuerelement klickt und *Eigenschaften* wählt.

Eine Eigenschaftenseite wird ähnlich einem Formular erstellt:

1 Legen Sie eine neue Eigenschaftenseite an.

2 Fügen Sie der Eigenschaftenseite Steuerelemente hinzu.

3 Weisen Sie die Steuerelemente auf der Eigenschaftenseite den Eigenschaften eines ActiveX-Steuerelements zu.

4 Verknüpfen Sie die Eigenschaftenseite mit dem ActiveX-Steuerelement.

Hinweis Beachten Sie, daß beim Hinzufügen von Eigenschaften zu einem ActiveX-Steuerelement oder einem Active-Formular (ActiveForm) die Eigenschaften als published deklariert werden müssen, wenn sie persistent sein sollen. Einzelheiten hierzu finden Sie im Abschnitt »Eigenschaften eines ActiveX-Steuerelements als published deklarieren« auf Seite 47-24.

Eine neue Eigenschaftenseite erstellen

Verwenden Sie zur Erstellung einer neuen Eigenschaftenseite den Eigenschaftenseiten-Experten.

So erstellen Sie eine neue Eigenschaftenseite:

1 Wählen Sie *Datei / Neu*.

2 Wählen Sie die Registerkarte *ActiveX*.

3 Doppelklicken Sie auf das Symbol *Eigenschaftenseite*.

Der Experte erzeugt ein neues Formular und eine Implementierungs-Unit für die Eigenschaftenseite.

Steuerelemente zu einer Eigenschaftenseite hinzufügen

Für jede Eigenschaft des ActiveX-Steuerelements, auf das die Benutzer Zugriff haben sollen, muß der Eigenschaftenseite ein entsprechendes Steuerelement hinzugefügt werden.

Die folgende Abbildung zeigt eine Eigenschaftenseite zum Festlegen der Eigenschaft *MaskEdit* eines ActiveX-Steuerelements.

Abbildung 48.1 Die Eigenschaftenseite für MaskEdit im Entwurfsmodus

Mit Hilfe des Listenfeldes kann der Benutzer eine Beispielmaske auswählen. Mit den Eingabesteuerelementen kann er die Maske testen, bevor er sie dem ActiveX-Steuerelement zuweist.

Steuerelemente auf Eigenschaftenseiten mit Eigenschaften von ActiveX-Steuerelementen verbinden

Nachdem Sie die benötigten Steuerelemente zur Eigenschaftenseite hinzugefügt haben, müssen Sie sie mit den entsprechenden Eigenschaften verbinden. Der dazu erforderliche Quelltext (insbesondere für die Methoden *UpdatePropertyPage* und *UpdateObject*) wird in die Implementierungs-Unit der Eigenschaftenseite eingefügt.

Die Eigenschaftenseite aktualisieren

Fügen Sie zur Methode *UpdatePropertyPage* Quelltext hinzu, über den das Steuerelement für die Eigenschaft auf der Eigenschaftenseite aktualisiert wird, wenn sich die Eigenschaften des ActiveX-Steuerelements ändern. Sie müssen für diese Methode Quelltext erstellen, der die Eigenschaftenseite mit den aktuellen Werten der Eigenschaften des ActiveX-Steuerelements aktualisiert.

Die folgende Anweisung aktualisiert das Eingabefeld (*InputMask*) der Eigenschaftenseite mit dem aktuellen Wert der Eigenschaft *EditMask* des ActiveX-Steuerelements (OleObjects):

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
      { Steuerelemente aus OleObjects aktualisieren }
```

```
      InputMask.Text := OleObject.EditMask;
end;
```

Das Objekt aktualisieren

Fügen Sie zur Methode *UpdateObject* Quelltext hinzu, damit die Eigenschaft aktualisiert wird, wenn der Benutzer die Steuerelemente auf der Eigenschaftenseite ändert. Sie müssen für diese Methode Quelltext bereitstellen, der den Eigenschaften des ActiveX-Steuerelements die neuen Werte zuweist.

Der folgende Quelltext belegt die Eigenschaft *EditMask* eines ActiveX-Steuerelements (OleObjects) mit dem Wert im Eingabefeld (*InputMask*) der Eigenschaftenseite:

Hinweis Zu den Include-Dateien muß die TLB-Datei gehören, in der die Schnittstelle *ICoClassNameDisp* deklariert ist.

```
procedure TPropertyPage1.UpdateObject;
begin
      {OleObjects mit Werten aus Steuerelementen aktualisieren. }
      OleObject.EditMask := InputMask.Text;
end;
```

Eine Eigenschaftenseite mit einem ActiveX-Steuerelement verbinden

So verbinden Sie eine Eigenschaftenseite mit einem ActiveX-Steuerelement:

1. Fügen Sie *DefinePropertyPage* mit der GUID-Konstanten der Eigenschaftenseite als Parameter zur Implementierung der Methode *DefinePropertyPages* in der Implementierung des Steuerelements in der Unit hinzu. Beispiel:

   ```
   procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
   begin
         DefinePropertyPage(Class_PropertyPage1);
   end;
   ```

   Die GUID-Konstante *Class_PropertyPage1* der Eigenschaftenseite ist in der Eigenschaftenseiten-Unit enthalten.

   Der GUID ist in der Implementierungs-Unit der Eigenschaftenseite definiert und wird vom Experten für Eigenschaftenseiten automatisch generiert.

2. Fügen Sie die Eigenschaftenseiten-Unit zur **uses**-Klausel der Implementierungs-Unit für das Steuerelement hinzu.

Eigenschaften eines ActiveX-Steuerelements als published deklarieren

Die Eigenschaften eines ActiveX-Steuerelements oder eines Active-Formulars können in einer Entwicklungsumgebung wie Visual Basic, Delphi oder C++Builder erscheinen. Dann hat der Benutzer des Steuerelements oder des Formulars die Möglichkeit, dessen Eigenschaften zu verändern. Damit solche Eigenschaften angezeigt werden, müssen sie in der Typbibliothek des betreffenden Steuerelements definiert sein. Wenn die an den Eigenschaften vorgenommenen Änderungen von Dauer sein sollen, müssen sie als **published** deklariert werden. Sie können Eigenschaften entweder manuell in die Typbibliothek aufnehmen oder dies Delphi überlassen. Wenn Sie sich für letzteres entscheiden, fügt Delphi automatisch alle als **published** deklarierten Eigenschaften in die Typbibliothek ein.

Hinweis Eigenschaften, die nicht als **published** deklariert und die manuell in die Typbibliothek eingefügt wurden, erscheinen zwar in der jeweiligen Entwicklungsumgebung, doch sind die daren vorgenommenen Änderungen nicht persistent. Das bedeutet: Wenn der Benutzer des betreffenden Steuerelements den Wert einer Eigenschaft ändert, wird dies bei der Ausführung des Steuerelements ignoriert.

Wenn Sie eine Eigenschaft für ein Steuerelement, das Sie mit dem ActiveX-Steuerelement-Experten erstellt haben, für den Benutzer verfügbar machen möchten, müssen Sie die Quelle des VCL-Steuerelements aufrufen und dort die erforderlichen Eigenschaften als **published** deklarieren. Wenn es sich bei der Quelle um ein VCL-Objekt handelt und die betreffende Eigenschaft noch nicht als **published** deklariert ist, müssen Sie einen Nachkommen des VCL-Objekts erstellen und die Eigenschaft dort als **published** deklarieren. Beispiel: Wenn Sie die Eigenschaft *Align* von *Tbutton* als **published** deklarieren möchten, fügen Sie den folgenden Quelltext in die Implementierungs-Unit des ActiveX-Steuerelements ein:

```
TAlignButon = class (TButton)
   published
     property Align;
end;
```

So machen Sie eine Eigenschaft für ein Active-Formular (ActiveForm) für den Benutzer verfügbar:

1. Wählen Sie *Bearbeiten / Zur Schnittstelle hinzufügen*, während die Implementierungs-Unit im Editor geöffnet ist.
2. Wählen Sie in der Dropdown-Liste der Prozedurentypen den Eintrag *Properties/ Methods* aus.
3. Geben Sie die Deklaration für die Eigenschaft ein, welche die Eigenschaft des Steuerelements bereitstellt. Beispiel: Für die Eigenschaft *Caption* einer Schaltfläche könnten Sie folgendes eingeben:

   ```
   property MyButtonCaption: WideString;
   ```

4. Wählen Sie *OK*.

Hiermit werden Methoden zum Lesen und zum Setzen zur Implementierungs-Unit hinzugefügt.

5 Fügen Sie Quelltext zu den Methoden hinzu, mit denen die Eigenschaft des Steuerelements gelesen und gesetzt werden kann. Für das oben behandelte Schaltflächen-Steuerelement würde dieser Quelltext folgendermaßen lauten:

```
function TActiveFormX.Get_MyButtonCaption: WideString;
begin
  Result := Button1.Caption;
end;
procedure TActiveFormX.Set_MyButtonCaption(const Value: WideString);
begin
  Button1.Caption := Value;
end;
```

Da es sich bei einem Active-Formular (ActiveForm) um ein ActiveX-Steuerelement handelt, ist eine Typbibliothek damit verknüpft, und Sie können zusätzliche Eigenschaften und Methoden zum Active-Formular hinzufügen. Informationen darüber, wie Sie dazu vorgehen müssen, finden Sie Im Abschnitt. »Eigenschaften, Methoden und Ereignisse von ActiveX-Steuerelementen« auf Seite 48-11.

ActiveX-Steuerelemente registrieren

Nach der Erstellung muß das ActiveX-Steuerelement registriert werden, damit andere Anwendungen darauf zugreifen können.

So registrieren Sie ein ActiveX-Steuerelement:

- Wählen Sie *Start / ActiveX-Server eintragen.*

Hinweis Soll ein ActiveX-Steuerelement aus dem System entfernt werden, empfiehlt es sich, zuerst seine Registrierung aufzuheben.

So heben Sie die Registrierung eines ActiveX-Steuerelements auf:

- Wählen Sie *Start / ActiveX-Server austragen.*

Alternativ dazu können Sie den Befehl tregsvr in der Kommandozeile eingeben oder die Datei regsvr32.exe auf Betriebssystemebene ausführen.

ActiveX-Steuerelemente testen

Zum Testen des Steuerelements fügen Sie es in ein Package ein und importieren es als ActiveX-Steuerelement. Dadurch wird es in die Komponentenpalette von Delphi aufgenommen. Anschließend können Sie das Steuerelement in einem Formular ablegen und nach Belieben testen.

Ein ActiveX-Steuerelement sollte auch in allen Zielanwendungen getestet werden, in denen es voraussichtlich eingesetzt wird.

Zum Debuggen des ActiveX-Steuerelements wählen Sie *Start / Parameter* und geben den Client-Namen in das Eingabefeld Host-Anwendung ein.

Danach werden die Parameter für die Host-Anwendung eingesetzt. Mit *Start / Start* wird die Host- bzw. Client-Anwendung gestartet. Sie können im Steuerelement Haltepunkte setzen.

ActiveX-Steuerelemente im Web weitergeben

Damit ein neu erstelltes ActiveX-Steuerelement von Web-Clients verwendet werden kann, muß es auf dem Web-Server bereitgestellt werden. Nach jeder Änderung müssen Sie das ActiveX-Steuerelement neu compilieren und weitergeben, damit die Client-Anwendungen die Änderungen realisieren können.

Für die Weitergabe eines ActiveX-Steuerelements benötigen Sie einen Web-Server, der auf Client-Botschaften antworten kann. Sie können den Web-Server eines Drittanbieters verwenden (z. B. den mit Intrabuilder ausgelieferten Inprise Web Server) oder, wenn Sie über eine Delphi-Version mit Socket-Komponenten verfügen, einen eigenen Web-Server erstellen.

So geben Sie ein ActiveX-Steuerelement weiter:

1. Wählen Sie *Projekt / Optionen* für Distribution über das Web.
2. Geben Sie im Feld *Zielverzeichnis* der Registerkarte *Projekt* den Standort der DLL des ActiveX-Steuerelements auf dem Web-Server als Pfad an. Dies kann ein lokaler Pfadname oder ein UNC-Pfad sein, z. B. C:\INETPUB\wwwroot.
3. Geben Sie im Feld *Ziel-URL* den Standort der DLL des ActiveX-Steuerelements auf dem Web-Server als URL an, z. B. http://mymachine.inprise.com/. Den Dateinamen lassen Sie dabei weg. Einzelheiten hierzu finden Sie in der Dokumentation Ihres Web-Servers.
4. Legen Sie im Feld *HTML-Verzeichnis* den Standort (z. B. C:\INETPUB\ wwwroot.) fest, an dem die HTML-Datei, die eine Referenz auf das ActiveX-Steuerelement enthält, abgelegt werden soll. Dies kann ein Standard-Pfadname oder ein UNC-Pfad sein.
5. Stellen Sie die gewünschten Optionen für die Weitergabe über das Web ein, wie im Abschnitt »Optionen einstellen« auf Seite 448-26 beschrieben ist.
6. Wählen Sie *OK*.
7. Wählen Sie *Projekt / Distribution über das Web*.

   Dadurch wird eine Quelltextbasis für die Weitergabe erzeugt, die das ActiveX-Steuerelement in einer ActiveX-Bibliothek (mit der Erweiterung OCX) enthält. Je nach angegebenen Optionen kann diese Quelltextbasis für die Weitergabe zudem eine Cabinet-Datei (mit der Erweiterung CAB) oder eine Datei mit der Erweiterung INF (für INFormationen) enthalten.

   Die ActiveX-Bibliothek wird in dem in Schritt 2 angegebenen Zielverzeichnis abgelegt. Die HTML-Datei hat denselben Namen wie die Projektdatei, trägt aber die Namenserweiterung HTM. Sie wird in dem in Schritt 4 angegebenen HTML-Verzeichnis erzeugt und enthält eine URL-Referenz auf die ActiveX-Bibliothek, die sich an dem in Schritt 3 angegebenen Standort befindet.

Hinweis Wenn Sie diese Dateien auf Ihren Web-Server stellen möchten, verwenden Sie ein externes Hilfsprogramm wie z. B. ftp.

8 Rufen Sie Ihren ActiveX-fähigen Web-Browser auf, und zeigen Sie die neu erstellte HTML-Seite an.

Bei der Anzeige dieser HTML-Seite im Web-Browser wird Ihr Formular oder Steuerelement angezeigt und als eingebettete Anwendung innerhalb des Browsers ausgeführt. Dies bedeutet, daß die Bibliothek im selben Prozeß läuft wie die Browser-Anwendung.

Optionen einstellen

Bevor Sie ein ActiveX-Steuerelement weitergeben können, müssen Sie die Optionen für die Distribution über das Web angeben. Diese Optionseinstellungen müssen bei der Erzeugung der ActiveX-Bibliothek befolgt werden.

Das Dialogfeld *Optionen für Distribution über das Web* enthält Einstellungen, die folgendes ermöglichen:

CAB-Dateikompression verwenden	Eine Cabinet-Datei ist eine einzelne Datei (in der Regel mit der Namenserweiterung CAB), die andere, komprimierte Dateien als Dateibibliothek enthält. Durch eine CAB-Dateikompression kann die Zeit zum Herunterladen einer Datei um bis zu 70% verkürzt werden. Während der Installation dekomprimiert der Browser die in der CAB-Datei enthaltenen Dateien und kopiert sie in das System des Benutzers. Jede weitergegebene Datei kann zu einer CAB-Datei komprimiert werden.
Einstellungsoptionen für Packages	Sie können spezielle Optionen für jede als Bestandteil der Weitergabe erforderliche **Package**-Datei einstellen. Jedes dieser Packages kann in eine CAB-Datei eingefügt werden. Packages, die mit Delphi ausgeliefert wurden, sind bereits mit der Inprise-Signatur versehen.

Je nachdem, welche Kombination aus den Optionen für Packages und die CAB-Dateikompression Sie wählen, kann es sich bei der erzeugten ActiveX-Bibliothek um eine OCX-Datei, eine CAB-Datei, die eine OCX-Datei enthält, oder um eine INF-Datei handeln. Nähere Einzelheiten hierzu finden Sie in der Tabelle der Optionskombinationen weiter unten.

Optionen für Distribution über das Web, Kontrollfeld Vorgabe

Wenn Sie dieses Kontrollfeld aktivieren, werden die aktuellen Einstellungen aus den Seiten *Projekt*, *Packages* und *Zusätzliche Dateien* der Web-Weitergabeoptionen des Dialogfelds *Optionen für Distribution über das Web* als Standardoptionen gespeichert. Wenn dieses Kontrollfeld nicht aktiviert ist, betreffen die entsprechenden Einstellungen nur das offene ActiveX-Projekt.

Um die ursprünglichen Einstellungen wiederherzustellen, können Sie die Datei DEFPROJ.DOF löschen oder umbenennen.

INF-Datei

Wenn für ein ActiveX-Steuerelement Packages oder andere zusätzliche Dateien erforderlich sind, müssen diese zusammen mit dem Steuerelement weitergegeben werden. Bei der Weitergabe wird automatisch eine Datei mit der Erweiterung INF (für INFormation) erzeugt. In dieser Datei sind alle Dateien festgelegt, die heruntergeladen und eingerichtet werden müssen, damit die ActiveX-Bibliothek lauffähig ist. Die Syntax der INF-Datei erlaubt es, mit einer URL auf zu ladende Packages bzw. zusätzliche Dateien zu verweisen.

Die Optionen für die Distribution über das Web sind in den folgenden Registerkarten enthalten und werden in den folgenden Abschnitten beschrieben:

- Registerkarte *Projekt*
- Registerkarte *Packages*
- Registerkarte *Zusätzliche Dateien*

Optionskombinationen

In der folgenden Tabelle sind die Ergebnisse der verschiedenen Kombinationen von Web-Weitergabeoptionen für Packages und CAB-Dateikompression zusammengefaßt.

Packages und/ oder zusätzliche Dateien	CAB-Dateikompression	Ergebnis
Nein	Nein	Eine ActiveX-Bibliotheksdatei (OCX).
Nein	Ja	Eine CAB-Datei, die eine ActiveX-Bibliotheksdatei enthält.
Ja	Nein	Eine INF-Datei, eine ActiveX-Bibliotheksdatei und alle zusätzlichen Dateien und Packages.
Ja	Ja	Eine INF-Datei, eine CAB-Datei, die eine ActiveX-Bibliotheksdatei enthält, und je eine CAB-Datei für alle zusätzlichen Dateien und Packages.

Die Registerkarte Projekt

Über die Registerkarte *Projekt* können Sie neben dem Standort der Dateien und der URL weitere Optionen für die Weitergabe des Projekts festlegen. Die Optionen dieser Registerkarte gelten nur für die ActiveX-Bibilotheksdatei bzw. die CAB-Datei, in der die ActiveX-Bibilotheksdatei enthalten ist. Sie dienen als Standardeinstellungen für alle mit dem Projekt weitergegebenen Packages und zusätzlichen Dateien.

Verzeichnisse und URLs	Bedeutung
Zielverzeichnis	Die Position der ActiveX-Bibilotheksdatei auf dem Web-Server als Verzeichnispfad. Es kann ein Standard-Pfadname oder ein UNC-Pfad verwendet werden. **Beispiel:** C:\INETPUB\wwwroot
Ziel-URL	Der Pfad der ActiveX-Bibilotheksdatei auf dem Web-Server als URL. **Beispiel:** http://mymachine.inprise.com/
HTML-Verzeichnis	Der Pfad der HTML-Datei, die eine Referenz auf die ActiveX-Bibilotheksdatei enthält. Es kann ein Standard-Pfadname oder ein UNC-Pfad verwendet werden. **Beispiel:** C:\INETPUB\wwwroot

Hinweis Es handelt bei den angegebenen Positionen nicht um vollständige Dateinamen, sondern nur um Pfade.

Zusätzlich zur Angabe des Standorts der ActiveX-Datei können Sie in der Registerkarte *Projekt* festlegen, ob eine CAB-Dateikompression oder Versionsnummern hinzugefügt werden sollen. Die entsprechenden Optionen sind in der folgenden Tabelle aufgeführt:

Allgemeine Optionen	Bedeutung
CAB-Dateikompression verwenden	Sofern nicht anderweitig angegeben, werden die ActiveX-Bibliothek, die erforderlichen Packages und die zusätzlichen Dateien komprimiert. Mit der Cabinet-Kompression werden die Dateien in einer Dateibibliothek gespeichert, wodurch die zum Herunterladen einer Datei erforderliche Zeit um bis zu 70% verkürzt werden kann.
VersionsInfo der Datei übernehmen	Die in *Projekt / Optionen VersionsInfo* enthaltene Versionsnummer wird übernommen.
Versionsnr. autom. inkrementieren	Die in *Projekt / Optionen VersionsInfo* enthaltene Projekt-Versionsnummer wird automatisch inkrementiert.
Benötigte Packages weitergeben	Wenn diese Option aktiviert ist, werden alle für das Projekt erforderlichen Packages weitergegeben.
Zusätzliche Dateien weitergeben	Wenn diese Option aktiviert ist, werden alle Dateien, die in der Registerkarte *Zusätzliche Dateien* benannt wurden, zusammen mit dem Projekt weitergegeben.

Die Registerkarte Packages

Über die Registerkarte *Packages* können Sie angeben, wie die vom Projekt verwendeten Packages weitergegeben werden sollen. Für jedes Package können eigene Einstellungen festgelegt sein. Bei der Distribution Ihres ActiveX-Steuerelements können Sie individuelle Weitergabeoptionen für jede erforderliche Package-Datei angeben, die im Rahmen dieses Projekts an das Web weitergegeben wird. Jedes dieser Packages kann in CAB-Dateien komprimiert werden. Packages, die mit Delphi ausgeliefert wurden, sind mit der Inprise-Signatur versehen.

Von diesem Projekt verwendete Packages

Um die Einstellungen für ein bestimmtes Package zu ändern, markieren Sie dieses im Listenfeld *Von diesem Projekt verwendete Packages*. Anschließend können Sie die gewünschten Optionen festlegen.

CAB-Optionen

CAB-Option	Bedeutung
In separatem CAB komprimieren	Für das Package wird eine separate CAB-Datei erzeugt. Dies ist die Voreinstellung.
In Projekt-CAB komprimieren	Das Package wird in die CAB-Datei des Projekts aufgenommen.

VersionsInfo

Mit dem Kontrollkästchen *VersionsInfo* können Sie angeben, ob Versionsinformationen zum Package existieren. Die Versionsinformationen stammen aus der Ressource in der Package-Datei und werden in die INF-Datei für das Projekt eingetragen.

Optionen für Verzeichnis und URL

Optionen für Verzeichnis und URL	Bedeutung
Ziel-URL (leer, wenn die Datei auf den Zielsystemen existiert)	Der Standort des Package auf dem Web-Server als URL. Wenn keine Ziel-URL angegeben wird, nimmt der Web-Browser an, daß die Datei auf dem Zielcomputer existiert. Wird das Package dort nicht gefunden, schlägt das Herunterladen der ActiveX-Bibliotheksdatei fehl.
Zielverzeichnis (leer, wenn die Datei auf dem Server existiert)	Die Position des Package auf dem Web-Server als Pfad. Es kann ein Standard-Pfadname oder ein UNC-Pfad benutzt werden. Wenn Sie hier kein Zielverzeichnis angeben, wird davon ausgegangen, daß die Datei existiert. Sie stellen auf diese Weise sicher, daß die Datei nicht überschrieben wird.

Die Registerkarte Zusätzliche Dateien

Auf der Registerkarte *Zusätzliche Dateien* können Sie weitere Dateien (z. B. DLL-Dateien, INI-Dateien, Ressourcen usw.) angeben, die zusammen mit dem ActiveX-Steuerelement weitergegeben werden sollen.

Um eine Datei für die Weitergabe festzulegen, klicken Sie auf die Schaltfläche *Hinzufügen*. Im angezeigten Dialogfeld können Sie eine Datei auswählen. Jede hinzugefügte Datei wird im Listenfeld *Mit dem Projekt verbundene Dateien* angezeigt.

Mit dem Projekt verbundene Dateien

Um die Einstellungen für eine bestimmte Datei zu ändern, markieren Sie diese im Listenfeld *Mit dem Projekt verbundene Dateien*. Anschließend können Sie die gewünschten Optionen festlegen.

CAB-Optionen

CAB-Option	Bedeutung
In separatem CAB komprimieren	Für das Package wird eine separate CAB-Datei erzeugt. Dies ist die Voreinstellung.
In Projekt-CAB komprimieren	Das Package wird in die CAB-Datei des Projekts aufgenommen.

VersionsInfo

Mit dem Kontrollkästchen *VersionsInfo* können Sie angeben, daß die INF-Datei Versionsinformationen enthalten soll. Die Informationen werden aus der Ressource in der Datei angerufen.

Optionen für Verzeichnis und URL

Option für Verzeichnis und URL	Bedeutung
Ziel-URL (leer, wenn die Datei auf den Zielsystemen existiert)	Der Standort der Datei auf dem Web-Server als URL. Wenn Sie keine Ziel-URL angeben, nimmt der Web-Browser an, daß die Datei auf dem Zielcomputer existiert. Wird die Datei dort nicht gefunden, schlägt das Herunterladen der ActiveX-Bibliotheksdatei fehl.
Zielverzeichnis (leer, wenn die Datei auf dem Ziel-Server existiert)	Der Standort der zusätzlichen Datei auf dem Web-Server als Pfad. Es kann ein Standard-Pfadname oder ein UNC-Pfad verwendet werden. Wenn Sie kein Zielverzeichnis angeben, wird davon ausgegangen, daß die Datei auf dem Server existiert. Sie stellen auf diese Weise sicher, daß die Datei nicht überschrieben wird.

Kapitel

49

Eine Active-Server-Seite erstellen

Wenn Sie für Ihre Web-Seiten Microsoft Internet Information Server (IIS) verwenden, können Sie mit Hilfe von Active-Server-Seiten (Active Server Pages = ASP) dynamische Client/Server-Anwendungen für das Web erstellen. Active-Server-Seiten ermöglichen das Einbetten von Steuerelementen in eine Web-Seite, die dann automatisch aufgerufen werden, sobald der Server die Web-Seite lädt. Sie können beispielsweise einen Delphi-Automatisierungsserver schreiben, der ein Bitmap erstellt oder die Verbindung zu einer Datenbank herstellt. Dieses Steuerelement greift auf Daten zu, die bei jedem Laden der Web-Seite durch den Server aktualisiert werden.

Mit Active-Server-Seiten können Web-Anwendungen zur Verwendung von ActiveX-Server-Komponenten (Automatisierungsobjekte) erstellt werden. Es handelt sich um Komponenten auf dem Server, die Sie mit Delphi und vielen anderen Sprachen wie C++, Java oder Visual Basic entwickeln können. Auf dem Client ist die ASP ein HTML-Standarddokument und kann vom Benutzer unter jedem Betriebssystem mit einem beliebigen Web-Browser angezeigt werden.

Vor der Entwicklung dieser Technologie mußten die Client-Anwendungen auf jedem Computer installiert werden, der auf den Server zugreifen sollte. Mit diesem ASP-Modell wird ein Großteil der Client-Anwendung auf dem Server ausgeführt. Nur die eigentliche Benutzeroberfläche wird üblicherweise auf dem Client ausgeführt. Dies erleichtert die Aktualisierung der Clients, wenn eine Anwendung geändert wird.

Sie können die Active-Server-Seiten in Verbindung mit Microsoft Transaction Server einsetzen, um die Verwaltung von COM-Server-Komponenten zu automatisieren. Details zu MTS finden Sie in Kapitel 51, »MTS-Objekte erstellen«

Dieses Kapitel beschreibt das Erstellen einer Active-Server-Seite mit dem entsprechenden Delphi-Experten. Mit diesem Experten können Sie Eigenschaften und Methoden einer vorhandenen Anwendung für Automatisierungssteuerelemente bereitstellen.

Mit den folgenden Schritten erstellen Sie eine Active-Server-Seite aus einer vorhandenen Anwendung:

- Ein ASP-Objekt erstellen (für die Anwendung).

- Eigenschaften, Methoden und Ereignisse einer Anwendung für die Automatisierung bereitstellen.
- Registrieren der Anwendung als ASP-Objekt (Active Server Page).
- Testen und debuggen der Anwendung.

Grundlageninformationen zu den COM-Technologien finden Sie in Kapitel 44, »COM-Technologien im Überblick« Informationen zum Erstellen eines Automatisierungs-Controllers enthält Kapitel 46, »Automatisierungs-Controller erzeugen«.

Ein ASP-Objekt erstellen

Ein ASP-Objekt ist ein Automatisierungsobjekt, das für Web-Anforderungen auf die Schnittstellen zugreifen kann. Wie andere Automatisierungsobjekte handelt es sich um eine ObjectPascal-Klasse, die von *TAutoObject* abgeleitet wurde. *TAutoObject* unterstützt Automatisierungsprotokolle und macht sich für andere Anwendungen verfügbar. Sie erstellen ein ASP-Objekt mit dem entsprechenden Experten.

Öffnen Sie das Projekt einer Anwendung mit der bereitzustellenden Funktionalität, bevor Sie ein ASP-Objekt erstellen. Bei dem Projekt kann es sich in Abhängigkeit von den Anforderungen um eine Anwendung oder eine ActiveX-Bibliothek handeln.

Sie können *Server.CreateObject* in einer ASP-Seite verwenden, um nach Bedarf einen In-Process- oder Out-of-Process-Server zu starten. Beachten Sie in jedem Fall die Besonderheiten beim Starten eines Out-of-Process-Servers.

So öffnen Sie den Experten für Active-Server-Objekte:

1. Wählen Sie *Datei / Neu*.
2. Wählen Sie das Register mit der Beschriftung *ActiveX*.
3. Klicken Sie doppelt auf das Symbol *Active-Server-Objekt*.

Geben Sie im Experten folgendes an:

Name der CoClass	Geben Sie die Klasse an, deren Eigenschaften und Methoden Client-Anwendungen verfügbar gemacht werden sollen. (Delphi stellt dem Namen ein *T* voran.)
Instantiierung	Geben Sie einen Instantiierungsmodus an, der beschreibt, wie das ASP-Objekt gestartet wird. Informationen hierzu finden Sie unter »Instantiierungstypen für COM-Objekte« auf Seite 45-3. **Hinweis**: Wenn Sie das ASP-Objekt nur als In-Process-Server einsetzen, wird die Instantiierung ignoriert.

Threading-Modell	Wählen Sie das Threading-Modell, um anzugeben, wie Client-Anwendungen die Schnittstelle des Objekts aufrufen können. Dies ist das Threading-Modell, das Sie zur Implementierung im ASP-Objekt übergeben. Weitere Informationen zu Threading-Modellen finden Sie unter »Ein Threading-Modell auswählen« auf Seite 45-3. **Hinweis**: Die Auswahl des Threading-Modells bestimmt, wie das Objekt registriert wird. Sie müssen sicherstellen, daß Ihre Objektimplementierung dem gewählten Modell entspricht.
Active-Server-Typ	Wählen Sie für IIS 3 und IIS 4 Ereignismethoden auf Seitenebene. Dadurch wird ein ASP-Objekt erstellt, das die Methoden *OnStartPage* und *OnEndPage* implementiert. Diese Methoden werden vom Web-Server aufgerufen, wenn die Seite initialisiert wird und die Ausführung beendet. Verwenden Sie diese Option mit IIS 3 und IIS 4. Aktivieren Sie für IIS 5 den Objektkontext. Dieser nutzt die Funktionalität von MTS, um die richtigen Instanzdaten des Objekts abzurufen.
Ereignisunterstützung generieren	Generiert eine einfache ASP-Seite, die auf der Grundlage der *ProgID* das Delphi-Objekt erstellt.
Für dieses Objekt ein Vorlagen-Test-Script generieren	Markieren Sie dieses Kontrollkästchen, um den Experten zum Generieren einer separaten Schnittstelle zum Verwalten der Ereignisse für das ASP-Objekt zu veranlassen. Weitere Informationen zum Verwalten von Ereignissen finden Sie unter »Ereignisse in Ihrem Automatisierungsobjekt verwalten« auf Seite 47-3.

Nachdem Sie diese Prozedur abgeschlossen haben, wird eine neue Unit mit der Definition für das ASP-Objekt in das aktuelle Projekt eingefügt. Zusätzlich fügt der Experte ein Typbibliotheksprojekt hinzu und öffnet die Typbibliothek. Jetzt können Sie die Eigenschaften und Methoden der Schnittstelle über die Typbibliothek nach Maßgabe der Beschreibung unter »Eigenschaften, Methoden und Ereignisse einer Anwendung für die Automatisierung bereitstellen« auf Seite 47-3 verfügbar machen.

Das ASP-Objekt implementiert wie jedes andere Automatisierungsobjekt eine *duale Schnittstelle*, die sowohl das frühe Binden (beim Kompilieren) über die virtuelle Funktionstabelle (*VTable*) als auch das späte Binden (zur Laufzeit) über die Schnittstelle *IDispatch* unterstützt. Weitere Informationen zu dualen Schnittstellen finden Sie unter »Duale Schnittstellen« auf Seite 47-7.

ASP-Objekte für In-Process- oder Out-of-Process-Server erstellen

Sie können *Server.CreateObject* in einer ASP-Seite einsetzen, um nach Bedarf entweder einen In-Process-Server oder einen Out-of-Process-Server zu starten. Das Starten eines In-Process-Servers ist das gebräuchlichere Vorgehen.

DLLs mit In-Process-Komponenten sind schneller, sicherer und können von MTS-Systemen überwacht werden. Sie sind also besser für den Einsatz auf dem Server geeignet.

Die Out-of-Process-Server sind weniger sicher. Außerdem wird IIS häufig so konfiguriert, daß die Verwendung ausführbarer Out-of-Process-Dateien nicht zulässig ist. In diesen Fall wird eine Fehlermeldung wie die folgende ausgegeben:

