Von der Idee zur Delphi-Anwendung

3

Kapitelüberblick

Nachdem Sie jetzt die Entwicklungsumgebung und die Sprache von Delphi kennengelernt haben, kommen wir zur Anwendung dieser Kenntnisse. In diesem Kapitel wollen wir anhand einer einfachen Anwendung den Entwicklungszyklus einer Delphi-Anwendung nachvollziehen, ohne dabei allerdings besonders auf die Theorie der Softwareentwicklung einzugehen. Dabei lernen Sie auch die verschiedenen Techniken von Programmtest und Fehlersuche kennen, die in Delphi implementiert sind.

Außerdem lernen Sie in diesem Kapitel auch einige der wichtigsten visuellen Komponenten von Delphi kennen, die Sie in fast jeder Anwendung benötigen werden. Das Kapitel schließt damit den Einführungsteil ab, der besonders für Delphi-Anfänger interessant ist. Die darauffolgenden Kapitel behandeln dann verschiedene Aspekte der Programmierung mit Delphi etwas genauer.

3.1 In die erste Runde

Jede größere Anwendung hat normalerweise mehrere Entwicklungszyklen, in denen sie immer weiter vervollständigt wird. Unsere Beispielanwendung ist da keine Ausnahme, allerdings begnügen wir uns hier mit zwei Zyklen. Dieser Abschnitt ist der Erstellung der ersten Version der Anwendung gewidmet.

3.1.1 Am Anfang war die Idee...

Auch am Anfang jeder Anwendung steht das (gedachte) Wort, also die Idee, der Gedanke. Diese Idee bestimmt das primäre Ziel, das mit Hilfe der Anwendung erreicht werden soll. Das kann eine mathematische Berechnung, das vereinfachte Erstellen eines Serienbriefes oder eine Spielidee sein, die umgesetzt werden soll. Die Beispielanwendungen in diesem Buch wurden mit dem Ziel erstellt, das Erlernen der Delphi-Programmierung zu vereinfachen. Anwendungen können auch erstellt werden, um zu sehen, ob und wie etwas funktioniert, um zu zeigen, wie gut man ist und natürlich auch, um Geld zu verdienen.

Wir wollen in diesem Kapitel den Ablauf der Programmentwicklung am Beispiel einer typischen Delphi-Anwendung demonstrieren, bei der die grafische Oberfläche eine große Rolle spielt. Als Beispiel verwenden wir ein einfaches Rechenprogramm, das verschiedene Maße in verschiedene Einheiten umwandeln kann, also z. B. Meter in Zoll, englische Pfund in Kilogramm usw. Dabei soll der Anwender schnell eine solche Umwandlung durchführen können, ohne dazu erst eine längere Einarbeitung in das Programm zu benötigen.

3.1.2 Die Maus kommt zum Zuge

Es gibt verschiedene Möglichkeiten, den Entwurf einer Anwendung zu beginnen. Mittlerweile gibt es auch anerkannte Standards für objektorientierte Analyse und Design, die in einer eigenen grafischen Beschreibungssprache implementiert sind – der UML (*Unified Modeling Language*). Diese Sprache ist ein sehr mächtiges Mittel der Projektierung und Umsetzung vor allem großer Anwendungen mit objektorientierten Methoden.

Rapid Prototyping

Wir werden uns an dieser Stelle mit einem sehr intuitiven und einfachen Herangehen begnügen. Dazu wollen wir uns an eine Methode anlehnen, die als »Rapid Prototyping« bezeichnet wird und besonders für RAD-Tools wie Delphi geeignet ist. Bei dieser Vorgehensweise wird zunächst ein sehr grobes Modell der Anwendung erstellt (der »Prototyp«), der aber bereits funktionsfähig ist. Dieser Prototyp wird dann Schritt für Schritt verbessert und getestet, bis er zur voll funktionsfähigen Anwendung wird.

Dazu wollen wir zuerst eine Variante erstellen, die nur eine Umwandlung vornimmt. Wir wählen dafür die Temperatur, wo es nur drei allgemein verwendete Maßeinheiten gibt (Grad Celsius, Fahrenheit und Kelvin).

Erstellen Sie zunächst eine neue Anwendung (DATEI | NEUE ANWENDUNG). Wir benötigen ein Element zur Eingabe einer Zahl, die Möglichkeit der Auswahl einer Einheit, aus der konvertiert werden soll, und einer Einheit, in die konvertiert werden soll, außerdem ein Element zur Anzeige des Ergebnisses. Zur Eingabe von Zahlen wird die *Edit*-Komponente aus dem Register STANDARD verwendet, die Sie im nächsten Kapitel noch näher kennenlernen werden.

R 7 Optionsfelder verwenden

Zur Auswahl einer Einheit sind verschiedene Steuerelemente geeignet (Listenfelder, Kombinationsfelder, Optionsfelder), für die jeweils entsprechende Komponenten in Delphi existieren. Wir wählen hier die Optionsfelder. In Delphi gibt es zwei Komponenten im Register STANDARD, die Optionsfelder verwalten: *TRadioButton* und *TRadioGroup*. Optionsfelder zeigen Auswahlfelder an, von denen jeweils nur eines aktiv sein kann; sie treten deshalb immer in Gruppen von mindestens zwei Elementen auf. Um diese Gruppen auch optisch zusammenzufassen, werden sie meist mit einem Rahmen versehen.