```
Server-Objekt-Fehler 'ASP 0196'
Out-of-Process-Komponente kann nicht gestartet werden
/Pfad/outofprocess_exe.asp, Zeile 11
```

Außerdem erstellen Out-of-Process-Komponenten häufig individuelle Server-Prozesse für jede Objektinstanz und sind deshalb langsamer als CGI-Anwendungen. Sie können nicht so problemlos wie Komponenten-DLLs skaliert werden, die prozeßintern mit IIS oder MTS ausgeführt werden. Sind Leistung und Skalierbarkeit für Sie wichtige Kriterien, raten wir von der Verwendung Out-of-Process-Komponenten ab.

Intranet-Sites mit mittlerem bis geringem Datenaufkommen können unter Umständen Out-of-Process-Komponenten einsetzen, ohne die Gesamtleistung der Site zu beeinträchtigen.

Allgemeine Informationen zu Out-of-Process-Servern finden Sie unter »In-Process-Server, Out-of-Process-Server und Remote-Server« auf Seite 44-7.

Eine Anwendung als ASP-Objekt registrieren

Sie können die Active-Server-Seite als In-Process- oder Out-of-Process-Server registrieren. Normalerweise werden In-Process-Server verwendet. Informationen hierzu finden Sie unter »ASP-Objekte für In-Process- oder Out-of-Process-Server erstellen«.

Hinweis Wenn Sie ein ASP-Objekt aus dem System entfernen möchten, sollten Sie zunächst die Registrierung rückgängig machen, indem Sie die entsprechenden Einträge aus der Windows-Registrierung löschen.

Einen In-Process-Server registrieren

So registrieren Sie einen In-Process-Server (DLL oder OCX):

- Wählen Sie *Start* / *ActiveX-Server eintragen*.

So machen Sie die Registrierung eines In-Process-Servers rückgängig:

- Wählen Sie *Start* / *ActiveX-Server eintragen*.

Einen Out-of-Process-Server registrieren

So registrieren Sie einen Out-of-Process-Server:

- Starten Sie den Server mit der Kommandozeilenoption /regserver. (Sie können die Kommandozeilenoptionen in dem mit *Start / Parameter* zu öffnenden Dialogfeld einstellen.)

  Die Registrierung des Servers kann auch durch seinen Start erfolgen.

So machen Sie die Registrierung eines Out-of-Process-Servers rückgängig:

- Starten Sie den Server mit der Kommandozeilenoption /unregserver.

Die ASP-Anwendung testen

Das Testen eines In-Process-Servers ähnelt weitgehend dem Testen einer DLL. Sie können eine Host-Anwendung auswählen, welche die DLL lädt und dann wie gewohnt testen. So testen Sie ein ASP-Objekt:

1. Aktivieren Sie bei Bedarf die Debugger-Informationen auf der Registerkarte *Compiler* im Dialogfeld *Projekt / Optionen*. Aktivieren Sie außerdem die Option *Integrierte Fehlersuche* im Dialogfeld *Tools / Debugger-Optionen*.
2. Wählen Sie *Start / Parameter*, geben Sie den Namen des Web-Servers in das Feld *Host-Anwendung* ein, und wählen Sie *OK*.
3. Wählen Sie *Start / Start*.
4. Setzen Sie in der Active-Server-Seite Haltepunkte.
5. Interagieren Sie über den Web-Browser mit der Active-Server-Seite.

Die Ausführung der Active-Server-Seite wird unterbrochen, sobald die Haltepunkte erreicht werden.

Kapitel 4

50

Mit Typbibliotheken arbeiten

Dieses Kapitel beschreibt, wie Sie mit dem Typbibliothekseditor in Delphi Typbibliotheken erstellen und bearbeiten können. Typbibliotheken sind Dateien mit Informationen zu den Datentypen, Schnittstellen, Elementfunktionen und Objektklassen eines ActiveX-Steuerelements oder -Servers. Mit Hilfe einer Typbibliothek können Sie angeben, welche Objekte und Schnittstellen in Ihrem ActiveX-Server zur Verfügung stehen. Eine Übersicht über die Verwendung von Typbibliotheken finden Sie in Kapitel 44, »COM-Technologien im Überblick«.

Indem Sie Ihre COM-Anwendungen oder ActiveX-Bibliotheken mit einer Typbibliothek ausstatten, können Sie anderen Anwendungen oder Programmier-Tools Informationen über die Objekte in Ihrer COM-Anwendung zur Verfügung stellen.

In herkömmlichen Entwicklungsumgebungen erstellen Sie Typbibliotheken, indem Sie IDL- (Interface Definition Language) oder ODL-Skripts (Object Description Language) schreiben und diese mit dem entsprechenden Tool compilieren. Wenn Sie mit Delphi eigene Typbibliotheken erstellen, automatisiert der Typbibliothekseditor diesen Vorgang und nimmt Ihnen so einen Großteil der Arbeit ab.

Wenn Sie Ihr COM-Objekt, ActiveX-Steuerelement oder Automatisierungsobjekt mit Hilfe eines Experten erstellen, erzeugt der Typbibliothekseditor automatisch die Pascal-Syntax für Ihr vorhandenes Objekt. Während Sie die Typbibliothek mit dem Typbibliothekseditor bearbeiten, können die Änderungen automatisch in das zugeordnete Objekt übernommen oder überprüft und zurückgewiesen werden, wenn das Dialogfeld *Aktualisierung durchführen* aktiviert ist. (Einzelheiten hierzu finden Sie auf Seite 50-37.)

Außerdem können Sie den Typbibliothekseditor von Delphi zur Entwicklung von CORBA-Anwendungen (Common Object Request Broker Architecture) einsetzen. Mit herkömmlichen CORBA-Tools müssen Sie die Objektschnittstellen unter Verwendung der CORBA IDL (Interface Definition Language) unabhängig von der Anwendung definieren. Anschließend führen Sie ein Dienstprogramm aus, das aus dieser Definition den Stub-und-Skeleton-Quelltext generiert. Delphi generiert dagegen Stub, Skeleton und IDL automatisch. Sie können die Schnittstelle problemlos mit dem Typbibliothekseditor bearbeiten. Delphi aktualisiert die entsprechenden Quell-

textdateien automatisch. Weitere Informationen zu CORBA finden Sie in Kapitel 28, »CORBA-Anwendungen«.

Typbibliotheken können folgende Informationen enthalten:

- Informationen über Datentypen, einschließlich Aliasen, Aufzählungen, Strukturen und Unions.
- Beschreibungen eines oder mehrerer COM-Elemente, z. B. einer Schnittstelle, Dispatch-Schnittstelle oder CoClass. Jede dieser Beschreibungen nennt man normalerweise Typinformation.
- Beschreibungen der Konstanten und Methoden, die in externen Units definiert sind.
- Verweise auf Typbeschreibungen in anderen Typbibliotheken.

Dieses Kapitel enthält Informationen zu folgenden Themen:

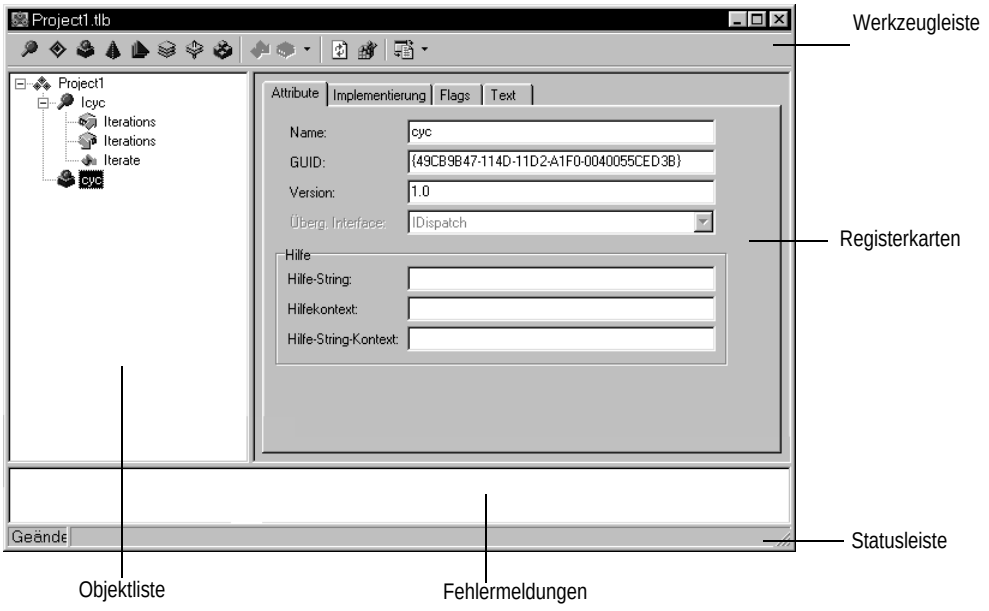
- Object-Pascal- oder IDL-Syntax verwenden
- Eine neue Typbibliothek erstellen
- Typbibliotheken weitergeben

Der Typbibliothekseditor

Mit Hilfe des Typbibliothekseditors können Typinformationen für ActiveX-Steuerelemente und COM-Objekte angezeigt und erstellt werden.

Abbildung 50.1 zeigt den Editor mit Informationen zu einer ActiveX-Schaltfläche.

Abbildung 50.1 Typbibliothekseditor



Zum Typbibliothekseditor gehören folgende Hauptelemente:

- Eine Werkzeugleiste, mit der Sie Ihrer Typbibliothek neue Schnittstellen und Schnittstellenelemente hinzufügen können.
- Eine Objektliste mit allen Objekten der Typbibliothek. Wenn Sie einen Eintrag in der Liste wählen, werden die für dieses Objekt gültigen Registerkarten angezeigt.
- Eine Statusleiste, in der beim Hinzufügen unzulässiger Typen die entsprechenden Syntaxfehler angezeigt werden.
- Registerkarten, in denen Informationen über das ausgewählte Objekt angezeigt werden. Je nach Objekttyp werden unterschiedliche Registerkarten angezeigt.
- Fehlerfenster mit den Fehlern, die beim Laden einer Typbibliothek erkannt wurden.

Die Werkzeugleiste

Auf der Werkzeugleiste des Typbibliothekseditors, die sich im oberen Bereich des Dialogfeldes befindet, sind Schaltflächen plaziert, über die Sie Ihrer Typbibliothek neue Objekte hinzufügen können.

Sie können über die Werkzeugleiste folgende Objekttypen hinzufügen

Symbol	Bedeutung
	Eine Typbibliothek. Kann aufgeklappt werden, so daß die einzelnen Typinformationen (einschließlich Objekten und Schnittstellen) zu sehen sind.
	Eine Schnittstellenbeschreibung
	Eine Beschreibung zur Dispatch-Schnittstelle
	Eine CoClass
	Eine Aufzählung
	Ein Alias
	Ein Record
	Eine Union
	Ein Modul
	Eine Methode der Schnittstelle, der Dispatch-Schnittstelle oder eines Einstiegspunkts in einem Modul
	Eine Put-By-Value-Eigenschaftsfunktion (Nur Schreiben)
	Eine Put-By-Reference-Eigenschaftsfunktion (Lesen / Schreiben / Schreiben per Ref.)
	Eine Get-Eigenschaftsfunktion (Nur Lesen)
	Eine Eigenschaft
	Ein Feld in einem Record oder in einer Union
	Eine Konstante in einer Aufzählung oder in einem Modul

Die Symbole in der linken Spalte (Interface, Dispatch, CoClass, Enum, Alias, Record, Union, und Module) stellen die Typinformationsobjekte dar, die Sie bearbeiten können.

Wenn Sie auf eine Schaltfläche der Werkzeugleiste klicken, erscheint das Symbol dieser Typinformation im unteren Bereich der Objektliste. Anschließend können Sie dessen Attribute im rechten Bereich des Dialogfeldes bearbeiten. Die angezeigten

Informationen im rechten Dialogfeldausschnitt hängen vom ausgewählten Typ des Symbols ab.

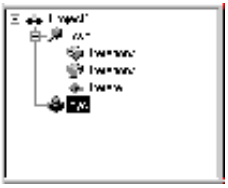
Wenn Sie ein Objekt auswählen, zeigt der Typbibliothekseditor die gültigen Elemente dieses Objekts an. Diese Elemente erscheinen auf der Werkzeugleiste im zweiten Dialogfeldbereich. Wählen Sie beispielsweise ein Objekt des Typs *Interface*, sind die Symbole *Methode* und *Property* im zweiten Dialogfeldbereich zu sehen, da Sie Ihrer Schnittstellendefinition Methoden und Eigenschaften hinzufügen können. Bei Auswahl eines *Enum*-Objekts wird nur das Symbol *Const* in die Werkzeugleiste aufgenommen, weil nur dieses Element für Enum-Typinformationen zulässig ist.

Im dritten Dialogfeldbereich können Sie eine Aktualisierung, eine Registrierung oder einen Export Ihrer Typbibliothek durchführen. Informationen hierzu finden Sie unter »Typinformationen speichern und registrieren« auf Seite 50-36.

Die Objektliste

Die Objektliste zeigt alle Elemente der aktuellen Typbibliothek an, beginnend mit der Schnittstelle. Die Schnittstelle kann aufgeklappt werden, so daß die Eigenschaften, Methoden und Ereignisse dieser Schnittstelle angezeigt werden:

Abbildung 50.2 Die Objektliste



Das lokale Menü im Objektlistenbereich enthält die folgenden Optionen:

Neu	Öffnet eine Liste der Objekte, die in die Typbibliothek eingefügt werden können. Diese Objekte sind auch in der Symbolleiste verfügbar.
Ausschneiden	Entfernt das ausgewählte Objekt und fügt es in die Windows-Zwischenablage ein.
Kopieren	Fügt das ausgewählte Objekt in die Windows-Zwischenablage ein.
Einfügen	Fügt ein Objekt in der Windows-Zwischenablage unter dem ausgewählten Objekt ein.
Löschen	Entfernt das ausgewählte Objekt.
Fehler anzeigen	Aktiviert bzw. deaktiviert die Anzeige des Fehlerfensters.
Symbolleiste	Aktiviert bzw. deaktiviert die Anzeige der Symbolleiste.

Die Statusleiste

Beim Bearbeiten oder Speichern einer Typbibliothek werden im Bereich der Statusleiste Syntax- und Übersetzungsfehler sowie Warnungen angezeigt.

Wenn Sie beispielsweise einen nicht vom Typbibliothekseditor unterstützten Typ angeben, wird ein Syntaxfehler angezeigt. Eine vollständige Aufstellung der vom Typbibliothekseditor unterstützten Typen finden Sie im Abschnitt »Gültige Typen« auf Seite 50-25.

Registerkarten mit Typinformationen

Wenn Sie in der Objektliste ein Objekt auswählen, zeigt der Editor die für diesen Typ verfügbaren Registerkarten mit Typinformationen an. Die folgende Tabelle zeigt, welche Registerkarten für welchen Objekttyp angezeigt wird:

Tabelle 50.1 Die Registerkarten der Typbibliothek

Objekt	Registerkarten
Typbibliothek	Attribute, Verwendet, Flags, Text
Schnittstelle	Attribute, Flags, Text
Dispatch-Schnittstelle	Attribute, Flags, Text
CoClass	Attribute, Implementierung, Flags, Text
Aufzählung	Attribute, Text
Alias	Attribute, Text
Record	Attribute, Text
Union	Attribute, Text
Modul	Attribute, Text
Methode	Attribute, Parameter, Flags, Text
Eigenschaft	Attribute, Parameter, Flags, Text
Konstante	Attribute, Flags, Text
Feld	Attribute, Flags, Text

Die Registerkarte Attribute

Alle Elemente einer Typbibliothek verfügen über die Registerkarte *Attribute*, in der Sie einen Namen und andere spezifische Attribute definieren können. So können beispielsweise bei einer Schnittstelle die GUID und die übergeordnete Schnittstelle und bei einem Feld dessen Typ angegeben werden. In den nachfolgenden Abschnitten finden Sie Informationen zu den Attributen der verschiedenen Elemente.

Bei Attributen, die für alle Elemente in der Typbibliothek zur Verfügung stehen, kann ein beschreibender Hilfetext eingegeben werden. Sie sollten diese Möglichkeit bei Ihren eigenen Typbibliotheken auf jeden Fall nutzen, um die Verwendung durch andere Anwendungen zu erleichtern.

Der Typbibliothekseditor unterstützt zwei Mechanismen für das Bereitstellen von Hilfeinformationen. Sie können die Informationen in einer Hilfedatei (herkömmliche Methode) oder einer separaten DLL (für Lokalisierungszwecke) speichern.

Die folgenden Hilfeattribute können für alle Elemente verwendet werden:

Tabelle 50.2 Gemeinsame Attribute aller Typen:

Attribut	Bedeutung
Hilfe-String	Eine kurze Beschreibung des Elements.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Dieses Attribut wird verwendet, wenn für das Hilfedateiattribut der Typbibliothek eine Windows-Standardhilfedatei angegeben wurde.
Hilfe-String-Kontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfe-DLL kennzeichnet. Dieses Attribut wird verwendet, wenn für das Attribut *Hilfe-String-DLL* der Typbibliothek eine Hilfe-DLL angegeben wurde.

Hinweis Die Hilfedatei muß vom Entwickler separat bereitgestellt werden.

Die Registerkarte Text

Alle Elemente einer Typbibliothek verfügen über die Registerkarte *Text* mit der Syntax für das Element. Diese Syntax gilt für eine Teilmenge der IDL (Microsoft Interface Definition Language) oder Object Pascal. Alle in den anderen Registerkarten des Elements vorgenommenen Änderungen werden auf der Registerkarte Text angezeigt. Wenn Sie Quelltext direkt eingeben, werden die anderen Registerkarten automatisch aktualisiert.

Der Typbibliothekseditor gibt Syntaxfehler aus, wenn Sie nicht unterstützte Bezeichner verwenden. Gegenwärtig werden Bezeichner unterstützt, die sich auf Typbibliothekunterstützung beziehen (nicht auf RPC-Unterstützung, vom Microsoft IDL Compiler für die C++-Codegenerierung verwendete Konstrukte oder die Marshalling-Unterstützung).

Die Registerkarte Flags

Einige Elemente einer Typbibliothek verfügen über Flags, mit denen bestimmte Merkmale oder implizite Markmale aktiviert oder deaktiviert werden können. In späteren Abschnitten dieses Kapitels folgt eine genaue Beschreibung der Flags für die verschiedenen Elemente einer Typbibliothek.

Typbibliotheksinformationen

Wenn in der Objektliste die Typbibliothek (oberster Knoten) ausgewählt ist, können Sie in den folgenden Registerkarten ihre Typinformationen ändern:

- Attribute
- Verwendet
- Flags

Die Registerkarte Attribute für Typbibliotheken

In der Registerkarte *Attribute* werden Typinformationen über die ausgewählte Typbibliothek angezeigt:

Tabelle 50.3 Typbibliothekattribute

Attribut	Bedeutung
Name	Der beschreibende Name der Typbibliothek (ohne Leer- und Interpunktionszeichen).
GUID	Der global eindeutige, 128 Bit lange Bezeichner der Typbibliothek.
Version	Die Version der Bibliothek (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als eine Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen gibt die Haupt-, die zweite die Unterversionsnummer an. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
LCID	Ein Bezeichner, der den Sprachtreiber für alle Text-Strings in der Typbibliothek und in Elementen angibt.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.
Hilfe-String-DLL	Der vollständige Name der für Hilfeinformationen verwendeten DLL, sofern vorhanden.
Hilfedatei	Der Name der Hilfedatei, die mit der Typbibliothek verknüpft ist, sofern vorhanden.

Die Registerkarte Verwendet für Typbibliotheken

Die Registerkarte *Verwendet* listet die Namen und Verzeichnisse aller Typbibliotheken auf, die diese Typbibliothek referenziert.

Die Registerkarte Flags für Typbibliotheken

Die nachfolgenden Flags erscheinen auf der Registerkarte *Flags*, wenn eine Typbibliothek markiert ist. Sie geben an, wie andere Anwendungen den mit dieser Typbibliothek verknüpften Server verwenden müssen:

Tabelle 50.4 Flags von Typbibliotheken

Flag	Bedeutung
Restricted	Stellt sicher, daß die Bibliothek von einem Makro-Programmierer nicht verwendet werden kann.
Control	Die Bibliothek stellt ein Steuerelement dar.
Hidden	Kennzeichnet die Bibliothek als vorhanden, verhindert jedoch, daß sie in einem benutzerorientierten Browser angezeigt wird.

Schnittstelleninformationen

Die Schnittstelle beschreibt die Methoden (sowie alle Eigenschaften, die als get- und set-Funktionen implementiert sind) eines Objekts, auf die über eine virtuelle Funktionstabelle (VTable) zugegriffen werden muß. Besitzt eine Schnittstelle das Flag *Dual* (Voreinstellung), ist eine Dispatch-Schnittstelle impliziert, auf die über die OLE-Automatisierung zugegriffen werden kann.

Sie können eine Schnittstelle ändern, indem Sie folgende Aktionen durchführen:

- Attribute ändern
- Flags ändern
- Schnittstellenelemente hinzufügen, entfernen oder ändern

Die Registerkarte Attribute für Schnittstellen

Die Registerkarte *Attribute* listet folgende Typinformationen auf:

Tabelle 50.5 Schnittstellenattribute

Attribut	Bedeutung
Name	Der Name der Schnittstelle.
GUID	Der global eindeutige, 128 Bit lange Bezeichner der Schnittstelle (optional).

Tabelle 50.5 Schnittstellenattribute (Fortsetzung)

Attribut	Bedeutung
Version	Die Version der Bibliothek (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als eine Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen gibt die Haupt-, die zweite die Unterversionsnummer an. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Überg. Interface	Der Name der Basisschnittstelle der ausgewählten Schnittstelle. Alle COM-Schnittstellen müssen ursprünglich von *IUnknown* abgeleitet sein.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Die Registerkarte Flags für Schnittstellen

Folgende Flags sind verfügbar, wenn eine Schnittstelle im Hauptbereich des Dialogfelds markiert ist.

Tabelle 50.6 Schnittstellen-Flags

Flag	Bedeutung
Hidden	Kennzeichnet die Schittstelle als vorhanden, verhindert jedoch, daß sie in einem benutzerorientierten Browser angezeigt wird.
Dual	Die Schnittstelle ermöglicht den Zugriff auf Eigenschaften und Methoden über *IDispatch* und direkt über eine virtuelle Funktionstabelle.
Ole Automation	Die Schnittstelle kann nur Typen verwenden, die zur OLE-Automatisierung kompatibel sind. Dieses Flag ist für Dispinterface-Objekte nicht zulässig, da diese durch ihre Definition automatisierungskompatibel sind.
Non-extensible	Die Schnittstelle kann nicht als Basisschnittstelle für eine andere Schnittstelle verwendet werden.

Schnittstellenelemente

Mit dem Typbibliothekseditor können Sie folgendes neu erzeugen oder anpassen:

- Eigenschaften
- Methoden

Sie können über die Registerkarte *Parameter* auch Eigenschafts- und Methodenparameter ändern.

Interface-Objekte (Schnittstellen) werden häufiger als Dispinterface-Objekte (Dispatch-Schnittstellen) dazu verwendet, die Eigenschaften und Methoden eines Objekts zu beschreiben.

Bei Schnittstellenelementen, die Exceptions auslösen müssen, sollte ein HRESULT-Wert zurückgegeben und der Parameter (PARAM_RETVAL) für den Rückgabewert definiert werden. Deklarieren Sie diese Methoden unter Verwendung der **Safecall**-Aufrufkonvention.

Schnittstellenmethoden

Wenn Sie mit dem Symbol *Methode* eine neue Methode hinzufügen, werden die für eine Methode zulässigen Registerkarten im Typbibliothekseditor angezeigt: *Attribute*, *Parameter*, *Flags* und *Text*.

Methodenattribute

Folgende Attribute können für Schnittstellenmethoden angegeben werden:

Tabelle 50.7 Methodenattribute

Attribut	Bedeutung
Name	Der Name des Elements.
ID	Die Dispatch-ID.
Aufruf	Dieses Attribut gibt an, ob die Methode bzw. Eigenschaft eine Funktion ist. Geben Sie für Methoden *Funktion* an.

Methodenparameter

Die Parameter für Methoden werden so angegeben, wie es im Abschnitt »Die Registerkarte Parameter für Eigenschaften und Methoden« auf Seite 50-13 beschrieben ist.

Methoden-Flags

Folgende Flags können für eine Schnittstellenmethode verwendet werden:

Tabelle 50.8 Methoden-Flags

Flag	IDL-Bezeichner	Bedeutung
Replaceable	replaceable	Das Objekt unterstützt *IConnectionPointWithDefault*.
Restricted	restricted	Die Eigenschaft oder Methode kann nicht von anderen Programmierern verwendet werden.
Source	source	Das Element gibt ein Objekt oder eine Variante zurück, das bzw. die eine Quelle von Ereignissen ist.
Bindable	bindable	Das Element unterstützt Datenbindung.

Tabelle 50.8 Methoden-Flags (Fortsetzung)

Flag	IDL-Bezeichner	Bedeutung
Hidden	hidden	Kennzeichnet die Eigenschaft als vorhanden, verhindert jedoch, daß sie in einem benutzerorientierten Browser angezeigt wird.
UI Default	uidefault	Zeigt an, daß das Typinformationselement das Standard-Element für die Anzeige in der Benutzeroberfläche ist.

Schnittstelleneigenschaften

Schnittstelleneigenschaften werden durch get- und set-Methoden repräsentiert, die dem Lesen und Schreiben der Daten dienen, die der Eigenschaft zugrundeliegen. Im Baumdiagramm werden sie mit speziellen Symbolen angezeigt, die auf ihren Zweck verweisen.

Hinweis Wenn ActiveX-Eigenschaften als Schreiben-per-Referenz definieren, werden sie als Zeiger und nicht als Wert übergeben. Einige Anwendungen wie Visual Basic nutzen das Schreiben per Referenz (sofern verfügbar), um die Leistung zu optimieren. Soll die Eigenschaft nur als Referenz und nicht als Wert übergeben werden, verwenden Sie den Eigenschaftstyp *Nur per Referenz*. Soll die Eigenschaft per Referenz und als Wert übergeben werden, wählen Sie *Lesen / Schreiben / Schreiben per Ref*. Sie können dieses Menü öffnen, indem Sie in der Symbolleiste auf den Pfeil neben dem Eigenschaftssymbol klicken.

Eigenschaftsattribute

Folgende Attribute sind für eine Schnittstelleneigenschaft verfügbar:

Tabelle 50.9 Eigenschaftsattribute

Attribut	Beschreibung
Name	Name des Elements.
ID	Die Dispatch-ID.
Typ	Der Typ der Eigenschaft. Hier kann jeder der »Gültige Typen« auf Seite 50-25 aufgeführten Typen verwendet werden.
Aufruf	Dieses Attribut gibt an, ob es sich bei der Eigenschaft um eine Abruffunktion oder um eine Zuweisungsfunktion (Wert oder Referenz) handelt.

Eigenschafts-Flags

Folgende Flags können für eine Schnittstelleneigenschaft angegeben werden:

Tabelle 50.10 Eigenschafts-Flags

Flag	IDL-Bezeichner	Bedeutung
Replaceable	replaceable	Das Objekt unterstützt *IConnectionPointWithDefault*.
Restricted	restricted	Die Eigenschaft kann nicht von anderen Programmierern verwendet werden.

Tabelle 50.10 Eigenschafts-Flags (Fortsetzung)

Flag	IDL-Bezeichner	Bedeutung
Source	source	Zeigt an, daß das Element ein Objekt oder eine Variante zurückgibt, die eine Ereignissquelle ist.
Bindable	bindable	Die Eigenschaft unterstützt Datenbindung.
Request Edit	requestedit	Die Eigenschaft unterstützt *OnRequestEdit*-Benachrichtigungen.
Display Bindable	displaybind	Die Eigenschaft wird dem Benutzer als bindbar angezeigt.
Default Bindable	defaultbind	Die bindbare Eigenschaft, die das Objekt am besten repräsentiert. Für Eigenschaften, bei denen dieses Flag gesetzt ist, muß auch *bindable* gesetzt sein. Dieses Flag kann in einem Dispinterface-Objekt nur für eine Eigenschaft gesetzt werden.
Hidden	hidden	Die Eigenschaft ist vorhanden, wird aber nicht in einem Objekt-Browser angezeigt.
Default Collection Element	defaultcollelem	Visual-Basic-Quelltext kann optimiert werden.
UI Default	uidefault	Zeigt an, daß das Typinformationselement das Standard-Element für die Anzeige in der Benutzeroberfläche ist.
Non Browsable	nonbrowsable	Die Eigenschaft wird in einem Objekt-Browser angezeigt, der keine Eigenschaftswerte anzeigt. Sie erscheint jedoch nicht in Anzeigeprogrammen, die Eigenschaftswerte anzeigen.
Immediate Bindable	immediatebind	Erlaubt einzelnen bindbaren Eigenschaften in einem Formular die unmittelbare Bindung. Ist das Flag gesetzt, werden bei allen Änderungen Benachrichtigungen gesendet. Dieses neue Flag wirkt sich nur aus, wenn auch die Bits *bindable* und *requestedit* gesetzt sind.

Die Registerkarte Parameter für Eigenschaften und Methoden

In der Registerkarte *Parameter* können Sie die Parameter und Rückgabewerte für Ihre Funktionen angeben.

Bei Eigenschaftsfunktionen entspricht der Typ der Eigenschaft entweder dem Typ des Rückgabewertes oder dem des letzten Parameters. Wenn Sie den Typ in der Registerkarte *Parameter* ändern, wird der angezeigte Eigenschaftstyp in der Registerkarte *Attribute* automatisch aktualisiert. Entsprechend wird der Inhalt der Registerkarte *Parameter* an Änderungen in der Registerkarte *Attribute* angepaßt.

Hinweis Das Ändern einer Eigenschaftsfunktion wirkt sich auf alle mit ihr in Beziehung stehenden Funktionen aus. Eigenschaftsfunktionen stehen im Typbibliothekseditor zueinander in Beziehung, wenn sie identische Namen und Dispatch-IDs haben.

Die Anzeige in der Registerkarte Parameter variiert in Abhängigkeit davon, ob Sie in IDL oder in Object Pascal arbeiten.

Wenn Sie bei der Arbeit mit Object Pascal einen Parameter hinzufügen, zeigt der Typbibliothekseditor folgendes an:

- Modifizierer
- Name
- Typ
- Standardwert

Wenn Sie in IDL arbeiten, zeigt der Typbibliothekseditor folgendes an:

- Name
- Typ
- Modifizierer

Sie können den Namen ändern, indem Sie einfach einen neuen Bezeichner angeben. Sie ändern den Typ, indem Sie einen neuen Wert aus der Dropdown-Liste auswählen. Die vom Typbibliothekseditor unterstützten, für Typen möglichen Werte sind unter »Gültige Typen« auf Seite 50-25 aufgeführt.

Wenn Sie in Object Pascal arbeiten, ändern Sie den Modifizierer, indem Sie einen neuen Wert aus der Dropdown-Liste auswählen. Folgende Werte sind erlaubt:

Tabelle 50.11 Parameter-Modifizierer (Object-Pascal-Syntax)

Modifizierer	Bedeutung
Leer (Standard)	Eingabeparameter. Es kann sich hierbei um einen Zeiger handeln. Der Wert, auf den er sich bezieht, wird nicht zurückgeliefert (entspricht [In] in IDL).
None	Für die Sequenzbildung von Parameterwerten wird keine Information bereitgestellt. Dieser Modifizierer sollte nur bei den Dispatch-Schnittstellen verwendet werden, da für diese keine Sequenzbildung stattfindet (entspricht dem Fehlen von Flags in IDL).
Out	Ausgabeparameter. Dies ist ein Referenzwert, der das Ergebnis empfängt (entspricht [Out] in IDL).
Optionalout	Ein optionaler Ausgabewert. Dies muß ein Variantentyp sein und alle nachfolgenden Parameter müssen optional sein (entspricht [Out, Optional] in IDL).
Var	Eingabe- / Ausgabe-Parameter. Eine Kombination aus Leer und Out (entspricht [In, Out] in IDL).
Optionalvar	Ein optionaler Eingabe- / Ausgabe-Parameter. Stellt eine Kombination der Parameter Optional und Optionalout dar (entspricht [In, Out, Optional] in IDL).
Optional	Ein optionaler Eingabeparameter. Er muß ein Variantentyp sein und alle nachfolgenden Parameter müssen optional sein (entspricht [In, Optional] in IDL).
Retval	Empfängt den Rückgabewert. Rückgabewerte werden als letzte Parameter aufgelistet (wie vom Typbibliothekseditor angefordert). Parameter mit diesem Wert werden in benutzerorientierten Browsern nicht angezeigt. Kann nur dann angewendet werden, wenn die Funktion ohne Safecall-Direktive deklariert wurde (entspricht [Out, RetVal] in IDL).

In der Spalte *Vorgabewert* geben Sie die Standardparameter ein. Wenn Sie einen Standardwert hier angeben, fügt der Typbibliothekseditor automatisch die entsprechenden Flags in die Typbibliothek ein.

Um ein Parameter-Flag zu ändern (bei der Arbeit in IDL), doppelklicken Sie auf das Feld mit dem Modifizierer. Im Dialogfeld *Parameter-Flags* stehen Ihnen folgende Auswahlmöglichkeiten zur Verfügung:

Tabelle 50.12 Parameter-Flags (IDL-Syntax)

Flag	Bedeutung
In	Ein Eingabeparameter, der auch ein Zeiger sein kann. Der referenzierte Wert wird jedoch nicht zurückgegeben.
Out	Ein Ausgabeparameter, der ein Zeiger auf ein Element sein muß, in dem das Funktionsergebnis gespeichert wird.
RetVal	Empfängt den Rückgabewert. Rückgabewerte müssen ein Out-Attribut darstellen und als letzter Parameter aufgelistet sein (wird vom Typbibliothekseditor überwacht). Parameter mit diesem Wert werden in benutzerorientierten Browsern nicht angezeigt.
LCID	Die lokale ID. Dieses Flag darf nur für einen Parameter gesetzt werden. Dieser Parameter muß außerdem das Attribut *in* und den Typ **long** haben. Auf diese Weise können Elemente in der virtuellen Tabelle beim Aufruf eine LCID erhalten. Parameter mit diesem Wert werden nicht in einem Objekt-Browser angezeigt. Laut Konvention ist LCID der Parameter vor *RetVal*. Bei Dispinterface-Objekten ist LCID nicht zulässig.
Optional	Ein optionaler Parameter. Natürlich müssen auch alle nachfolgenden Parameter optional sein.
Hat Vorgabewert	Standardwert für einen typisierten optionalen Parameter. Der Wert muß eine Konstante sein, die als Variante gespeichert werden kann.
Vorgabewert	Wenn Sie *Hat Vorgabewert* gewählt haben, geben Sie hier den Wert an. Er muß mit dem Typ des optionalen Parameters übereinstimmen.

Hinweis Bei der Arbeit mit IDL werden Standardwerte mit Hilfe von Flags angegeben, statt in einer separaten Spalte. Darüber hinaus werden lokale IDs mit Hilfe von Flags festgelegt, statt über die Verwendung von TLCID-Paramater-Typspezifizierern.

Die Reihenfolge der Parameter kann mit Hilfe der Schaltflächen *Nach Oben* und *Nach Unten* geändert werden. Die Operation wird jedoch nicht durchgeführt, wenn dadurch eine IDL-Sprachregel verletzt wird. Der Typbibliothekseditor überwacht beispielsweise die Regel, daß Rückgabewerte immer der letzte Parameter in der Liste sein müssen.

Dispatch-Typinformationen

Schnittstellen werden häufiger als Dispatch-Schnittstellen dazu verwendet, die Eigenschaften und Methoden eines Objekts zu beschreiben. Auf Dispatch-Schnittstellen kann nur über das dynamische Binden zugegriffen werden, während für Schnittstellen das statische Binden über eine virtuelle Funktionstabelle möglich ist.

Sie können eine Dispatch-Schnittstelle (Dispinterface) ändern, indem Sie:

- Attribute ändern
- Flags ändern
- Schnittstellenelemente hinzufügen, entfernen oder ändern

Die Registerkarte Attribute für Dispatch-Schnittstellen

Die nachfolgenden Attribute beziehen sich auf die Dispatch-Schnittstelle:

Tabelle 50.13 Dispinterface-Attribute

Attribut	Bedeutung
Name	Der Name der Dispatch-Schnittstelle. Er muß innerhalb der Typbibliothek eindeutig sein.
GUID	Der global eindeutige, 128 Bit lange Bezeichner der Dispatch-Schnittstelle (optional).
Version	Die Version der Dispatch-Schnittstelle (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Die Registerkarte Flags für Dispatch-Schnittstellen

Die Registerkarte *Flags* für Dispatch-Schnittstellen ist mit der gleichnamigen Registerkarte für Interface-Objekte (Schnittstellen) identisch. (Siehe »Die Registerkarte Flags für Schnittstellen« auf Seite 50-10)

Dispatch-Elemente

Für Dispatch-Schnittstellen können folgende Elemente definiert werden:

- Methoden
- Eigenschaften

Sie können Methoden und Eigenschaften zu Dispatch-Schnittstellen auf die gleiche Weise wie zu Schnittstellen hinzufügen (siehe »Die Registerkarte Parameter für Eigenschaften und Methoden« auf Seite 50-13).

Beachten Sie beim Erstellen einer Eigenschaft für eine Dispatch-Schnittstelle, daß Sie keine Funktionsart oder Parametertypen angeben können. Die Eigenschafts- und Methoden-Flags sind mit denen für Schnittstellen identisch (siehe »Schnittstellenmethoden« auf Seite 50-11 und »Schnittstelleneigenschaften« auf Seite 50-12).

CoClass-Typinformationen

CoClass beschreibt ein eindeutiges COM-Objekt, das mindestens eine Schnittstelle implementiert und die für das Objekt implementierte Standardschnittstelle sowie optional die Dispatch-Schnittstelle angibt, welche die Standardquelle für Ereignisse bildet.

Mit einer CoClass-Definition im Typbibliothekseditor können Sie folgende Aktionen durchführen:

- Attribute ändern
- Inhalt der Regiterkarte *Implementierung ändern*
- Flags ändern

Die Registerkarte Attribute für CoClass-Objekte

Die Registerkarte *Attribute* enthält für CoClass-Objekte folgende Informationen:

Tabelle 50.14 CoClass-Attribute

Attribut	Bedeutung
Name	Der Name des CoClass-Objekts.
GUID	Der global eindeutige, 128 Bit lange Bezeichner des CoClass-Objekts (optional).
Version	Die Version des CoClass-Objekts (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer der Schnittstelle. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Die Registerkarte Implementierung für CoClass-Objekte

Die Registerkarte *Implementierung* dient der Angabe der für das CoClass-Objekt zu implementierenden Schnittstellen und Dispatch-Schnittstellen. In der Registerkarte *Implementierung* kann für jede Schnittstelle festgelegt werden, ob die folgenden Elemente unterstützt werden:

Tabelle 50.15 Optionen der CoClass-Registerkarte Implementierung

Schnittstelle	Beschreibung
Standardschnittstelle	Der Name einer Schnittstelle, die vom CoClass-Objekt implementiert wird.
GUID	Die GUID der Elementschnittstelle des Objekts.
Quelle	Das Element kann Ereignisse auslösen.
Vorgabe	Die Schnittstelle oder Dispatch-Schnittstelle wird als Standardschnittstelle verwendet. Diese Schnittstelle wird standardmäßig beim Erstellen einer Instanz der Klasse zurückgegeben. Ein CoClass-Objekt kann höchstens zwei Standardelemente haben. Eines ist die primäre Schnittstelle oder -Dispatch-Schnittstelle, die andere die optionale Dispatch-Schnittstelle, die als Ereignisquelle dient.
Restricted	Das Element kann nicht von anderen Programmierern verwendet werden. Bei einem Schnittstellenelement können die Attribute *Restricted* und *Vorgabe* nicht zusammen angegeben werden.
VTable	Das Objekt kann zwei verschiedene Quellschnittstellen haben.

Das lokale Menü der Registerkarte *Implementierung* enthält Menübefehle zum Aktivieren bzw. Deaktivieren der obigen Optionen. Die Menüoption *Standardschnittstelle einfügen* öffnet ein Dialogfeld, in dem Sie eine Schnittstelle auswählen können, die dem CoClass-Objekt hinzugefügt werden soll. Es wird eine Liste mit den Schnittstellen angezeigt, die in der aktuellen und in den referenzierten Typbibliotheken definiert sind.

Die Registerkarte Flags für CoClass-Objekte

Folgende Flags sind zulässig, wenn im Hauptbereich der Objektliste eine CoClass ausgewählt ist.

Tabelle 50.16 CoClass-Flags

Flag	Bedeutung
Hidden	Die Schnittstelle wird nicht in einem Objekt-Browser angezeigt.
Can Create	Eine Instanz kann mit CoCreateInstance erstellt werden.
Application Object	Das CoClass-Objekt ist ein Anwendungsobjekt, dem eine vollständige EXE-Anwendung zugeordnet ist. Die Funktionen und Eigenschaften des Objekts sind in der Typbibliothek global verfügbar.
Licensed	Das CoClass-Objekt ist lizenziert und muß mit IClassFactory2 instantiiert werden.

Tabelle 50.16 CoClass-Flags (Fortsetzung)

Flag	Bedeutung
Predefined	Die Client-Anwendung sollte automatisch eine einzelne Instanz eines Objekts mit diesem Attribut erstellen.
Control	Das CoClass-Objekt ist ein ActiveX-Steuerelement, von dem ein Container weitere Typbibliotheken oder CoClasses ableitet.
Aggregatable	Die Elemente der Klasse können in ein Aggregat-Objekt eingebunden werden.
Replaceable	Das Objekt unterstützt *IConnectionPointWithDefault*.

Enumeration-Typinformationen

Das Ändern und Hinzufügen von Enumeration-Definitionen in der Typbibliothek umfaßt folgende Aktionen:

- Attribute ändern
- Enum-Elemente hinzufügen, entfernen oder ändern

Die Registerkarte Attribute für Enum-Objekte

Die Registerkarte *Attribute* enthält für Enum-Objekte folgende Informationen:

Tabelle 50.17 Enum-Attribute

Attribut	Bedeutung
Name	Der beschreibende Name des Enum-Objekts.
GUID	Der global eindeutige, 128 Bit lange Bezeichner der Schnittstelle (optional).
Version	Die Version des Enum-Objekts (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer. Wird nur eine Zahl angegeben, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Sie sollten für Ihre Enum-Objekte immer Hilfe-Strings angeben, die deren Bedeutung beschreiben. Das folgende Beispiel zeigt einen Enum-Typ für eine Maustaste und Hilfetexte für die verschiedenen Elemente.

```
mbLeft = 0 [helpstring 'mbLeft'];
mbRight = 1 [helpstring 'mbRight'];
mbMiddle = 3 [helpstring 'mbMiddle'];
```

Enum-Elemente

Enum-Objekte (Aufzählungen) bestehen aus einer Liste von Konstanten, die numerisch sein müssen. Als Werte werden normalerweise dezimale oder hexadezimale Integer-Zahlen verwendet. Der Basiswert ist standardmäßig Null.

Um die Konstanten einer Aufzählung zu definieren, klicken Sie auf die Schaltfläche *Neu Const.*

Alias-Typinformationen

Mit einem Alias-Objekt erstellen Sie einen Alias (Typedefinition) für einen Typ. Sie können auf diese Weise Typen für die Verwendung in anderen Typinformationen wie Records oder Unions definieren.

Mit einer Alias-Definition können Sie im Typbibliothekseditor folgende Aktion durchführen:

- Attribute ändern

Die Registerkarte Attribute für Alias-Objekte

Die Registerkarte *Attribute* enthält für Alias-Objekte folgende Informationen::

Tabelle 50.18 Alias-Attribute

Attribut	Bedeutung
Name	Der Name des Alias-Objekts in der Typbibliothek.
GUID	Der global eindeutige, 128 Bit lange Bezeichner des Objekts (optional). Ohne dieses Attribut kann der Alias im System nicht eindeutig identifiziert werden.
Version	Die Version des Alias-Objekts (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer des Objekts. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Type	Geben Sie an, was Sie mit dem Aliasnamen versehen wollen.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.

Tabelle 50.18 Alias-Attribute (Fortsetzung)

Attribut	Bedeutung
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen..

Record-Typinformationen

Mit einer Record-Definition können Sie im Typbibliothekseditor folgende Aktionen durchführen:

- Attribute ändern
- Record-Elemente hinzufügen, entfernen oder ändern

Die Registerkarte Attribute für Record-Objekte

Die Registerkarte *Attribute* für Records enthält folgendes:

Tabelle 50.19 Record-Attribute

Attribut	Bedeutung
Name	Der Name des Record-Objekts in der Typbibliothek.
GUID	Der global eindeutige, 128 Bit lange Bezeichner des Objekts (optional). Ohne dieses Attribut kann der Record im System nicht eindeutig identifiziert werden.
Version	Die Version des Record-Objekts (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer des Objekts. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Record-Elemente

Ein Record besteht aus einer Liste von Strukturelementen oder Feldern. Für ein Feld können folgende Informationen angegeben werden:

- Name
- Typ

Sie können für die Elemente einen der vordefinierten Typen angeben oder mit Hilfe eines Alias-Objekts einen Typ definieren.

Records können mit einem optionalen Tag versehen werden.

Union-Typinformationen

Eine Union ist ein Record mit nur einem varianten Teil.

Mit einer Union-Definition können Sie im Typbibliothekseditor folgende Aktionen durchführen:

- Attribute ändern
- Union-Elemente hinzufügen, entfernen oder ändern

Die Registerkarte Attribute für Union-Objekte

Die Registerkarte *Attribute* enthält für Union-Objekte folgende Informationen::

Tabelle 50.20 Union-Attribute

Attribut	Bedeutung
Name	Der Name des Union-Objekts in der Typbibliothek.
GUID	Der global eindeutige, 128 Bit lange Bezeichner des Objekts (optional). Ohne dieses Attribut kann das Union-Objekt im System nicht eindeutig identifiziert werden.
Version	Die Version des Union-Objekts (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer des Objekts. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Union-Elemente

Ein Union-Objekt besteht wie ein Record aus einer Liste von Strukturelementen oder Feldern. Für ein Feld können folgende Informationen angegeben werden:

- Name
- Typ

Sie können für Elemente einen der vordefinierten Typen angeben oder mit Hilfe eines Alias-Objekts einen Typ definieren.

Unions können mit einem optionalen Tag definiert werden.

Modul-Typinformationen

Modul definieren eine Gruppe von Funktionen, normalerweise DLL-Einstiegspunkte.

Sie können eine Moduldefinition in der Typbibliothek erzeugen oder ändern, indem Sie:

- Attribute ändern
- Modulelemente hinzufügen, entfernen oder ändern

Hinweis Delphi generiert für das Modul nicht automatisch Deklarationen oder Implementierungen. Die angegebene DLL wird vom Benutzer als separates Projekt erstellt.

Die Registerkarte Attribute für Modul

Die Registerkarte *Attribute* enthält für ein Modul folgende Informationen:

Tabelle 50.21 Modulattribute:

Attribut	Bedeutung
Name	Der Name des Moduls in der Typbibliothek.
GUID	Der global eindeutige, 128 Bit lange Bezeichner des Objekts (optional). Ohne dieses Attribut kann das Modul im System nicht eindeutig identifiziert werden.
Version	Die Version des Moduls (falls mehrere Versionen vorhanden sind). Der Wert wird entweder in Form von zwei durch einen Punkt getrennten dezimalen Ganzzahlen (z. B. 1.0) oder als Ganzzahl (z. B. 1) angegeben. Die erste der beiden Zahlen entspricht der Haupt-, die zweite der Unterversionsnummer des Moduls. Wird nur eine Zahl verwendet, ist dies die Hauptversionsnummer. Beide Zahlen sind vorzeichenlose **short**-Werte zwischen 0 und 65535.
Name der DLL	Der Name der DLL, für die diese Einstiegspunkte verwendet werden.
Hilfe-String	Eine kurze Beschreibung des Elements. Wird mit *Hilfekontext* verwendet, um Hilfe in Form einer Hilfedatei bereitzustellen.

Tabelle 50.21 Modulattribute: (Fortsetzung)

Attribut	Bedeutung
Hilfekontext	Die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet.
Hilfe-String-Kontext	Bei Hilfe-DLLs ist dies die Hilfekontext-ID des Elements, die das Hilfethema für dieses Element innerhalb der Hilfedatei kennzeichnet. Wird mit *Hilfe-String-DLL* verwendet, um Hilfe als separate DLL bereitzustellen.

Modul-Elemente

Ein Modul kann folgende Elemente haben:

- Methoden
- Konstanten

Modulmethoden

Modulmethoden verfügen über folgende Attribute:

Tabelle 50.22 Modulmethoden-Attribute

Attribut	Bedeutung
Name	Der beschreibende Name des Moduls.
DLL-Einsprung	Einstiegspunkt in der zugeordneten DLL.

Parameter für Modulmethoden werden auf dieselbe Weise wie Schnittstellenparameter angegeben (siehe »Die Registerkarte Parameter für Eigenschaften und Methoden« auf Seite 50-13).

Modulkonstanten

Um die Konstanten eines Moduls zu definieren, geben Sie folgendes an:

- Name
- Wert
- Typ

Eine Modulkonstante kann je nach Attribut ein numerischer oder ein String-Wert sein. Numerische Werte werden normalerweise als dezimale oder hexadezimale Integer-Zahlen angegeben. Es können aber auch einzelne Zeichen (z. B. \0) verwendet werden. Strings müssen in Anführungszeichen (”...”) gesetzt werden und dürfen nicht mehrzeilig sein. Der umgekehrte Schrägstrich (\) dient dabei als Escape-Zeichen. Jedes auf ein \ folgende Zeichen wird nicht als Sonderzeichen, sondern literal interpretiert. Folgendermaßen können Sie beispielsweise einen umgekehrten Schrägstrich in den Text einfügen:

```
"Pfad: c:\\bin\
```

Typbibliotheken erstellen

Mit dem Typbibliothekseditor können Sie Typbibliotheken für ActiveX-Steuerelemente, ActiveX-Server und andere COM-Objekte erstellen.

Der Editor unterstützt in einer Typbibliothek eine Teilmenge von gültigen Typen und sicheren Arrays, wie nachfolgend beschrieben.

Dieser Abschnitt beschreibt folgende Vorgänge:

- Eine neue Typbibliothek erstellen
- Eine vorhandene Typbibliothek öffnen
- Eine Schnittstelle hinzufügen
- Eigenschaften und Methoden hinzufügen
- Ein CoClass-Objekt hinzufügen
- Eine Aufzählung zur Typbibliothek hinzufügen
- Typinformationen speichern und registrieren

Gültige Typen

Im Typbibliothekseditor werden verschiedene Typbezeichner verwendet, je nachdem, ob Sie mit IDL oder mit Object Pascal arbeiten. Geben Sie die Sprache, mit der Sie arbeiten möchten, im Dialogfeld *Umgebungsoptionen* an.

Die nachfolgenden Typen sind in einer Typbibliothek für die COM-Entwicklung verfügbar. Die Spalte *Automatisierungskompatibe*l zeigt an, ob der Typ von einer Schnittstelle verwendet werden kann, bei der die Flags Automation oder DispInterface markiert sind. Für diese Typen kann COM die Sequenzbildung über die Typbibliothek automatisch durchführen.

Tabelle 50.23Gültige Typen

Pascal-Typ	IDL-Typ	Variantentyp	Automatisierungs-kompatibel	Beschreibung
Smallint	short	VT_I2	Ja	Vorzeichenbehafteter ganzzahliger 2-Byte-Wert
Integer	long	VT_I4	Ja	Vorzeichenbehafteter ganzzahliger 4-Byte-Wert
Single	single	VT_R4	Ja	Reeller 4-Byte-Wert
Double	double	VT_R8	Ja	Reeller 8-Byte-Wert
Currency	CURRENCY	VT_CY	Ja	Währung
TDateTime	DATE	VT_DATE	Ja	Datum
WideString	BSTR	VT_BSTR	Ja	Binäre Zeichenkette
IDispatch	IDispatch	VT_DISPATCH	Ja	Zeiger auf *IDispatch*-Schnittstelle
SCODE	SCODE	VT_ERROR	Ja	OLE-Fehlercode

Tabelle 50.23Gültige Typen (Fortsetzung)

Pascal-Typ	IDL-Typ	Variantentyp	Automatisierungs-kompatibel	Beschreibung
WordBool	VARIANT_BOOL	VT_BOOL	Ja	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	Ja	OLE-Variante
IUnknown	IUnknown	VT_UNKNOWN	Ja	Zeiger auf *Iunknown*-Schnittstelle
Shortint	byte	VT_I1	Nein	Vorzeichenbehafteter ganzzahliger 1-Byte-Wert
Byte	unsigned char	VT_UI1	Ja	Vorzeichenloser ganzzahliger 1-Byte-Wert
Word	unsigned short	VT_UI2	Nein*	Vorzeichenloser ganzzahliger 2-Byte-Wert
LongWord	unsigned long	VT_UI4	Nein*	Vorzeichenloser ganzzahliger 4-Byte-Wert
Int64	__int64	VT_I8	Nein	Vorzeichenbehafteter reeller 8-Byte-Wert
Largeuint	uint64	VT_UI8	Nein	Vorzeichenloser reeller 8-Byte-Wert
SYSINT	int	VT_INT	Nein*	Systemabhängiger ganzzahliger Wert (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	Nein*	Systemabhängiger vorzeichenloser ganzzahliger Wert
HResult	HRESULT	VT_HRESULT	Nein	32-Bit-Fehlercode
Pointer		VT_PTR -> VT_VOID	Nein	Untypisierter Zeiger
SafeArray	SAFEARRAY	VT_SAFEARRAY	Nein	OLE sicheres Array (SafeArray)
PChar	LPSTR	VT_LPSTR	Nein	Zeiger auf **Char**
PWideChar	LPWSTR	VT_LPWSTR	Nein	Zeiger auf **WideChar**

* Word, LongWord, SYSINT und SYSUINT können bei einigen Anwendungen automatisierungskompatibel sein.

Hinweis Die gültigen Typen für die CORBA-Entwicklung werden in Kapitel 28, »CORBA-Anwendungen« beschrieben.

Hinweis Byte (VT_UI1) ist automatisierungskompatibel, jedoch in den Typen Variant und OleVariant nicht zulässig, da zahlreiche Automatisierungsserver diesen Wert nicht korrekt behandeln.

Zusätzlich zu diesen Typen können alle Schnittstellen und Typen, die in der Bibliothek oder in referenzierten Bibliotheken definiert sind, in einer Typbibliotheksdefinition verwendet werden.

Der Typbibliothekseditor speichert Typinformationen, die in der Syntax der IDL (Interface Definition Language) ausgedrückt sind, im Binärformat in der erzeugten Typbibliotheksdatei (TLB).

Falls dem Parametertyp ein Zeigertyp folgt, übersetzt der Typbibliothekseditor diesen Typ normalerweise in einen Variablenparameter. Beim Speichern der Typbibliothek werden die mit dem Variablenparameter verknüpften ElemDesc-IDL-Flags als IDL_FIN oder IDL_FOUT markiert.

Wenn dem Typ ein Zeiger folgt, werden ElemDesc-IDL-Flags oft nicht durch IDL_FIN oder IDL_FOUT markiert. Bei Dispatch-Schnittstellen werden OR-IDL-Flags normalerweise nicht verwendet. In diesen Fällen kann es vorkommen, daß Kommentare neben dem Variablenbezeichner, wie zum Beispiel {IDL_None} oder {IDL_In}, stehen. Diese Kommentare werden beim Speichern einer Typbibliothek verwendet, um die IDL-Flags korrekt zu markieren.

Sichere Arrays

COM verlangt, daß Arrays mittels eines speziellen Datentyps, nämlich als *SafeArray*, übergeben werden. Um das zu erreichen, können Sie *SafeArrays* durch den Aufruf bestimmter COM-Funktionen erstellen und wieder freigeben. Sämtliche Elemente eines *SafeArrays* müssen gültige automatisierungskompatible Typen sein. Der Delphi-Compiler ist in der Lage, *SafeArrays* von COM zu identifizieren und ruft automatisch die COM API auf, um *SafeArrays* zu erstellen, zu kopieren und wieder freizugeben.

Im Typbibliothekseditor muß ein *SafeArray* seinen Komponententyp angeben. So gibt beispielsweise das folgende *SafeArray* den Komponententyp *Integer* an:

```
procedure HighLightLines(Lines: SafeArray of Integer);
```

Der Komponententyp für ein *SafeArray* muß zur Automatisierung kompatibel sein. Im Übersetzungsprogramm für Typbibliotheken in Object Pascal ist der Komponententyp nicht zulässig.

Object-Pascal- oder IDL-Syntax verwenden

Per Vorgabe zeigt die Registerkarte *Text* des Typbibliothekseditors Ihre Typinformation unter Verwendung einer erweiterten Object-Pascal-Syntax an. Sie können jedoch auch direkt mit IDL arbeiten, indem Sie die entsprechende Einstellung im Dialogfeld *Umgebungsoptionen* ändern. Wählen Sie dazu *Tools* / *Umgebungsoptionen*, und wählen Sie auf der Registerkarte Typbibliothek als Sprache IDL aus.

Hinweis Die Auswahl der IDL- bzw. Object-Pascal-Syntax beeinflußt auch die verfügbaren Optionen auf der Registerkarte *Parameterattribute.*

Wie überall in Object-Pascal-Anwendungen, wird auch bei Bezeichnern in Typbibliotheken zwischen Groß- und Kleinschreibung nicht unterschieden. Bezeichner können bis zu 255 Zeichen lang sein und müssen mit einem Buchstaben oder einem Unterstrich (_) beginnen.

Attribute-Spezifikationen

Die Sprachdefinition von Object Pascal wurde erweitert, so daß es nun in Typbibliotheken möglich ist, Attribute-Spezifikationen einzufügen. Attribute-Spezifikationen

werden in eckigen Klammern eingefügt und durch Kommas getrennt. Jede Attribute-Spezifikation besteht aus einem Attributnamen, gefolgt von einem Wert (sofern erforderlich).

Die folgende Tabelle listet die Attributnamen und die entsprechenden Werte auf.

Tabelle 50.24 Attribute-Syntax

Attributname	Beispiel	Betrifft
aggregatable	[aggregatable]	Typinformation
appobject	[appobject]	CoClass-Typinformation
bindable	[bindable]	Elemente (außer CoClass-Elementen)
control	[control]	Typbibliothek, Typinformation
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	Alles
default	[default]	CoClass-Elemente
defaultbind	[defaultbind]	Elemente (außer CoClass-Elementen)
defaultcollection	[defaultcollection]	Elemente (außer CoClass-Elementen)
defaultvtbl	[defaultvtbl]	CoClass-Elemente
dispid	[dispid]	Elemente (außer CoClass-Elementen)
displaybind	[displaybind]	Elemente (außer CoClass-Elementen)
dllname	[dllname 'Helper.dll']	Modul-Typinformation
dual	[dual]	Schnittstellen-Typinformation
helpfile	[helpfile 'c:\hilfe\myhelp.hlp']	Typbibliothek
helpstringdll	[helpstringdll 'c:\hilfe\myhelp.dll']	Typbibliothek
helpcontext	[helpcontext 2005]	Alles (außer CoClass-Elementen und Parametern)
helpstring	[helpstring 'Gehaltslisten-Schnittstelle']	Alles (außer CoClass-Elementen und Parametern)
helpstringcontext	[helpstringcontext $17]	Alles (außer CoClass-Elementen und Parametern)
hidden	[hidden]	Alles (außer Parametern)
immediatebind	[immediatebind]	Alles (außer CoClass-Elementen)
lcid	[lcid $324]	Typbibliothek
licensed	[licensed]	Typbibliothek, CoClass-Typinformation
nonbrowsable	[nonbrowsable]	Alles (außer CoClass-Elementen)
nonextensible	[nonextensible]	Schnittstellen-Typinformation
oleautomation	[oleautomation]	Schnittstellen-Typinformation
predeclid	[predeclid]	Typinformation
propget	[propget]	Elemente (außer CoClass-Elementen)
propput	[propput]	Elemente (außer CoClass-Elementen)
propputref	[propputref]	Elemente (außer CoClass-Elementen)
public	[public]	Alias-Typinformation

Tabelle 50.24 Attribute-Syntax (Fortsetzung)

Attributname	Beispiel	Betrifft
readonly	[readonly]	Elemente (außer CoClass-Elementen)
replaceable	[replaceable]	Alles (außer CoClass-Elementen und Parametern)
requestedit	[requestedit]	Elemente (außer CoClass-Elementen)
restricted	[restricted]	Alles (außer Parametern)
source	[source]	Alle Elemente
uidefault	[uidefault]	Elemente (außer CoClass-Elementen)
usesgetlasterror	[usesgetlasterror]	Elemente (außer CoClass-Elementen)
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	Typbibliothek, Typinformation (erforderlich)
vararg	[vararg]	Elemente (außer CoClass-Elementen)
version	[version 1.1]	Typbibliothek, Typinformation

Schnittstellen-Syntax

Die Object-Pascal-Syntax für die Definition von Schnittstellen-Typinformation sieht wie folgt aus:

```
SchnittstellenName = interface[(Basisschnittstelle)] [Attribute]
Funktionsliste
[EigenschaftenMethodenListe]
end;
```

Beispielsweise deklariert der folgende Quelltextausschnitt eine Schnittstelle mit zwei Methoden und einer Eigenschaft:

```
Interface1 = interface (IDispatch)
  [uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
  function Calculate(optional seed:Integer=0): Integer;
  procedure Reset;
  procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
  function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

Die entsprechende IDL-Syntax lautet wie folgt:

```
[uuid ‘{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}’,version 1.0]
interface Interface1 :IDispatch
{
  long Calculate([in, optional, defaultvalue(0) ] long seed);
  void Reset(void);
  [propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
  [propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
 };
```

Syntax von Dispatch-Schnittstellen

Die Object-Pascal-Syntax für die Definition von Dispatch-Schnittstellen-Typinformation sieht wie folgt aus:

```
DispatchSchnittstellenName = dispinterface [Attribute]
Funktionsliste
[EigenschaftenListe]
end;
```

Beispielsweise deklariert der folgende Quelltextausschnitt eine Dispatch-Schnittstelle mit den gleichen Methoden und der gleichen Eigenschaft wie die vorherige Schnittstelle:

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
 version 1.0,
 helpstring 'dispatch interface for MyObj']
  function Calculate(seed:Integer): Integer [dispid 1];
  procedure Reset [dispid 2];
  property Range: Integer [dispid 3];
end;
```

Die äquivalente Syntax in IDL:

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
 version 1.0,
 helpstring "Dispatch-Schnittstelle für MyObj"]
dispinterface Interface1
{
methods:
  [id(1)] int Calculate([in] int seed);
  [id(2)] void Reset(void);
  properties:
  [id(3)] int Value;
 };
```

CoClass-Syntax

Die Object-Pascal-Syntax für die Deklaration von CoClass-Typinformation sieht wie folgt aus:

```
Klassenname = CoClass(Schnittstellenname[SchnittstellenAttribute], ...); [Attribute];
```

Zur Veranschaulichung wird im folgenden eine CoClass für die Schnittstelle *IMyInt* und die Dispatch-Schnittstelle *DMyInt* deklariert

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]
```

Die äquivalente Syntax in IDL:

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]
coclass myapp
{
  methods:
```

```
  [source] interface IMyInt);
  dispinterface DMyInt;
};
```

Enum-Syntax

Die Object-Pascal-Syntax für die Deklaration von Enum-Typinformation sieht wie folgt aus:

```
EnumName = ([Attribute] AufzaehlungsListe);
```

Dieser Quelltextausschnitt deklariert einen Aufzählungstyp mit drei Werten:

```
location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
            helpstring 'Standort des Messestands']
  Inside = 1 [helpstring 'Im Pavillon'];
  Outside = 2 [helpstring 'Vor dem Pavillon'];
  Offsite = 3 [helpstring 'Nicht in der Nähe des Pavillons'];);
```

Die entsprechende Syntax in IDL sieht folgendermaßen aus:

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring 'Standort des Messestands']
 typedef enum
  {
    [helpstring 'Im Pavillon'] Inside = 1,
    [helpstring 'Vor dem Pavillon'] Outside = 2,
    [helpstring 'Nicht in der Nähe des Pavillons'] Offsite = 3
   } location;
```

Alias-Syntax

Die Object-Pascal-Syntax für die Deklaration von Alias-Typinformation sieht wie folgt aus:

```
AliasName = BasisTyp[Attribute];
```

Das folgende Beispiel deklariert DWORD als einen Alias für Integer:

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

Die äquivalente Syntax in IDL lautet:

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Record-Syntax

Die Object-Pascal-Syntax für die Deklaration von Record-Typinformation sieht wie folgt aus:

```
RecordName = record [Attribute] Feldliste end;
```

Das folgende Beispiel deklariert ein Record:

```
Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
            helpstring 'Aufgabenbeschreibung']
  ID: Integer;
  StartDatum: TDate;
```

```
  EndDatum: TDate;
  BeauftragtePerson: WideString;
  Teilaufgaben: safearray of Integer;
end;
```

Die äquivalente Syntax in IDL lautet:

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring 'Aufgabenbeschreibung']
 typedef struct
  {
    long ID;
    DATE StartDatum;
    DATE EndDatum;
    BSTR BeauftragtePerson;
    SAFEARRAY (int) Teilaufgaben;
   } Tasks;
```

Union-Syntax

Die Object-Pascal-Syntax für die Deklaration von Union-Typinformation sieht wie folgt aus:

```
UnionName = record [Attribute]
case Integer of
  0: feld1;
  1: feld2;
  ...
 end;
```

Dieses Beispiel deklariert eine Union:

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
           helpstring 'Elementbeschreibung']
case Integer of
  0: (Name: WideString);
  1: (ID: Integer);
  3: (Wert: Double);
end;
```

Die äquivalente Syntax in IDL lautet:

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring 'Elementbeschreibung']
 typedef union
  {
    BSTR Name;
    long ID;
    double Wert;
  } MyUnion;
```

Modul-Syntax

Die Object-Pascal-Syntax für die Deklaration von Modul-Typinformation sieht wie folgt aus:

```
ModuleName = module Konstanten Einstiegspunkte end;
```

Im folgenden Quelltextausschnitt wird eine Typinformation für ein Modul deklariert:

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                   dllname 'circle.dll']
  PI: Double = 3.14159;
  function area(radius: Double): Double [ entry 1 ]; stdcall;
  function circumference(radius: Double): Double [ entry 2 ]; stdcall;
end;
```

Die äquivalente Syntax in IDL lautet:

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        dllname("circle.dll")]
 module MyModule
  {
    double PI = 3.14159;
    [entry(1)] double _stdcall area([in] double radius);
    [entry(2)] double _stdcall circumference([in] double radius);
  };
```

Eine neue Typbibliothek erstellen

Sie können auch Typbibliotheken erstellen, die unabhängig von einem ActiveX-Steuerelement sind, wenn Sie beispielsweise eine Bibliothek für eine ActiveX-Komponente definieren wollen, die noch nicht implementiert ist.

So erstellen Sie eine neue Typbibliothek:

1. Wählen Sie *Datei / Neu*, um das Dialogfeld *Objektgalerie* zu öffnen.
2. Wählen Sie die Registerkarte *ActiveX*.
3. Wählen Sie das Symbol *Typbibliothek*.
4. Bestätigen Sie mit *OK*.

   Der Typbibliothekseditor wird geöffnet, und Sie werden zur Eingabe eines Namens aufgefordert.
5. Geben Sie einen Namen für die Typbibliothek ein.

Eine vorhandene Typbibliothek öffnen

Wenn Sie mit Hilfe eines Experten ein ActiveX-Steuerelement, Automatisierungsobjekt, ActiveForm-Objekt, COM-Objekt, MTS-Objekt, Remote-Datenmodul oder MTS-Datenmodul erstellen, wird automatisch eine Typbibliothek mit einer Implementierungs-Unit erstellt.

So öffnen Sie eine vorhandene Typbibliothek unabhängig von einem Projekt:

1. Wählen Sie *Datei / Öffnen*, um das Dialogfeld *Öffnen* anzuzeigen.
2. Wählen Sie in der Dropdown-Liste *Dateityp* den Eintrag für Typbibliotheken, damit die verfügbaren Bibliotheken angezeigt werden.
3. Wählen Sie die gewünschte Typbibliothek aus.

4 Wählen Sie *Öffnen*.

So öffnen Sie eine dem aktuellen Projekt zugeordnete Typbibliothek:

1 Wählen Sie *Ansicht / Typbibliothek*.

Nun können Sie beliebig Schnittstellen, CoClasses und andere Elemente wie Aufzählungen, Eigenschaften und Methoden hinzufügen.

Hinweis Alle Änderungen, die Sie mit dem Typbibliothekseditor an einer Typbibliothek vornehmen, können sich automatisch auf das zugehörige ActiveX-Steuerelement auswirken. Wenn es Ihnen lieber ist, die Änderungen im vorhinein zu prüfen, stellen Sie sicher, daß das Dialogfeld *Aktualisierung durchführen* aktiviert ist. Dies ist auch voreingestellt. Mit Hilfe der Einstellung *Aktualisierte Einträge vor Neudarstellung anzeigen* auf der Registerkarte *Tools / Umgebungsoptionen / Typbibliothek* können Sie dies auch ändern. Weitere Informationen hierzu finden Sie unter »Aktualisierung durchführen (Dialogfeld)« auf Seite 50-37.

Eine Schnittstelle hinzufügen

So fügen Sie eine Schnittstelle hinzu:

1 Klicken Sie in der Werkzeugleiste auf das Schnittstellensymbol.

In die Objektliste wird eine Schnittstelle aufgenommen, und Sie werden zur Eingabe eines Namens aufgefordert.

2 Geben Sie einen Namen für die Schnittstelle ein.

Die neue Schnittstelle enthält Standardattribute, die Sie nach Bedarf ändern können. Sie können durch *get*- bzw. *set*-Funktionen und -Methoden repräsentierte Eigenschaften hinzufügen, um die Schnittstelle dem Einsatzzweck anzupassen.

Eigenschaften und Methoden einer Schnittstelle oder Dispatch-Schnittstelle hinzufügen

So fügen Sie einer Schnittstelle oder Dispatch-Schnittstelle Elemente hinzu:

1 Wählen Sie die Schnittstelle aus, und klicken Sie auf das Eigenschafts- oder Methodensymbol in der Werkzeugleiste.

In die Objektliste wird ein Schnittstellenelement aufgenommen, und Sie werden zur Eingabe eines Namens aufgefordert.

2 Geben Sie einen Namen für das Element ein.

Die Registerkarte *Attribute* enthält Standardwerte, die Sie beliebig ändern können.

Sie können Eigenschaften und Methoden auch hinzufügen, indem Sie die entsprechende IDL-Syntax direkt in die Registerkarte *Text eingeben*. Geben Sie beispielsweise folgende Eigenschaftsdeklarationen in die Registerkarte *Text* einer Schnittstelle ein:

```
property AutoSelect: WordBool; dispid 1;
property AutoSize: WordBool; dispid 2;
```

```
property BorderStyle: BorderStyle; dispid 3;
```

Nachdem Sie einer Schnittstelle Elemente hinzugefügt haben, werden diese als eigenständige Einträge in die Objektliste aufgenommen. Jedes Element verfügt über eine eigene Registerkarte *Attribute*, in der Sie Änderungen vornehmen können.

Wenn das Dialogfeld *Aktualisierung durchführen* aktiviert ist, erhalten Sie vom Typbibliothekseditor eine entsprechende Nachricht, bevor die Quelltextdateien beim Speichern der Typbibliothek aktualisiert werden. Sie werden außerdem auf potentielle Probleme hingewiesen. Benennen Sie beispielsweise versehentlich ein Ereignis um, wird in der Quelltextdatei eine Warnung angezeigt, die folgendermaßen lauten könnte:

```
Da sich in Ihrer Implementierungsdatei Instanzvariablen befinden, war
Delphi nicht in der Lage, die Datei zu aktualisieren, um die Änderung
des Ereignisnamens wiederzugeben. Sie müssen die Implementierungsdatei
von Hand aktualisieren.
```

Außerdem wird unmittelbar darüber eine Instruktion in die Datei eingefügt.

Hinweis Wenn Sie diese Warnung und die Instruktion ignorieren, kann der Quelltext nicht compiliert werden.

Ein CoClass-Objekt hinzufügen

So fügen Sie einer Typbibliothek ein CoClass-Objekt hinzu:

1. Wählen Sie in der Werkzeugleiste das CoClass-Symbol.

   In die Objektliste wird ein CoClass-Objekt aufgenommen, und Sie werden zur Eingabe eines Namens aufgefordert.

2. Geben Sie einen Namen für die Klasse ein.

   Die Registerkarte *Attribute* enthält Standardwerte, die Sie beliebig ändern können. Fügen Sie folgendermaßen Elemente hinzu:

3. Klicken Sie mit der rechten Maustaste die Registerkarte *Implementierung der Klasse* an, um eine Liste der verfügbaren Schnittstellen anzuzeigen.

   Die Liste enthält alle in der aktuellen und in den referenzierten Typbibliotheken definierten Schnittstellen.

4. Doppelklicken Sie auf die gewünschte Schnittstelle.

   Die Schnittstelle wird zusammen mit ihrer GUID und anderen Standardattributen in die Registerkarte eingefügt.

Eine Aufzählung zur Typbibliothek hinzufügen

So fügen Sie einer Typbibliothek eine Aufzählung (ein Enum-Objekt) hinzu:

1. Wählen Sie das Aufzählungssymbol in der Werkzeugleiste.

   In die Objektliste wird ein Enum-Objekt aufgenommen, und Sie werden zur Eingabe eines Namens aufgefordert.

2 Geben Sie einen Namen für das Element ein.

Die Registerkarte *Attribute* enthält Standardwerte, die Sie beliebig ändern können.

Über die Schaltfläche *Neu Const* fügen Sie der Aufzählung neue Werte hinzu. Wählen Sie anschließend jeden Aufzählungswert einzeln aus, und weisen Sie ihm die Attribute über die Registerkarte *Attribute* zu.

Typinformationen speichern und registrieren

Nach dem Ändern der Typbibliothek sollten Sie die Typinformationen speichern und registrieren. Wenn die Typbibliothek mit einem der ActiveX-Server-Projekttypen bzw. -objekte erstellt wurde, wird beim Speichern automatisch die binäre Typbibliothek, der ihren Inhalt repräsentierende Object-Pascal-Quelltext und der für die Typbibliothek verwaltete Implementierungsquelltext aktualisiert.

Der Typbibliothekseditor speichert Typinformationen in zwei Formaten:

- Als OLE-Verbunddokument mit dem Namen *Projekt.TLB*.
- Als Delphi-Unit.

Diese Unit besteht aus den Deklarationen der Elemente, die mit Object Pascal in der Typbibliothek definiert sind. Sie wird von Delphi verwendet, um die Typbibliothek als Ressource in die OCX- oder EXE-Datei einzubinden. Diese Dateien werden bei jedem Speichern der Typbibliothek im Typbibliothekseditor generiert.

Hinweis Die Typbibliothek wird als separate Binärdatei (.TLB) gespeichert und zudem in den Server gelinkt (EXE, DLL oder OCX).

Hinweis Wenn Sie den Typbibliothekseditor für CORBA-Schnittstellen verwenden, definiert diese Unit die Stub- und Skeleton-Objekte, die von der CORBA-Anwendung benötigt werden.

Im Typbibliothekseditor haben Sie folgende Möglichkeiten, Ihre Typinformationen zu speichern:

- Mit *Datei speichern* können Sie die TLB-Datei und die Delphi-Unit auf der Festplatte speichern. .
- Klicken Sie auf die Schaltfläche *Aktualisieren*, werden die Delphi-Units nur im Speicher aktualisiert.
- Wählen Sie die Schaltfläche *Registrieren*, wird die Typbibliothek in die Windows-Registrierung eingetragen. Dies erfolgt automatisch, sobald der Server registriert wird, dem die TLB-Datei zugeordnet ist. .
- Klicken Sie auf die Schaltfläche *Exportieren*, um eine IDL-Datei zu speichern, die die Typ- und Schnittstellendefinitionen in IDL-Syntax enthält.

In jedem Fall wird eine Syntaxprüfung durchgeführt. Wenn Sie die Typbibliothek aktualisieren, registrieren oder speichern, aktualisiert Delphi automatisch die Quelltextdatei des zugehörigen Objekts. Sie haben jedoch auch die Möglichkeit, die Änderungen vor der Übernahme zu prüfen. Dazu muß die Option *Aktualisierung durchführen* des Typbibliothekseditors aktiviert worden sein.

Aktualisierung durchführen (Dialogfeld)

Das Dialogfeld *Aktualisierung durchführen* wird angezeigt, wenn Sie die Typbibliothek aktualisieren, registrieren oder speichern, während die Option *Aktualisierte Einträge vor Neudarstellung anzeigen* in der Registerkarte *Tools / Umgebungsoptionen / Typbibliothek* aktiviert ist (Voreinstellung).

Ohne diese Option aktualisiert der Typbibliothekseditor automatisch die Quelltextdateien des zugehörigen Objekts, wenn Sie Änderungen im Editor vornehmen. Ist die Option dagegen aktiviert, können Sie die vorgeschlagenen Änderungen zurückweisen, wenn Sie die Typbibliothek aktualisieren, speichern oder registrieren.

Das Dialogfeld *Aktualisierung durchführen* weist auf mögliche Fehler hin und fügt entsprechende Instruktionen in die Quelltextdatei ein. Wenn Sie beispielsweise ein Ereignis versehentlich umbenennen, wird eine Warnung wie die folgende in die Quelltextdatei eingefügt:

```
Da sich in Ihrer Implementierungsdatei Instanzvariablen befinden, war
Delphi nicht in der Lage, die Datei zu aktualisieren, um die Änderung
des Ereignisnamens wiederzugeben. Sie müssen die Implementierungsdatei
von Hand aktualisieren.
```

Direkt über dieser Warnung wird eine entsprechende Instruktion in die Quelltextdatei eingefügt.

Hinweis Wenn Sie die Warnung und die Instruktion ignorieren, wird der Quelltext nicht compiliert.

Eine Typbibliothek speichern

Eine Typbibliothek speichern

- Syntax und Gültigkeit werden geprüft.
- Die Typinformationen werden in eine TLB-Datei geschrieben.
- Die Typinformationen werden in eine Delphi-Unit geschrieben.
- Die Modulverwaltung der IDE wird benachrichtigt, die Implementierung zu aktualisieren, wenn die Typbibliothek einem ActiveForm-Objekt, ActiveX-Steuerelement oder Automatisierungsobjekt zugeordnet ist.

Um die Typbibliothek zu speichern, wählen Sie aus dem Hauptmenü von Delphi *Datei / Speichern*.

Eine Typbibliothek aktualisieren

Beim Aktualisieren einer Typbibliothek werden folgende Operationen durchgeführt:

- Die Syntax wird geprüft.
- Die Delphi-Units werden nur im Speicher aktualisiert und nicht auf die Festplatte geschrieben.

- Die Modulverwaltung wird benachrichtigt, die Implementierung zu aktualisieren, wenn die Typbibliothek einem ActiveForm-Objekt, ActiveX-Steuerelement oder Automatisierungsobjekt zugeordnet ist.

Um die Typbibliothek zu aktualisieren, wählen Sie in der Werkzeugleiste das Symbol *Aktualisieren*.

Hinweis Wenn Elemente in der Typbibliothek umbenannt oder entfernt wurden, können durch das Aktualisieren der Implementierung doppelte Einträge entstehen. Sie müssen in diesem Fall Ihren Quelltext zu dem richtigen Eintrag verschieben und die Duplikate löschen.

Eine Typbibliothek registrieren

Beim Registrieren einer Typbibliothek werden folgende Operationen durchgeführt:

- Die Syntax wird geprüft.
- Der Windows-Registrierung wird ein Eintrag für die Typbibliothek hinzugefügt.

Um die Typbibliothek zu registrieren, wählen Sie in der Werkzeugleiste das Symbol *Registrieren*.

Eine IDL-Datei exportieren

Beim Exportieren einer Typbibliothek werden folgende Operationen durchgeführt:

- Es wird eine Syntaxprüfung durchgeführt.
- Eine IDL-Datei mit den Typinformationsdeklarationen wird angelegt. Diese Datei kann die Typinformation sowohl in CORBA-IDL als auch in Microsoft-IDL beschreiben.

Um die Typbibliothek zu exportieren klicken Sie in der Werkzeugleiste des Typbibliothekseditors auf das Symbol *Exportieren*.

Typbibliotheken weitergeben

Wenn eine als Teil eines ActiveX-Server-Projekts erstellte Typbibliothek vorliegt, wird sie standardmäßig automatisch als Ressource in die DLL-, OCX- oder EXE-Datei gelinkt.

Sie können die Typbibliothek jedoch zusammen mit Ihrer Anwendung als separate TLB-Datei weitergeben, da Delphi die Typbibliothek verwaltet.

Früher wurden Typbibliotheken für Automatisierungsanwendungen in getrennten Dateien mit der Namenserweiterung TLB gespeichert. Inzwischen werden bei Automatisierungsanwendungen die Typbibliotheken normalerweise direkt in die OCX- oder EXE-Datei compiliert. Das Betriebssystem erwartet die Typbibliothek als erste Ressource in der ausführbaren Datei (DLL, OCX oder EXE).

Wenn Sie eine Typbibliothek, bei der es sich nicht um die primäre Typbibliothek des Projekts handelt, anderen Anwendungsentwicklern zur Verfügung stellen wollen, kann sie eine der folgenden Formen haben:

- Ressource. Die Ressource sollte den Typ TYPELIB und eine Integer-ID haben. Soll die Typbibliothek mit einem Ressourcen-Compiler erzeugt werden, muß sie in der Ressourcendatei (RC) folgendermaßen deklariert sein:

  ```
  1 typelib mylib1.tlb
  2 typelib mylib2.tlb
  ```

  In einer ActiveX-Bibliotheksdatei können mehrere Typbibliothek-Ressourcen vorhanden sein. Die TLB-Datei kann mit Hilfe des Ressourcen-Compilers zu einer ActiveX-Bibliotheksdatei hinzugefügt werden.

- Eigenständige Binärdatei. Die TLB-Ausgabedatei des Typbibliothekseditors ist eine Binärdatei.

Kapitel

51

MTS-Objekte erstellen

MTS ist eine robuste Laufzeitumgebung, die verteilten COM-Anwendungen Transaktionsdienste, Sicherheit und Ressourcen-Pooling zur Verfügung stellt.

Delphi verfügt über einen MTS-Objekt-Experten, der ein MTS-Objekt erzeugt, so daß Sie Server-Komponenten erstellen können, welche die Vorteile der MTS-Umgebung nutzen können. MTS stellt zahlreiche Dienste zur Verfügung, die das Implementieren von COM-Clients und -Servern, insbesondere Remote-Servern, erleichtern.

MTS-Komponenten bieten eine Reihe von Low-level-Diensten. Dazu gehören:

- Verwalten von Systemressourcen, darunter Prozesse, Threads und Datenbankverbindungen, so daß eine Server-Anwendung mehrere Benutzer gleichzeitig behandeln kann.
- Automatisches Initiieren und Steuern von Transaktionen, um Ihre Anwendung zuverlässig zu machen.
- Erstellen, Ausführen und Löschen von Server-Komponenten, wenn erforderlich.
- Bereitstellung von rollenbasierter Sicherheit, so daß nur autorisierte Benutzer auf Ihre Anwendung zugreifen können.

Indem es Ihnen diese grundlegenden Dienste zur Verfügung stellt, ermöglicht Ihnen MTS die Konzentration auf die Entwicklung der Spezifika Ihrer jeweiligen verteilten Anwendung. Mit MTS implementieren Sie Ihre Businesslogik in MTS-Objekten oder in MTS-Remote-Datenmodulen. Beim Erstellen der Komponenten in Bibliotheken (DLLs) werden die DLLs in der MTS-Laufzeitumgebung installiert.

In Delphi können MTS-Clients Einzelanwendungen oder ActiveForms sein. In der MTS-Laufzeitumgebung können beliebige COM-Server laufen.

Dieses Kapitel bietet einen Überblick über die Technologie von Microsoft Transaction Server (MTS) und Informationen wie Sie es zum Schreiben von Anwendungen verwenden können, die auf MTS-Objekten basieren. Delphi unterstützt auch MTS-Remote-Datenmodule, die in Kapitel 14, »Mehrschichtige Anwendungen erstellen«, beschrieben werden.

Komponenten von Microsoft Transaction Server

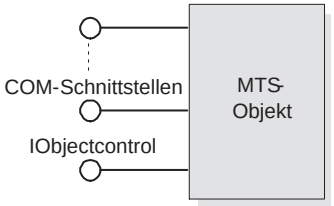
MTS-Komponenten sind in Dynamic Link Libraries (DLLs) enthaltene COM-In-Process-Server-Komponenten. Sie unterscheiden sich von anderen COM-Komponenten insofern, als sie in der MTS-Laufzeitumgebung ausgeführt werden. Sie können diese Komponenten mit Delphi oder einem beliebigen ActiveX-kompatiblen Werkzeug erstellen und implementieren.

Hinweis In der MTS-Terminologie bezeichnet Komponente den Code, der ein COM-Objekt implementiert. Beispielsweise werden MTS-Komponenten in Delphi als Klassen implementiert. Die Verwendung des Begriffs Komponente im Zusammenhang mit MTS entspricht nicht dem traditionellen Gebrauch dieses Begriffs in Delphi. Wir verwenden den Begriff Komponente, um eine Klasse oder ein Objekt zu bezeichnen, das aus der spezifischen Klasse, *TComponent*, abgeleitet wurde. Um jedoch die MTS-Terminologie einzuhalten, werden wie den Begriff MTS-Komponente verwenden, wenn speziell von MTS-Klassen die Rede ist. Sowohl in MTS als auch in Delphi verwenden wir den Begriff Objekt für eine Instanz einer MTS-Komponente.

Üblicherweise sind MTS-Server-Objekte klein und werden für in sich geschlossene Business-Funktionen verwendet. Beispielsweise können MTS-Komponenten die Business-Regeln einer Anwendung implementieren und Ansichten und Transformationen des Zustandes der Anwendung zur Verfügung stellen. Betrachten Sie als Beispiel eine medizinische Anwendung eines Arztes. In verschiedenen Datenbanken gespeicherte medizinische Aufzeichnungen repräsentieren den durchgängigen Zustand der medizinischen Anwendung, etwa die Krankheitsgeschichte eines Patienten. MTS-Komponenten aktualisieren diesen Zustand, um Veränderungen wie neue Patienten, Ergebnisse von Bluttests und Röntgenakten wiederzuspiegeln.

Wie in Abbildung 51.1 gezeigt, kann ein MTS-Objekt als beliebiges anderes COM-Objekt angesehen werden. Neben einer beliebigen Zahl von COM-Schnittstellen unterstützt es auch MTS-Schnittstellen. So wie *IUnknown* die allen COM-Objekten gemeinsame Schnittstelle ist, so ist *IObjectControl* allen MTS-Objekten gemeinsam. *IObjectControl* enthält Methoden zum Aktivieren und Deaktivieren des MTS-Objekts sowie zum Behandeln von Ressourcen wie Datenbankverbindungen.

Abbildung 51.1 MTS-Objektschnittstelle

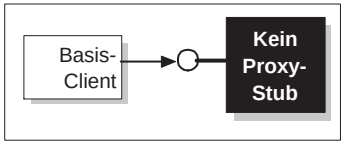


Der Client eines Servers innerhalb der MTS-Umgebung wird als *Basis-Client* bezeichnet. Aus Sicht eines Basis-Clients sieht ein COM-Objekt innerhalb der MTS-Umgebung aus wie jedes andere COM-Objekt. Das MTS-Objekt wird als DLL in der ausführbaren Datei von MTS installiert. Da MTS-Objekte zusammen mit der ausführbaren Datei von MTS ausgeführt werden, können sie Funktionen der MTS-

Laufzeitumgebung, wie Ressourcen-Pooling und Unterstützung von Transaktionen, nutzen.

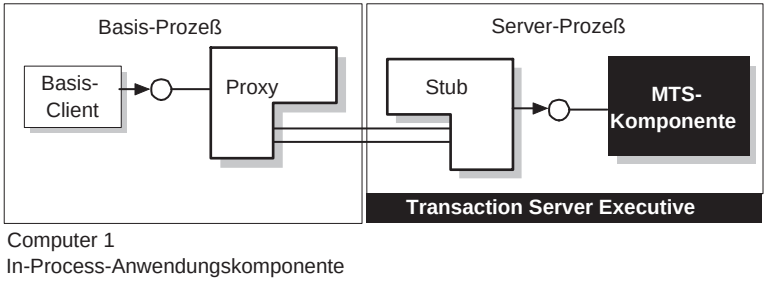
Die ausführbare Datei (.EXE) von MTS kann im selben Prozeß ablaufen wie der Basis-Client, wie in Abbildung 51.2 gezeigt.

Abbildung 51.2 MTS-In-Process-Komponente



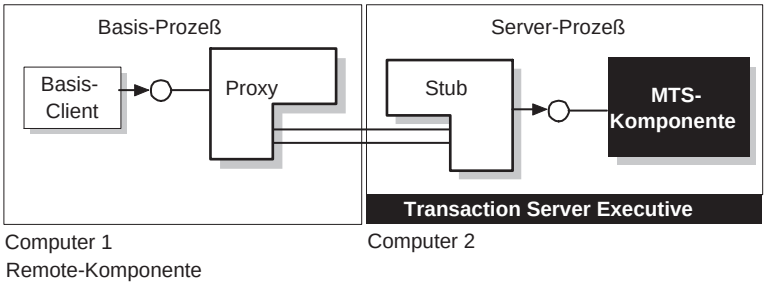
Die MTS-Komponente kann in einem Remote-Server-Prozeß auf derselben Maschine installiert werden, wie in Abbildung 51.3 gezeigt. Der Basis-Client kommuniziert mit einem Proxy, der die Anfrage des Clients an den Rumpf der MTS-Komponente weiterleitet, der wiederum über ihre Schnittstelle auf die MTS-Komponente zugreift.

Abbildung 51.3 Eine MTS-Komponente in einem Out-of-Process-Server



Die MTS-Komponente kann in einem Remote-Server-Prozeß auf einem separaten Rechner installiert werden, wie in Abbildung 51.4 dargestellt. Genau wie bei jedem anderen Server-Prozeß kommunizieren Client und Remote-Server zwischen Maschinen unter Verwendung von DCOM.

Abbildung 51.4 Eine MTS-Komponente in einem Remote-Server-Prozeß



Die Verbindungsinformation wird im MTS-Proxy gehalten. Die Verbindung zwischen MTS-Client und -Proxy bleibt so lange geöffnet, wie der Client eine Verbindung zum Server erfordert, so daß es dem Client erscheint, als habe er den Zugriff zum Server aufrecht erhalten. In Wahrheit aber kann der MTS-Stub das Objekt de- und reaktivieren, wodurch er Ressourcen einspart und es so anderen Clients ermöglicht, die Verbindung zu nutzen. Details über das Aktivieren und Deaktivieren finden Sie unter »Verwalten von Ressourcen durch Just-in-time-Aktivierung und Ressourcen-Pooling« auf Seite 51-5.

Anforderungen an eine MTS-Komponente

Über die Anforderungen an COM hinaus, macht es MTS erforderlich, daß es sich bei der Komponente um eine Dynamic Link Library (DLL) handelt. Komponenten, die als ausführbare (.EXE-) Dateien implementiert wurden, können in der MTS-Laufzeitumgebung nicht ausgeführt werden.

Darüber hinaus muß eine MTS-Komponente folgenden Anforderungen genügen:

- Wenn der MTS-Objektexperte verwendet wird, muß die Komponente über eine Standardklassen-Factory verfügen, die von Delphi automatisch bereitgestellt wird.
- Die Komponente muß ihr Klassenobjekt durch den Export der Standardmethode *DllGetClassObject* verfügbar machen.
- Alle Komponentenschnittstellen und Nebenklassen müssen durch eine Typbibliothek beschrieben werden, die vom MTS-Objektexperten bereitgestellt wird. Sie können der Typbibliothek Methoden und Eigenschaften hinzufügen, indem sie den Typbibliothekeditor verwenden. Die Typbibliothek wird vom MTS-Explorer verwendet, um ihr zur Laufzeit Informationen über installierte Komponenten zu entnehmen.
- Die Komponente muß nur solche Schnittstellen exportieren, die Standard-COM-Weiterleitung verwenden, was automatisch vom MTS-Objektexperten bereitgestellt wird.
- Die in Delphi implementierte MTS-Unterstützung erlaubt keine manuelle Sequenzbildung (Marshaling) für benutzerdefinierte Schnittstellen. Alle Schnittstellen müssen als duale Schnittstellen implementiert werden, die die automatische Sequenzbildung von COM unterstützen.
- Die Komponente muß die Funktion *DllRegisterServer* exportieren und in dieser Routine eine Selbstregistrierung ihres CLSID, ihres ProgID, ihrer Schnittstellen und ihrer Typbibliothek durchführen. Diese wird vom MTS-Objektexperten bereitgestellt.
- Eine im MTS-Prozeßraum ausgeführte Komponente kann nicht kombiniert werden mit Komponenten, die nicht in MTS ausgeführt werden.

Verwalten von Ressourcen durch Just-in-time-Aktivierung und Ressourcen-Pooling

MTS verwaltet Ressourcen durch Bereitstellung von

- Just-in-time-Aktivierung
- Ressourcen-Pooling
- Objekt-Pooling

Just-in-time-Aktivierung

Die Fähigkeit eines Objekts, de- und reaktiviert zu werden, während Clients Referenzen darauf halten, wird als *Just-in-time-Aktivierung* bezeichnet. Aus Sicht des Clients existiert zum Zeitpunkt ihrer Erstellung durch den Client bis zum Zeitpunkt ihrer letztendlichen Freigabe nur eine einzige Instanz des Objekts. Tatsächlich ist es möglich, daß das Objekt mehrmals de- und reaktiviert wurde. Indem Objekte deaktiviert werden, können Clients längere Zeit Referenzen auf das Objekt halten, ohne Systemressourcen zu beanspruchen. Wird ein Objekt deaktiviert, so gibt MTS alle Ressourcen des Objekts frei, beispielsweise seine Datenbankverbindung.

Wenn ein COM-Objekt als Teil der MTS-Umgebung erstellt wird, so wird außerdem ein zugehöriges Kontextobjekt angelegt. Dieses Kontextobjekt existiert während der gesamten Lebensdauer seines MTS-Objekts, über einen oder mehrere Reaktivierungszyklen hinweg. MTS verwendet den Objektkontext, um das Objekt während der Deaktivierung zu verfolgen. Dieses Kontextobjekt, auf das von der Schnittstelle *IObjectContext* zugegriffen wird, koordiniert Transaktionen. Ein COM-Objekt wird in deaktiviertem Zustand erstellt und wird aktiv, wenn es eine Client-Anfrage empfängt.

Ein MTS-Objekt wird deaktiviert, wenn einer der folgenden Fälle eintritt:

- **Das Objekt fordert mit *SetComplete oder SetAbort* eine Deaktivierung an**: Ein Objekt ruft die Methode *SetComplete* von *IObjectContext* auf, wenn es seine Aufgabe erfolgreich abgeschlossen hat und den internen Objektzustand nicht für den nächsten Aufruf durch den Client speichern muß. Ein Objekt ruft *SetAbort* auf, um anzugeben, daß es seine Aufgabe nicht erfolgreich abschließen kann und sein Objektstatus nicht gespeichert werden muß. Das bedeutet, daß der Status eines Objekts zu dem Status vor der Transaktion zurückkehrt. Oftmals können Objekte als *statuslos* bezeichnet werden, was bedeutet, daß sich dieses Objekt nach Rückkehr von jeder Methode deaktiviert.
- **Eine Transaktion wird festgeschrieben oder abgebrochen**: Wenn die Transaktion eines Objekts festgeschrieben oder abgebrochen wird, so wird das Objekt deaktiviert. Von diesen deaktivierten Objekten können nur solche fortbestehen, die von Clients außerhalb der Transaktion referenziert werden. Nachfolgende Aufrufe dieser Objekte reaktivieren diese und bewirken, daß sie in der nächsten Transaktion ausgeführt werden.

- **Der letzte Client gibt das Objekt frei:** Wenn ein Client das Objekt freigibt, so wird natürlich das Objekt deaktiviert und der Objektkontext gleichfalls freigegeben.

Ressourcen-Pooling

Da MTS untätige Systemressourcen bei einer Deaktivierung freigibt, werden die freigegebenen Ressourcen für andere Server-Objekte verfügbar. Das bedeutet, daß eine Datenbankverbindung, die nicht mehr von einem Server-Objekt beansprucht wird, von einem anderen Client wiederverwendet werden kann. Dies wird als *Ressourcen-Pooling* bezeichnet.

Das Öffnen und Schließen von Datenbankverbindungen kann zeitaufwendig sein. Deshalb setzt MTS Ressourcenspender ein, um eine Wiederverwendung bestehender Datenbankverbindungen zu ermöglichen, statt neue zu erstellen. Ein Ressourcenspender veranlaßt eine Zwischenspeicherung von Ressourcen, wie etwa Datenbanken, so daß die in einem Package befindlichen Komponenten Ressourcen gemeinsam nutzen können. Wenn Sie beispielsweise in einer Kundenbetreuungsanwendung eine Datenbanknachschlage- und eine Datenbankaktualisierungs-Komponente ausführen, so könnten Sie diese Komponenten in einem Package zusammenfassen, so daß sich diese Datenbankverbindungen gemeinsam nutzen ließen.

In der Delphi-Umgebung ist der Ressourcenspender die Borland Database Engine (BDE).

Beim Entwickeln von MTS-Anwendungen sind Sie für das Freigeben von Ressourcen selbst verantwortlich.

Ressourcen freigeben

Für die Freigabe der Ressourcen eines Objekts sind Sie verantwortlich. Dazu rufen Sie üblicherweise nach dem Abarbeiten jeder Client-Anforderung die Methoden *SetComplete* und *SetAbort* auf. Diese Methoden geben die vom MTS-Ressourcenspender belegten Ressourcen frei.

Gleichzeitig müssen Sie auch die Referenzen auf alle anderen Ressourcen freigeben, einschließlich die Referenzen auf andere Objekte (darunter MTS- und Kontextobjekte), und den von etwaigen Instanzen der Komponente belegten Speicher freigeben, wie etwa bei der Verwendung von **free** in Object Pascal.

Die einzige Situation, in der Sie diese Aufrufe nicht einschließen sollten, ist die, daß Sie den Status zwischen Client-Aufrufen beibehalten möchten. Details finden Sie unter »Statusbehaftete und statuslose Objekte« auf Seite 51-10.

Objekt-Pooling

So wie MTS dafür konzipiert ist, Ressourcen in einem Pool zu verwalten, so ist es auch dafür konzipiert, Objekte in einem Pool zu verwalten. Nachdem MTS die Methode *deactivate* aufruft, ruft es die Methode *CanBePooled* auf, die angibt, daß das Objekt für die Wiederverwendung in einem Pool gehalten werden kann. Ist *CanBePooled* auf *True* gesetzt, verschiebt MTS das Objekt in den Objekt-Pool, anstatt es nach der Deaktivierung zu zerstören. Objekte im Objekt-Pool sind zur sofortigen Verwendung für andere Clients verfügbar, die dieses Objekt anfordern. Nur wenn der Objekt-Pool leer ist, erstellt MTS eine neue Instanz des Objekts.

Objekte, die *False* zurückgeben oder die Schnittstelle *IObjectControl* nicht unterstützen, werden zerstört.

Hinweis In der gegenwärtigen Version von MTS ist das Verwalten von Objekten in Pools und ihre Wiederverwertung noch nicht verfügbar. MTS ruft zwar *CanBePooled* auf, wie beschrieben, doch findet keine Verwaltung in Pools statt. Diese Funktionalität wird zu Verfügung gestellt, um die Aufwärtskompatibilität zu gewährleisten. Entwickler können jetzt *CanBePooled* in ihren Anwendungen verwenden, so daß diese Anwendungen Pooling einsetzen können, sobald es verfügbar ist. Gegenwärtig initialisiert Delphi *CanBePooled* mit dem Wert *False*, da Objekt-Pooling in MTS noch nicht verfügbar ist.

Auf den Objektkontext zugreifen

Wie bei jedem COM-Objekt, so muß auch ein MTS verwendendes COM-Objekt bereits erstellt worden sein, bevor es verwendet wird. COM-Clients erstellen ein Objekt, indem sie die COM-Bibliotheksfunktion, *CoCreateInstance*, aufrufen.

Jedes in der MTS-Umgebung ausgeführte COM-Objekt muß ein zugehöriges Kontextobjekt besitzen. Dieses Kontextobjekt wird automatisch von MTS implementiert und wird verwendet, um die MTS-Komponente zu verwalten und Transaktionen zu koordinieren. Die Schnittstelle des Kontextobjekts ist *IObjectContext*. Um auf die meisten Methoden des Objektkontexts zuzugreifen, können Sie die Eigenschaft *ObjectContext* des Objekts *TMtsAutoObject* verwenden. Beispielsweise können Sie die Eigenschaft *ObjectContext* folgendermaßen verwenden:

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Ein anderer Weg des Zugriffs auf den Objektkontext besteht in der Verwendung der Methoden im Objekt *TMtsAutoObject*:

```
if IsCallerInRole ('Manager') ...
```

Sie können die beiden genannten Methoden verwenden. Die Verwendung der *TMtsAutoObject*-Methoden bietet jedoch einen kleinen Vorteil gegenüber der Eigenschaft *ObjectContext* beim Testen Ihrer Anwendung. Eine Erläuterung der Unterschiede finden Sie in »Fehlersuche und Testen von MTS-Objekten« auf Seite 51-24.

Unterstützung von Transaktionen in MTS

Die von MTS geleistete Unterstützung von Transaktionen ermöglicht es Ihnen, Aktionen zu Transaktionen zu gruppieren. Hätten Sie beispielsweise in einer Anwendung für medizinische Aufzeichnungen eine Transferkomponente zum Transferieren von Aufzeichnungen von einem Arzt zum anderen, so könnten Sie Ihre Methoden zum Hinzufügen und Löschen in derselben Transaktion zusammenfassen. Auf diese Weise funktioniert entweder der gesamte Transfer, oder er kann zu seinem vorherigen Status rückabgewickelt werden. Transaktionen vereinfachen die Behebung von Fehlern im Falle von Anwendungen, die auf *multiple* Datenbanken zugreifen müssen.

MTS-Transaktionen gewährleisten folgendes:

- Alle Aktualisierungen in einer einzelnen Transaktion werden entweder festgeschrieben oder abgebrochen und zu ihrem vorherigen Status rückabgewickelt. Dies wird als *Atomizität* bezeichnet.
- Eine Transaktion ist eine korrekte Transformation des Systemstatus, wobei die Statusinvarianten aufbewahrt werden. Dies wird als *Konsistenz* bezeichnet.
- Gleichzeitige Transaktionen sehen nicht die unvollständigen oder nicht festgeschriebenen Ergebnisse der anderen Seite, was zu Inkonsistenzen beim Status der Anwendung führen würde. Dies wird als *Isolierung* bezeichnet. Ressourcen-Manager verwenden transaktionsbasierte Synchronisierungsprotokolle, um die nicht festgeschriebene Arbeit aktiver Transaktionen zu isolieren.
- Festgeschriebene Aktualisierungen verwalteter Ressourcen (beispielsweise Datensätze) überstehen Ausfälle, einschließlich Kommunikationsausfällen, Prozeßausfällen und Ausfällen des Server-Systems. Dies wird als *Resistenz* bezeichnet. Das Protokollieren der Transaktionen ermöglicht es Ihnen, den Status der Resistenz nach einem Ausfall von Speichermedien wiederherzustellen.

Wenn Sie eine MTS-Komponente als Teil einer Transaktion deklarieren, so verknüpft MTS Transaktionen mit den Objekten der Komponente. Wird eine Methode eines Objekts ausgeführt, so werden Dienste, die Ressourcenmanager und Ressourcenspender seinetwegen ausführen, als Transaktion ausgeführt. Aufgaben aus mehreren Objekten können zu einer einzelnen Transaktion zusammengesetzt werden.

Transaktionsattribute

Jede MTS-Komponente besitzt ein Transaktionsattribut, das im MTS-Katalog verzeichnet ist. Dieser Katalog beinhaltet Konfigurationsinformationen für Komponenten, Packages und Rollen. Sie verwalten den Katalog mit Hilfe des MTS-Explorers, wie in »MTS-Objekte mit dem MTS-Explorer verwalten« auf Seite 51-25 beschrieben.

Jedes Transaktionsattribut kann auf folgende Einstellungen gesetzt werden:

Erfordert eine Transaktion	MTS-Objekte müssen *innerhalb des Gültigkeitsbereichs einer Transaktion* ausgeführt werden. Wird ein neues Objekt erstellt, so erbt sein Objektkontext die Transaktion vom Kontext des Clients. Besitzt der Client keinen Transaktionskontext, so erstellt MTS automatisch einen neuen Transaktionskontext für das Objekt.
Erfordert eine neue Transaktion	MTS-Objekte müssen *innerhalb ihrer eigenen Transaktionen* ausgeführt werden. Wird ein neues Objekt erstellt, so erstellt MTS automatisch eine neue Transaktion für das Objekt, unabhängig davon, ob sein Client eine Transaktion hat. Ein Objekt läuft niemals innerhalb des Gültigkeitsbereichs der Transaktion seines Clients. Statt dessen erzeugt das System immer unabhängige Transaktionen für die neuen Objekte.
Unterstützt Transaktionen	MTS-Objekte können *innerhalb des Gültigkeitsbereichs der Transaktionen ihrer Clients* ausgeführt werden. Wenn ein neues Objekt erstellt wird, erbt sein Objektkontext die Transaktion vom Kontext des Clients. Dies macht es möglich, daß mehrere Objekte zu einer einzigen Transaktion zusammengesetzt werden. Besitzt der Client keine Transaktion, so wird auch der neue Kontext ohne eine solche erzeugt.
Unterstützt keine Transaktionen	MTS-Objekte *laufen nicht innerhalb des Gültigkeitsbereichs von Transaktionen*. Wenn ein neues Objekt erstellt wird, so wird sein Objektkontext ohne Transaktion erstellt, unabhängig davon, ob der Client eine Transaktion besitzt. Verwenden Sie dies für COM-Objekte, die vor der MTS-Unterstützung entworfen wurden.

Der Objektkontext enthält das Transaktionsattribut

Das mit einem Objekt verknüpfte Kontextobjekt gibt an, ob das Objekt innerhalb einer Transaktion ausgeführt wird und, falls ja, die Identität der Transaktion.

Ressourcenspender können das Kontextobjekt verwenden, um dem MTS-Objekt transaktionsbasierte Dienste zur Verfügung zu stellen. Wenn beispielsweise ein Objekt, das innerhalb einer Transaktion abläuft, eine Datenbankverbindung unter Einsatz des BDE-Ressourcenspenders belegt, so wird die Verbindung automatisch in der Transaktion ausgeführt. Alle Datenbankaktualisierungen, die diese Verbindung verwenden, werden Teil der Transaktion und werden entweder festgeschrieben oder abgebrochen. Weitere Informationen finden Sie unter »Enlisting Resources in Transactions« in der Dokumentation zu MTS.

Statusbehaftete und statuslose Objekte

Wie jedes COM-Objekt können MTS-Objekte ihren internen Status über mehrere Interaktionen mit einem Client hinweg beibehalten. Ein solches Objekt wird als *statusbehaftet* bezeichnet. MTS-Objekte können auch *statuslos* sein, was bedeutet, daß sie keinen intermediären Status besitzen, während sie auf den nächsten Aufruf durch einen Client warten.

Wenn eine Transaktion festgeschrieben oder abgebrochen wird, werden alle daran beteiligten Objekte deaktiviert, was dazu führt, daß sie jeglichen im Verlauf der Transaktion angenommen Status verlieren. Dies hilft, die Isolierung der Transaktion und die Konsistenz der Datenbank sicherzustellen; es macht außerdem Server-Ressourcen frei für die Verwendung in anderen Transaktionen. Das Abschließen einer Transaktion ermöglicht es MTS, ein Objekt zu deaktivieren und seine Ressourcen zurückzuerlangen. Im folgenden Abschnittwird beschrieben, was zu tun ist, wenn MTS den von Ihnen festgelegten Objektstatus freigibt.

Das Aufrechterhalten des Objektstatus macht es erforderlich, daß das Objekt aktiviert bleibt, wodurch potentiell wertvolle Ressourcen wie Datenbankverbindungen aufrechterhalten bleiben.

Hinweis Statuslose Objekte sind effizienter und daher vorzuziehen.

Aktivieren multipler Objekte zum Unterstützen von Transaktionen

Sie verwenden die Methoden von *IObjectContext*, wie in der folgenden Tabelle gezeigt, um ein MTS-Objekt in die Lage zu versetzen, festzustellen, wie eine Transaktion abgeschlossen wird. Diese Methoden ermöglichen es Ihnen, zusammen mit dem Transaktionsattribut der Komponente, ein oder mehrere Objekte in einer einzigen Transaktion aufzuführen.

Tabelle 51.1 IobjectContext-Methoden zur Transaktionsunterstützung

Methode	Beschreibung
SetComplete	Gibt an, daß das Objekt seine Arbeit im Rahmen der Transaktion erfolgreich abgeschlossen hat. Das Objekt wird nach Rückkehr von derjenigen Methode deaktiviert, die als erste in den Kontext eingetreten ist. MTS reaktiviert das Objekt beim nächsten Aufruf, der die Ausführung eines Objekts erfordert.
SetAbort	Gibt an, daß die Arbeit des Objekts niemals festgeschrieben werden kann. Das Objekt wird nach Rückkehr von derjenigen Methode deaktiviert, die als erste in den Kontext eingetreten ist. MTS reaktiviert das Objekt beim nächsten Aufruf, der die Ausführung eines Objekts erfordert.

Tabelle 51.1 IobjectContext-Methoden zur Transaktionsunterstützung (Fortsetzung)

Methode	Beschreibung
EnableCommit	Gibt an, daß die Arbeit des Objekts nicht notwendigerweise erledigt ist, die Änderungen bezüglich der Transaktion jedoch in ihrer jetzigen Form festgeschrieben werden können. Verwenden Sie diese, um den Status eines Objekts über mehrere Aufrufe eines Clients beizubehalten. Wenn ein Objekt *EnableCommit* aufruft, so erlaubt es, daß die Transaktion, an der es teilnimmt festgeschrieben werden kann, behält jedoch seinen internen Status über Aufrufe durch seinen Client hinweg bei, bis es *SetComplete* oder *SetAbort* aufruft oder die Transaktion abgeschlossen ist. *EnableCommit* ist der vorgegebene Status, wenn ein Objekt aktiviert ist. Daher sollte ein Objekt immer *SetComplete* oder *SetAbort* aufrufen, bevor von einer Methode zurückgekeh*rt wird*, sofern Sie nicht wünschen, daß das Objekt seinen internen Status für den nächsten Aufruf durch einen Client beibehält.
DisableCommit	Gibt an, daß die Arbeit des Objekts inkonsistent ist und daß es seine Arbeit nicht abschließen kann, bis es weitere Methodenaufrufe vom Client erhält. Rufen Sie diese Methode vor Rückgabe der Kontrolle an den Client auf, um den Status über mehrere Client-Aufrufe hinweg beizubehalten. Dadurch wird die MTS-Laufzeitumgebung davon abgehalten, das Objekt zu deaktivieren und seine Ressourcen bei der Rückkehr von einem Methodenaufruf zurückzufordern. Hat ein Objekt *DisableCommit* aufgerufen, so wird die Transaktion abgebrochen, falls ein Client versucht, die Transaktion festzuschreiben, bevor das Objekt *EnableCommit* oder *SetComplete* aufgerufen hat. Sie können dies beispielsweise verwenden, um den vorgegebenen Status beim Aktivieren eines Objekts zu ändern.

MTS- oder client-gesteuerte Transaktionen

Transaktionen können entweder direkt vom Client oder automatisch durch die MTS-Laufzeitumgebung gesteuert werden.

Clients können direkte Kontrolle über Transaktionen haben, indem sie ein Transaktionskontextobjekt verwenden (mit Hilfe der Schnittstelle *ITransactionContext*). MTS ist jedoch dafür gedacht, die Client-Entwicklung zu vereinfachen, indem es automatisch die Transaktionsverwaltung besorgt.

MTS-Komponenten können so deklariert werden, daß ihre Objekte immer innerhalb einer Transaktion ausgeführt werden, unabhängig davon, wie die Objekte erzeugt werden. Auf diese Weise brauchen Objekte keine Logik zum Behandeln des Spezialfalls zu enthalten, daß ein Objekt von einem Client erzeugt wurde, der keine Transaktionen verwendet. Diese Funktion entlastet auch Client-Anwendungen. Clients müssen keine Transaktion initiieren, aus dem einfachen Grund, daß die von ihnen verwendete Komponente es erfordert.

Mit MTS-Transaktionen können Sie die Business-Logik Ihrer Anwendung in den Server-Objekten implementieren. Die Server-Objekte können die Regeln in Kraft setzen, so daß der Client nichts über sie wissen muß. Im Beispiel der medizinischen Anwendung für Ärzte kann ein Röntgentechnik-Client Röntgenbilder zu jeder beliebigen Aufzeichnung hinzufügen und sichten. Er muß nicht wissen, daß die Anwendung es

dem Röntgentechniker nicht gestattet, einen anderen Typ medizinischer Aufzeichnungen hinzuzufügen oder einzusehen. Diese Logik befindet sich in anderen Server-Objekten innerhalb der Anwendung.

Vorteile von Transaktionen

Daß eine Komponente entweder in ihrer eigenen Transaktion existieren oder Teil einer größeren Gruppe zu einer einzigen Transaktion gehörender Komponenten sein kann, ist ein wesentlicher Vorteil der MTS-Laufzeitumgebung. Das ermöglicht es einer Komponente, auf verschiedene Arten verwendet zu werden, so daß Anwendungsentwickler den Quelltext von Anwendungen wiederverwenden können, ohne die Anwendungslogik neu zu schreiben. Tatsächlich können Entwickler beim Packen ihrer Komponente festlegen, wie diese in Transaktionen verwendet werden. Sie können das Verhalten der Transaktion ändern, indem sie eine Komponente einer neuen Package hinzufügen. Details über das Zusammenfassen von Komponenten zu Packages finden Sie unter »MTS-Objekte in einem MTS-Package installieren« auf Seite 51-24.

Zeitüberschreitung bei Transaktionen

Das Zeitlimit einer Transaktion legt fest, wie lange (in Sekunden) eine Transaktion aktiv sein darf. Transaktionen, die nach Überschreiten des Zeitlimits noch existieren, werden automatisch vom System abgebrochen. Per Vorgabe liegt der Wert des Zeitlimits bei 60 Sekunden. Sie können Zeitlimits für Transaktionen deaktivieren, indem Sie einen Wert von 0 angeben, was bei der Fehlersuche bei MTS-Objekten nützlich ist.

So setzen Sie das Zeitlimit für Ihren Computer:

1. Wählen Sie im MTS-Explorer *Computer, My Computer*.

   Per Vorgabe entspricht *My Computer* dem lokalen Computer, auf dem MTS installiert ist.

2. Klicken Sie mit der rechten Maustaste, wählen Sie *Properties* und dann die Registerkarte *Options*.

   Die Registerkarte *Options* wird verwendet, um das Zeitlimit bei Transaktionen festzulegen.

3. Setzen Sie den Timeout-Wert auf 0. Damit gibt es für Transaktionen kein Zeitlimit mehr.

Klicken Sie auf *OK*, um die Einstellung zu speichern und zum MTS-Explorer zurückzukehren.Weitere Informationen über die Fehlersuche in MTS-Anwendungen finden Sie unter »Fehlersuche und Testen von MTS-Objekten« auf Seite 51-24.

Rollenbasierte Sicherheit

MTS unterstützt gegenwärtig rollenbasierte Sicherheit, bei der Sie einer logischen Benutzergruppe eine Rolle zuweisen. Beispielsweise könnte eine Anwendung für medizinische Informationen Rollen für Arzt, Röntgentechniker und Patient definieren.

Sie definieren die Autorisierung für jede Komponente und Komponentenschnittstelle durch Zuweisung von Rollen. Zum Beispiel könnte in der medizinischen Anwendung des Arztes allein der Arzt autorisiert sein, alle medizinischen Aufzeichnungen zu sichten; der Röntgentechniker dürfte nur Röntgenbilder sichten, und Patienten dürften nur ihre eigene medizinische Akte einsehen.

Üblicherweise definieren Sie Rollen während der Entwicklung der Anwendung und weisen Rollen für jedes Package von Komponenten zu. Diese Rollen werden dann bestimmten Benutzern zugewiesen, wenn die Anwendung eingesetzt wird. Administratoren können die Rollen mit Hilfe des MTS-Explorers konfigurieren.

Sie können Rollen auch programmiert mit Hilfe der Eigenschaft *TMtsAutoObject* von *ObjectContext* setzen. Beispiel:

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Ein weiterer Weg des Zugriffs auf den Objektkontext besteht in der Verwendung der Methoden des Objekts MTS *TMtsAutoObject*:

```
if IsCallerInRole ('Manager') ...
```

Hinweis Bei Anwendungen mit höheren Sicherheitsanforderungen implementieren Kontextobjekte die Schnittstelle *ISecurityProperty*. Deren Methoden geben Ihnen die Möglichkeit, auf den Sicherheitsbezeichner (security identifier, SID) von Windows zuzugreifen, und zwar sowohl auf den SID für den direkten Aufrufer und Erzeuger des Objekts als auch für die Clients, die dieses Objekt benutzen.

Ressourcenspender

Ein Ressourcenspender verwaltet den nicht resistenten gemeinsam benutzten Status bezüglich der Anwendungskomponenten innerhalb eines Prozesses. Ressourcenspender ähneln Ressourcenmanagern wie SQL Server, jedoch ohne garantierte Resistenz. Delphi stellt zwei Ressourcenspender zur Verfügung:

- BDE-Ressourcenspender
- Shared Property Manager

BDE-Ressourcenspender

Der BDE-Ressourcenspender verwaltet Pools von Datenbankverbindungen für MTS-Komponenten, welche die Standard-Datenbankschnittstellen verwenden, und reserviert so Verbindungen zu Objekten schnell und effizient. Für entfernte MTS-Datenmodule werden Verbindungen automatisch in den Transaktionen eines Objekts eingetragen, und der Ressourcenspender kann Verbindungen automatisch zurückfordern und wiederverwenden.

Shared Property Manager

Der Shared Property Manager ist ein Ressourcenspender, den Sie verwenden können, um den Status zwischen mehreren Objekten innerhalb eines Server-Prozesses gemeinsam zu nutzen. Beispielsweise können Sie ihn verwenden, um den gemeinsamen Status für ein Mehrbenutzerspiel beizubehalten.

Durch die Verwendungen des Shared Property Managers vermeiden Sie es, eine Menge Code zu Ihrer Anwendung hinzufügen zu müssen; MTS liefert die Unterstützung für Sie. Das heißt, der Shared Property Manager schützt den Objektstatus, indem er Schlösser und Semaphoren implementiert, um gemeinsam benutzte Eigenschaften vor gleichzeitigem Zugriff zu schützen. Der Shared Property Manager eliminiert Namenskonflikte, indem er *Gruppen gemeinsam benutzter Eigenschaften* bereitstellt, die einen eindeutigen Namensraum für die gemeinsam benutzten Eigenschaften herstellen, die sie enthalten.

Um die Ressource Shared Property Manager einsetzen zu können, verwenden Sie zunächst die Helferfunktion *CreateSharedPropertyGroup*, um eine Shared Property-Gruppe zu erstellen. Sodann können Sie alle Eigenschaften für diese Gruppe schreiben und alle Eigenschaften aus dieser Gruppe lesen. Durch die Verwendung einer Shared Property-Gruppe wird die Statusinformation über alle Deaktivierungen eines MTS-Objekts hinweg erhalten. Darüber hinaus können Statusinformationen unter allen MTS-Objekten gemeinsam verwendet werden, die in demselben Package installiert sind. Sie können MTS-Komponenten in einem Package installieren wie in »MTS-Objekte in einem MTS-Package installieren« auf Seite 51-24 beschrieben.

Folgendes Beispiel zeigt, wie Code zur Unterstützung des Shared Property Managers einem MTS-Objekt hinzugefügt werden kann. Nach dem Beispiel folgen Tips, die Sie beim Entwerfen Ihrer MTS-Anwendung zur gemeinsamen Verwendung von Eigenschaften befolgen sollten.

Beispiel: Gemeinsame Verwendung von Eigenschaften zwischen MTS-Objektinstanzen

Folgendes Beispiel erstellt eine Eigenschaftsgruppe namens *MyGroup*, die jene Eigenschaften enthalten soll, die zwischen Objekten und Objektinstanzen gemeinsam verwendet werden sollen. Dieses Beispiel enthält eine Zählereigenschaft, die gemeinsam benutzt wird. Es verwendet die Helferfunktion *CreateSharedPropertyGroup*, um den Eigenschaftsgruppenmanager und die Eigenschaftsgruppe zu erstellen, und verwen-

det dann die Methode *CreateProperty* des Gruppenobjekts, um eine Eigenschaft namens *Counter* zu erstellen.

Um den Wert einer Eigenschaft zu erhalten, verwenden Sie die Methode *PropertyByName* des Gruppenobjekts, wie unten gezeigt. Sie können auch die Methode *PropertyByPosition* verwenden.

```
unit Unit1;
interface
uses
  MtsObj, Mtx, ComObj, Project2_TLB;
type
  Tfoobar = class(TMtsAutoObject, Ifoobar)
  private
    Group: ISharedPropertyGroup;
  protected
    procedure OnActivate; override;
    procedure OnDeactivate; override;
    procedure IncCounter;
  end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
  Exists: WordBool;
  Counter: ISharedProperty;
begin
  Group := CreateSharedPropertyGroup('MyGroup');
  Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
  Counter: ISharedProperty;
begin
  Counter := Group.PropertyByName['Counter'];
  Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
  Group := nil;
end;
initialization
  TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance, tmApartment);
end.
```

Tips für die Verwendung des Shared Property Managers

Damit Objekte den Status gemeinsam verwenden, müssen alle im selben Server-Prozeß ablaufen.

Sie können gemeinsam benutzte Eigenschaften nur zur gemeinsamen Benutzung zwischen Objekten verwenden, die im selben Prozeß ablaufen. Wenn Sie wünschen, daß Instanzen aus verschiedenen Komponenten Eigenschaften gemeinsam benutzen,

so müssen Sie die Komponenten in demselben MTS-Package installieren. Da das Risiko besteht, daß Administratoren Komponenten aus einem Package in eine anderes verschieben, ist es am sichersten, die Größe einer Gruppe gemeinsam benutzter Eigenschaften auf Instanzen von Komponenten zu begrenzen, die in derselben DLL definiert sind.

Komponenten, die Eigenschaften gemeinsam benutzen, müssen dasselbe Aktivierungsattribut haben. Haben zwei Komponenten in demselben Package unterschiedliche Aktivierungsattribute, so werden sie im allgemeinen nicht in der Lage sein, Eigenschaften gemeinsam zu benutzen. Ist beispielsweise eine Komponente so konfiguriert, daß sie im Prozeß eines Clients abläuft und die andere zum Ablaufen in einem Server-Prozeß, so werden ihre Objekte üblicherweise in verschiedenen Prozessen ablaufen, selbst wenn sie sich in demselben Package befinden.

Basis-Clients und MTS-Komponenten

Es ist wichtig, den Unterschied zwischen Clients und Objekten in der MTS-Laufzeitumgebung zu verstehen. Clients, oder *Basis-Clients* in der MTS-Terminologie, laufen nicht unter MTS. Basis-Clients sind die Primärverbraucher von MTS-Objekten. Üblicherweise stellen sie die Benutzerschnittstelle der Anwendung zur Verfügung oder bilden Endbenutzeranfragen auf Geschäftsfunktionen ab, die in den MTS-Server-Objekten definiert wurden. Anders ausgedrückt können Clients nicht die zugrundeliegenden MTS-Funktionen nutzen. Clients erhalten weder Transaktionsunterstützung noch können sie auf Ressourcenspendern beruhen.

In der nachfolgenden Tabelle werden MTS-Komponenten und Basis-Client-Anwendungen einander gegenübergestellt.

Tabelle 51.2 MTS-Server-Objekte versus Basis-Clients

MTS-Komponenten	Basis-Clients
MTS-Komponenten sind in COM-Dynamic Link Libraries (DLLs) enthalten; MTS lädt DLLs auf Anforderung in Prozesse.	Basis-Clients können als ausführbare Dateien (EXE) oder Dynamic Link Libraries (DLL) geschrieben werden. MTS ist an ihrer Initialisierung oder ihrem Laden nicht beteiligt.
MTS verwaltet Server-Prozesse, die MTS-Komponenten beherbergen.	MTS verwaltet keine Basis-Client-Prozesse.
MTS erstellt und verwaltet die von Komponenten benutzten Threads.	MTS erstellt und verwaltet die von Basis-Client-Anwendungen benutzten Threads nicht.
Jedes MTS-Objekt besitzt ein assoziiertes Kontextobjekt. MTS erstellt und verwaltet Kontextobjekte automatisch und gibt sie frei.	Basis-Clients haben keine impliziten Kontextobjekte. Sie können Transaktionskontextobjekte verwenden, doch sie müssen diese explizit erstellen, verwalten und freigeben.
MTS-Objekte können Ressourcenspender verwenden. Ressourcenspender haben Zugriff auf das Kontextobjekt und gestatten es so, daß erhaltene Ressourcen automatisch mit dem Kontext assoziiert werden.	Basis-Clients können keine Ressourcenspender verwenden.

Zugrundeliegende Technologien von MTS, COM und DCOM

MTS verwendet das Component Object Model (COM) als Grundstein für die Unterstützung von COM-Objekten in Client-/Server-Anwendungen. COM definiert einen Satz strukturierter Schnittstellen, die Komponenten zur Kommunikation befähigen.

MTS verwendet DCOM für Fernübertragung. Um auf ein COM-Objekt auf einer anderen Maschine zuzugreifen, verwendet der Client DCOM, das eine lokale Objektanforderung transparent an das Remote-Objekt überträgt, das auf einer anderen Maschine läuft. Für Remote-Prozeduraufrufe verwendet DCOM das Protokoll RPC, das von der Distributed Computing Environment (DCE) von Open Group bereitgestellt wurde.

Für die Sicherheit verteilter Anwendungen verwendet DCOM das Sicherheitsprotokoll NT Lan Manager (NTLM). Für Verzeichnisdienste verwendet DCOM das Domain Name System (DNS).

Ressourcen-Pooling wird in der MTS-Umgebung allgemein von einer zugrundeliegenden Datenbankmaschine bereitgestellt. In Delphi wird Ressourcen-Pooling von der Borland Database Engine bereitgestellt. Alle Datenbankverbindungen, die von MTS-Objekten belegt werden, kommen aus diesem Pool. Diese Verbindungen können an einer zweiphasigen Festschreibung mit MTS als Steuerung teilnehmen.

Übersicht über die Erstellung von MTS-Objekten

Die Erstellung einer MTS-Komponente verläuft folgendermaßen:

1. Verwenden Sie den MTS-Objektexperten, um eine MTS-Komponente zu erstellen.
2. Fügen Sie der Anwendung Methoden und Eigenschaften hinzu, indem Sie den Typbibliothekseditor verwenden. Details über das Hinzufügen von Methoden und Eigenschaften unter Verwendung des Typbibliothekseditors entnehmen Sie Kapitel 50, »Mit Typbibliotheken arbeiten«.
3. Führen Sie für die MTS-Komponente die Fehlersuche und Tests aus.
4. Installieren Sie die MTS-Komponente in einem neuen oder bestehenden MTS-Package.
5. Verwalten Sie die MTS-Umgebung mit dem MTS-Explorer.

Den MTS-Objektexperten verwenden

Verwenden Sie den MTS-Objektexperten, um ein MTS-Objekt zu erstellen, das es Client-Anwendungen gestattet, innerhalb der MTS-Laufzeitumgebung auf Ihren Server zuzugreifen. MTS bietet umfangreiche Laufzeitunterstützung, wie Ressourcen-Pooling, Transaktionsverarbeitung und rollenbasierte Sicherheit.

So wird der MTS-Objektexperte aufgerufen:

1. Wählen Sie *Datei / Neu*.

2 Wählen Sie die mit *Multi-Tier* bezeichnete Registerkarte.

3 Doppelklicken Sie auf das Symbol *MTS-Objekt*.

Geben Sie im Experten folgendes an:

Klassenname	Geben Sie den Namen für die MTS-Klasse an.
Threading-Modell	Wählen Sie das Threading-Modell, um festzulegen, wie Client-Anwendungen die Schnittstelle Ihres Objekts aufrufen können. Dies ist das Threading-Modell, das Sie für die Implementierung im MTS-Objekt festlegen. Weitere Informationen über Threading-Modelle finden Sie in »Ein Threading-Modell auswählen« auf Seite 45-3. **Hinweis**: Das von Ihnen gewählte Threading-Modell legt fest, wie das Objekt registriert wird. Sie müssen sicherstellen, daß Ihre Objektimplementierung dem gewählten Modell entspricht.
Transaktionsmodell	Geben Sie an, ob und wie dieses MTS-Objekt Transaktionen unterstützt.
Ereignisunterstützung generieren	Markieren Sie dieses Feld, um dem Experten mitzuteilen, daß er eine separate Schnittstelle zum Verwalten von Ereignissen Ihres MTS-Objekts erstellen soll.

Wenn Sie diesen Vorgang abgeschlossen haben, wird dem aktuellen Projekt eine neue Unit hinzugefügt, welche die Definition für das MTS-Objekt enthält. Darüber hinaus fügt der Experte ein Typbibliotheksprojekt hinzu und öffnet die neue Typbibliothek. Nun können Sie über die Typbibliothek die Eigenschaften und Methoden der Schnittstelle zur Verfügung stellen. Die Bereitstellung der Schnittstelle erfolgt wie bei jedem Automatisierungsobjekt, wie in »Eigenschaften, Methoden und Ereignisse einer Anwendung für die Automatisierung bereitstellen« auf Seite 47-3 beschrieben.

Das MTS-Objekt implementiert eine *duale Schnittstelle*, die sowohl frühe Bindung (zum Zeitpunkt des Kompilierens) über die *vtable* als auch späte Bindung (Laufzeit) über die Schnittstelle *IDispatch* unterstützt.

Der MTS-Objektexperte implementiert die *IObjectControl*-Schnittstellenmethoden *Activate*, *Deactivate* und *CanBePooled*.

Auswahl eines Threading-Modells für ein MTS-Objekt

Die MTS-Laufzeitumgebung verwaltet Threads für Sie. MTS-Komponenten sollten keine Threads anlegen. Komponenten dürfen niemals einen Thread beenden, der in eine DLL hinein aufruft.

Wenn Sie im MTS-Experten ein Threading-Modell angeben, legen Sie fest, wie die Objekte für die Ausführung von Methoden Threads zugewiesen werden.

Tabelle 51.3 Threading-Modelle für COM-Objekte

Threading-Modell	Beschreibung	Vor- und Nachteile der Implementierung
Einzeln	Keine Thread-Unterstützung. Client-Anfragen werden durch den Aufrufsmechanismus serialisiert. Alle Objekte einer Einzel-Thread-Komponente werden im Haupt-Thread ausgeführt. Dies ist mit dem Vorgabe-Threading-Modell von COM kompatibel, das für Komponenten verwendet wird, die nicht über ein Threading Model Registry-Attribut verfügen oder für nicht-reentrante COM-Komponenten. Die Methodenausführung wird über alle Objekte in der Komponente und über alle Komponenten in einem Prozeß serialisiert.	Gestattet Komponenten die Verwendung nicht-reentranter Bibliotheken. Sehr begrenzte Skalierbarkeit. Statusbehaftete Einzel-Thread-Komponenten sind anfällig für Aufhängen. Sie können dieses Problem eliminieren, indem Sie statuslose Objekte verwenden und vor der Rückkehr von einer Methode *SetComplete* aufrufen.
Apartment (oder Einzel-Thread-Apartment)	Jedes Objekt wird einem Thread und einem Apartment zugewiesen, was für die Lebensdauer des Objekts gültig ist; es können jedoch Mehrfach-Threads für multiple Objekte verwendet werden. Dies ist ein Standard-COM-Konkurrenzmodell. Jedes Apartment ist mit einem bestimmten Thread verknüpft und besitzt eine Windows-Nachrichten-Pipe.	Bietet erhebliche Verbesserungen gegenüber dem Einzel-Thread-Modell. Zwei Objekte können gleichzeitig ausgeführt werden, solange sie sich in verschiedenen Aktivitäten befinden. Diese Objekte können sich in derselben oder in unterschiedlichen Komponenten befinden. Ähnlich einem COM-Apartment, außer daß die Objekte über mehrere Prozesse verteilt werden können.

Hinweis Diese Threading-Modelle ähneln den durch COM-Objekte definierten. Da jedoch die MTS-Umgebung mehr Unterstützung für Threads bereitstellt, unterscheidet sich hier die Bedeutung jedes Threading-Modells. Außerdem läßt sich das freie Threading-Modell aufgrund der MTS-Unterstützung von Aktivitäten nicht auf Objekte anwenden, die in der MTS-Umgebung ausgeführt werden.

MTS-Aktivitäten

MTS unterstützt *Aktivitäten*. Jedes MTS-Objekt gehört einer Aktivität an, was im Kontext des Objekts aufgezeichnet wird. Die Verknüpfung zwischen einem Objekt und einer Aktivität kann nicht geändert werden. Eine Aktivität schließt das vom Basis-Client erstellte MTS-Objekt ein, ebenso wie etwaige von diesem Objekt oder seinen Nachfolgern erstellte Objekte. Diese Objekte können über einen oder mehrere Prozesse verteilt sein und auf einem oder mehreren Computern ausgeführt werden.

Beispielsweise kann eine medizinische Anwendung eines Arztes ein MTS-Objekt enthalten, um an verschiedenen medizinischen Datenbanken Aktualisierungen durchzuführen und Aufzeichnungen daraus zu entfernen, die jeweils durch ein anderes Objekt repräsentiert werden. Das Objekt zum Hinzufügen kann auch andere Objekte verwenden, etwa ein Rezeptobjekt zum Aufzeichnen der Transaktion. Dies resultiert in verschiedenen MTS-Objekten, die entweder direkt oder indirekt vom Basis-Client gesteuert werden. Diese Objekte gehören alle der selben Aktivität an.

MTS verfolgt den Ausführungsablauf durch jede Aktivität und verhindert so, daß unachtsame Parallelaktionen den Anwendungsstatus verletzen. Diese Funktion resultiert in einem einzigen logischen Handlungsfaden über eine potentiell verteilte Ansammlung von Objekten hinweg. Durch das Vorliegen eines einzigen logischen Thread ist das Schreiben von Anwendungen erheblich leichter.

Wird ein MTS-Objekt aus einem bestehenden Kontext erstellt, der entweder ein Transaktionskontextobjekt oder einen MTS-Objektkontext verwendet, so wird das neue Objekt Bestandteil derselben Aktivität. In anderen Worten, der neue Kontext erbt den Aktivitätsbezeichner desjenigen Kontexts, der verwendet wurde, um ihn zu erstellen.

MTS gestattet nur einen einzigen logischen Ausführungs-Thread innerhalb einer Aktivität. Dies ähnelt dem Verhalten eines COM-Apartments, außer daß die Objekte über mehrere Prozesse verteilt sein können. Wenn ein Basis-Client in eine Aktivität hinein aufruft, so werden alle anderen Anforderungen für Arbeiten in der Aktivität (etwa von einem anderen Client-Thread) blockiert, bis der anfängliche Ausführungs-Thread an den Client zurückgeht.

Weitere Informationen über Threading in der MTS-Umgebung erhalten Sie, indem Sie in der MTS-Dokumentation nach dem Thema »Components and Threading« suchen.

Setzen des Transaktionsattributs

Ein Transaktionsattribut setzen Sie entweder zur Entwurfs- oder zur Laufzeit.

Zur Laufzeit fordert der MTS-Objektexperte Sie auf, das Transaktionsattribut zu wählen.

Sie können das Transaktionsattribut zur Laufzeit mit dem Typbibliothekeditor ändern.

So ändern Sie zur Laufzeit ein Transaktionsattribut:

1. Wählen Sie *Ansicht / Typbibliothek*, um den Typbibliothekeditor zu öffnen.
2. Wählen Sie die dem MTS-Objekt entsprechende Klasse.
3. Klicken Sie auf die Registerkarte *Transaktionen*, und wählen Sie das gewünschte Transaktionsattribut.

Hinweis: Ist das MTS-Objekt bereits in der Laufzeitumgebung installiert, so müssen Sie zunächst das Objekt deinstallieren und dann wieder installieren. Verwenden Sie hierzu *Start / MTS-Objekte installieren*.

Darüber hinaus können Sie das Transaktionsattribut eines in der MTS-Laufzeitumgebung installierten Objekts auch mit Hilfe des MTS-Explorers ändern.

Objektreferenzen übergeben

Sie können Objektreferenzen, etwa zur Verwendung als Callback, nur auf folgende Arten übergeben:

- Durch Rückgabe von einer Objekterstellungsschnittstelle, beispielsweise *CoCreateInstance* (oder ihr Äquivalent), *ITransactionContext.CreateInstance* oder *IObjectContext.CreateInstance*.
- Durch einen Aufruf von *QueryInterface*.
- Durch eine Methode, die *SafeRef* aufgerufen hat, um die Objektreferenz zu erhalten.

Eine in der oben angegebene Weise erhaltene Objektreferenz ist eine *sichere Referenz*. MTS stellt sicher, daß Methoden, die unter Verwendung sicherer Referenzen aufgerufen wurden, innerhalb des korrekten Kontexts ausgeführt werden.

Aufrufe, die sichere Referenzen verwenden, verlaufen immer über die MTS-Laufzeitumgebung. Dies ermöglicht es MTS, Kontextschalter zu verwenden, und gibt MTS-Objekten eine von Client-Referenzen unabhängige Lebensdauer.

Die Methode SafeRef verwenden

Ein Objekt kann die Funktion *SafeRef* verwenden, um eine Referenz auf sich selbst zu erhalten, die außerhalb ihres Kontexts sicher übergeben werden kann.

Die Unit, in der die Funktion *SafeRef* definiert ist, ist *Mtx*.

SafeRef benötigt als Eingabe

- eine Referenz auf die Schnittstellen-ID (RIID) der Schnittstelle, die das aktuelle Objekt einem anderen Objekt oder Client übergeben will;
- eine Referenz auf die Schnittstelle *IUnknown* des aktuellen Objekts.

SafeRef gibt einen Zeiger auf die im Parameter RIID bezeichnete Schnittstelle zurück, der sicher außerhalb des Kontexts des aktuellen Objekts übergeben werden kann. Es gibt **nil** zurück, falls das Objekt eine sichere Referenz auf ein anderes Objekt als sich selbst anfordert oder die im Parameter RIID bezeichnete Schnittstelle nicht implementiert ist.

Wenn ein MTS-Objekt eine Selbstreferenz an einen Client oder ein anderes Objekt (beispielsweise zur Verwendung als Callback) übergeben möchte, sollte es immer zunächst *SafeRef* aufrufen und dann die von diesem Aufruf übergebene Referenz übergeben. Ein Objekt sollte einem Client oder einem beliebigen anderen Objekt niemals einen **self**-Zeiger oder eine über einen internen Aufruf von *QueryInterface* erhaltene Selbstreferenz übergeben. Wird eine soche Referenz außerhalb des Kontexts des Objekts übergeben, ist sie keine gültige Referenz mehr.

Der Aufruf von *SafeRef* für eine bereits sichere Referenz gibt die sichere Referenz unverändert zurück, außer daß der Referenzzähler der Schnittstelle heraufgezählt wird.

Ruft ein Client *QueryInterface* für eine sichere Referenz auf, so stellt MTS automatisch sicher, daß die an den Client zurückgegebene Referenz sicher ist.

Ein Objekt, daß eine sichere Referenz enthält, muß diese freigeben, wenn es damit fertig ist.

Um Details über *SafeRef* zu erhalten, sehen Sie in der MTS-Dokumentation unter dem Stichwort »SafeRef« nach.

Callbacks

Objekte können Callbacks an Clients und an andere MTS-Komponenten tätigen. Beispielsweise können Sie ein Objekt haben, das ein anderes Objekt erzeugt. Das erzeugende Objekt kann eine Referenz auf sich selbst an das erstellte Objekt ergeben; das erstellte Objekt kann dann diese Referenz verwenden, um das erstellende Objekt aufzurufen.

Wenn Sie sich entscheiden, Callbacks zu verwenden, beachten Sie folgende Einschränkungen:

- Ein Callback an den Basis-Client oder ein anderes Package erfordert Zugriffsebenensicherheit für den Client. Zusätzlich muß der Client ein DCOM-Server sein.
- Intervenierende Firewalls können Rückrufe an den Client blockieren.
- Arbeiten, die auf dem Callback ausgeführt werden, laufen in der Umgebung des aufgerufenden Objekts ab. Es kann Teil derselben oder einer anderen Transaktion sein, oder es ist nicht Teil einer Transaktion.
- Das erstellende Objekt muß *SafeRef* aufrufen und die erhaltene Referenz dem erstellten Objekt übergeben, um einen Rückruf auf sich selbst auszuführen.

Ein Transaktionsobjekt auf der Client-Seite einrichten

Eine Client-Basisanwendung kann den Transaktionskontext über die Schnittstelle *ITransactionContextEx* steuern. Das folgende Quelltextbeispiel zeigt, wie eine Client-Anwendung *CreateTransactionContextEx* zum Erstellen des Transaktionskontexts verwendet. Diese Methode gibt eine Schnittstelle auf dieses Objekt zurück.

Das Beispiel setzt den Aufruf zum Transaktionskontext zwischen Aufrufe an *OleCheck*, was erforderlich ist, da die Methode *CreateInstance* nicht als **safecall** deklariert ist.

```
procedure TForm1.MoveMoneyClick(Sender: TObject);
begin
  Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGuid; Amount: Currency);
var
  TransactionContextEx: ITransactionContextEx;
```

```
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  TransactionContextEx := CreateTransactionContextEx;
  try
    OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    TransactionContextEx.Abort;
    raise;
  end;
  TransactionContextEx.Commit;
end;
```

Ein Transaktionsobjekt auf der Server-Seite einrichten

Um den Transaktionskontext von Seiten des MTS-Servers zu kontrollieren, erstellen Sie eine Instanz von *ObjectContext*. Im folgenden Beispiel befindet sich die Transfermethode im MTS-Objekt. Indem *ObjectContext* auf diese Weise verwendet wird, erbt die zu erstellende Instanz des Objekts alle Transaktionsattribute des erstellenden Objekts. Der Aufruf wird zwischen Aufrufe von *OleCheck* gesetzt, da die Methode *CreateInstance* nicht als **safecall** deklariert ist.

```
procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGuid;
 Amount: Currency);
var
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  try
    OleCheck(ObjectContext.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(ObjectContext.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    DisableCommit;
    raise;
  end;
  EnableCommit;
end;
```

Fehlersuche und Testen von MTS-Objekten

Sie können eine Fehlersuche für lokale und Remote-MTS-Objekte durchführen. Bei der Fehlersuche für MTS-Objekte sollten Sie Zeitlimits für Transaktion ausschalten.

Das Transaktionszeitlimit gibt an, wie lange (in Sekunden) eine Transaktion aktiv bleiben darf. Transaktionen, die nach Überschreitung des Zeitlimits noch aktiv sind, werden vom System automatisch abgebrochen. Per Vorgabe beträgt das Zeitlimit 60 Sekunden. Sie können Zeitlimits für Transaktionen deaktivieren, indem Sie den Wert 0 angeben, was bei der Fehlersuche in MTS-Objekten sinnvoll ist.

Um Informationen über Remote-Fehlersuche zu erhalten, sehen Sie in der Online-Hilfe unter dem Stichwort »Remote-Fehlersuche« nach.

Beim Testen des MTS-Objekts sollten Sie Ihr Objekt zunächst außerhalb der MTS-Umgebung testen, um Ihre Testumgebung zu vereinfachen.

Beim Entwickeln eines MTS-Servers können Sie einen Server nicht neu compilieren, wenn er sich noch im Speicher befindet. Sie würden einen Compilier-Fehler erhalten. Um dies zu vermeiden, können Sie die Eigenschaften des Package im MTS-Explorer so einstellen, daß der Server heruntergefahren wird, wenn er untätig ist.

So können Sie den MTS-Server bei Untätigkeit herunterfahren:

1. Klicken Sie im MTS-Explorer mit der rechten Maustaste das Package an, in dem Ihre MTS-Komponente installiert ist, und wählen Sie *Eigenschaften*.
2. Wählen Sie die Registerkarte *Erweitert*.

   Die Registerkarte *Erweitert* gibt an, ob der mit einem Package verknüpfte Server-Prozeß immer läuft oder nach einer gewissen Zeit heruntergefahren wird.
3. Setzen Sie das Zeitintervall auf 0, was den Server herunterfährt, sobald er keinem Client mehr dienen muß.
4. Klicken Sie auf *OK*, um die Einstellung zu speichern und zum MTS-Explorer zurückzukehren.

Achten Sie darauf, daß Sie beim Testen außerhalb der MTS-Umgebung die *Object-Property* von *TMtsObject* nicht direkt referenzieren. *TMtsObject* implementiert Methoden wie *SetComplete* und *SetAbort*, die sicher aufgerufen werden können, wenn der Objektkontext **nil** ist.

MTS-Objekte in einem MTS-Package installieren

MTS-Anwendungen bestehen aus einer Gruppe von In-Process-MTS-Objekten (oder MTS-Remote-Datenmodulen), die in einer einzelnen Instanz der ausführbaren Datei (EXE) von MTS ablaufen. Eine Gruppe von COM-Objekten, die alle im selben Prozeß ablaufen, wird als *Package* bezeichnet. Auf einer einzelnen Maschine können mehrere verschiedene Packages ablaufen, wobei jedes Package innerhalb einer separaten MTS EXE abläuft.

Sie können die Komponenten Ihrer Anwendung in einem einzigen Package gruppieren, so daß diese innerhalb eines einzelnen Prozesses ablaufen. Sinnvollerweise verteilen Sie Ihre Komponenten über verschiedene Packages, um Ihre Anwendung über mehrere Prozesse oder Maschinen zu partitionieren.

So installieren Sie MTS-Objekte in einem Package:

1. Wählen Sie *Start / MTS Objekte installieren*, um MTS-Objekte in einem Package zu installieren.
2. Markieren Sie die zu installierenden MTS-Objekte.
3. Wählen Sie im Dialogfeld *Objekt installieren* das Feld *In neues Package*, um ein neues Package zu erstellen, in dem das MTS-Objekt installiert werden soll, oder wählen Sie *In existierendes Package*, um das Objekt in einem der bestehenden aufgeführten MTS-Packages zu installieren.
4. Wählen Sie *OK*, um den MTS-Katalog zu aktualisieren, wodurch die Objekte zur Laufzeit verfügbar werden.

Packages können Komponenten aus mehreren DLLs enthalten, und Komponenten aus einer einzelnen DLL können in verschiedenen Packages installiert werden. Eine einzelne Komponente kann jedoch nicht auf verschiedene Packages verteilt werden.

MTS-Objekte mit dem MTS-Explorer verwalten

Haben Sie MTS-Objekte in einer MTS-Laufzeitumgebung installiert, so können sie diese Laufzeitobjekte mit Hilfe des MTS-Explorers verwalten. Der MTS-Explorer ist eine grafische Benutzerschnittstelle zum Verwalten und Einsetzen von MTS-Komponenten. Mit dem MTS-Explorer können Sie

- MTS-Objekte, Packages und Rollen konfigurieren;
- Eigenschaften von Komponenten in einem Package einsehen und die auf einem Computer installierten Packages anzeigen;
- Transaktionen für MTS-Komponenten überwachen und verwalten;
- Packages zwischen Computern verschieben;
- Ein Remote-MTS-Objekt einem lokalen Client verfügbar machen.

Details über den MTS-Explorer finden Sie im MTS-Administratorenhandbuch.

Die MTS-Dokumentation

Einzelheiten zu MTS-Konzepten, Programmierszenarien und Administrationswerkzeugen lassen sich der MTS-Dokumentation von Microsoft entnehmen. Diese Dokumentation dürfte denen hilfreich sein, für die das Entwickeln von MTS-Anwendungen etwas Neues ist.

Hier ist eine Übersicht der das Microsoft-Produkt begleitenden MTS-Dokumentation.

Tabelle 51.4 Übersicht der MTS-Dokumentation von Microsoft

Buch	Beschreibung
Setting Up MTS	Beschreibt, wie MTS und seine Komponenten einzurichten sind, einschließlich Anweisungen zum Zugriff auf Oracle-Datenbanken aus MTS-Anwendungen und zum Installieren von MTS-Beispielanwendungen.
Getting Stated with MTS	Gibt eine Übersicht neuer Funktionen in MTS, gibt einen kurzen Überblick über die Dokumentation und enthält ein Glossar.
Quick Tour of MTS	Gibt eine Übersicht zu MTS.
MTS Administrator's Guide	
Roadmap to the MTS Administrator's Guide	Beschreibt die verschiedenen Möglichkeiten der Verwendung des MTS-Explorers zum Einsatz und zur Administration von Anwendungen und gibt eine Übersicht über die grafische Schnittstelle MTS-Explorer.
Creating MTS Packages	Gibt eine aufgabenorientierte Dokumentation zum Erstellen und Zusammensetzen von MTS-Packages.
Distributing MTS Packages	Gibt eine aufgabenorientierte Dokumentation zum Verteilen von MTS-Packages.
Installing MTS Packages	Gibt eine aufgabenorientierte Dokumentation zum Installieren und Konfigurieren von MTS-Packages.
Maintaining MTS Packages	Gibt aufgabenorientierte Informationen zum Warten und Überwachen von MTS-Packages.
Managing MTS Transactions	Beschreibt verteilte Transaktionen und die Verwaltung von Transaktionen mit dem MTS-Explorer.
Automating MTS Administration	Gibt eine konzeptuelle Übersicht, Prozeduren und Beispielquelltext, die erläutern, wie skriptfähige MTS-Objekte zum Automatisieren von Verfahren im MTS-Explorer eingesetzt werden können.
MTS Programmer's Guide	
Overwiev and Concepts	Gibt eine Übersicht über das Produkt und die Zusammenarbeit der Produktkomponenten, erläutert, wie MTS den Bedürfnissen von Client-/Server-Entwicklern und Systemadministratoren entgegenkommt und enthält eine weitgehende Beschreibung von Programmierkonzepten für MTS-Komponenten.
Building Applications for MTS	Gibt aufgabenorientierte Informationen zum Entwickeln von ActiveX-Komponenten für MTS.
MTS Administrative Reference	Gibt eine Referenz zur Verwendung skriptfähiger MTS-Objekte zum Automatisieren von Verfahren im MTS-Explorer.
MTS Reference	Gibt eine Referenz für das MTS-Application Programming Interface (API).

Index

Symbole

Zahlen

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z