Wenn Sie die *RadioButton*-Komponente verwenden, dann müssen Sie sich um die Gruppierung der Elemente selbst kümmern, da diese Komponente ein einzelnes Optionsfeld darstellt. Die *RadioGroup*-Komponente stellt dagegen bereits eine fertige Gruppe von Optionsschaltern mit Rahmen dar, die automatisch gleichmäßig innerhalb des Rahmens angeordnet werden

und die auch als Gruppe einfacher verwaltet werden. Deshalb werden Sie im allgemeinen diese Komponente für Optionsschalter verwenden, es sei denn, Sie wollen die Schalter nicht standardmäßig anordnen, oder der Rahmen wird nicht benötigt.

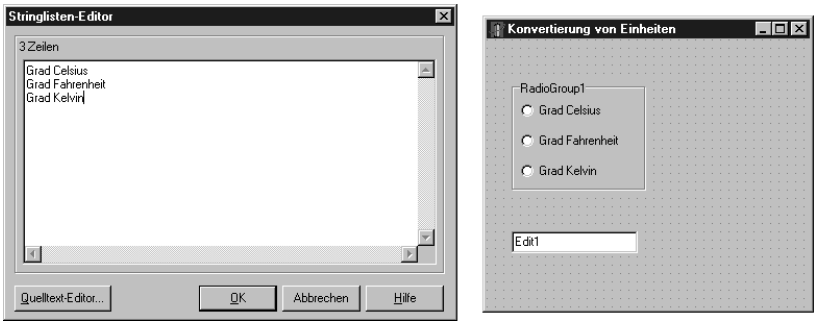

Abbildung 3.1: Verwendung des Stringlisten-Editors und das Ergebnis

Wenn Sie eine *RadioGroup*-Komponente auf dem Formular plazieren, dann ist sie zunächst leer (d.h., sie besitzt keine Schalter). Die einzelnen Schalter fügen Sie hinzu, indem Sie im Objektinspektor auf die Schaltfläche neben der Eigenschaft *Items* vom Typ *TStrings* klicken. Daraufhin erscheint ein Dialogfenster (der Stringlisten-Editor), in den Sie einzelne Zeichenketten eintragen können (siehe Abbildung 3.1). Für jede eingetragene Zeichenkette wird dann ein Optionsfeld erzeugt.

Wenn Sie diese Gruppe wie in der Abbildung erstellt haben, benötigen Sie noch eine zweite solche Gruppe. Anstatt diese aber neu zu erzeugen, können Sie die vorhandene Komponente einfach kopieren. Dazu markieren Sie die Komponente und drücken Strg C, oder Sie wählen den entsprechenden Menüpunkt im Menü BEARBEITEN, wie Sie das aus anderen Anwendungen gewohnt sind. Danach können Sie die Komponente aus der Zwischenablage beliebig oft wieder einfügen (z.B. mit Strg V).

Schließlich benötigen Sie auch noch ein Element zur Ausgabe des Ergebnisses; dazu plazieren wir eine *Label*-Komponente auf dem Formular. Damit haben wir bereits alle Elemente, die wir für die erste Version benötigen. Sie sollten nur noch die Namen der einzelnen Komponenten so anpassen, daß erkennbar wird, wozu sie verwendet werden; die Vorgabenamen sind dazu im allgemeinen ungeeignet. In Abbildung 3.2 sehen Sie das Ergebnis (Sie können natürlich andere Namen verwenden).

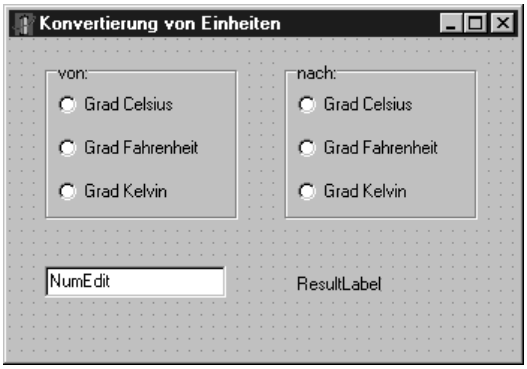

Abbildung 3.2: Das Formular für die erste Version

3.1.3 Die Schreibarbeit

Jetzt kümmern wir uns um die eigentliche Funktionalität: um den Code. Wir müssen also die Umrechnung selbst durchführen, und zwar immer, wenn der Anwender eine Bedingung ändert. Da das an verschiedenen Stellen der Fall sein kann, erstellen wir dafür eine eigene Prozedur, die wir dann an verschiedenen Stellen aufrufen können. Sie können die Prozedur im *private*-Teil der Formulardeklaration deklarieren:

```
  private
    { Private-Deklarationen }
    procedure Convert;
```

Danach erzeugen Sie die Prozedur selbst (zur Erinnerung: Das funktioniert mit Strg ⇧ C).

```
procedure TConversionForm.Convert;
var
  temp: Double;
begin
  temp := StrToFloat(NumEdit.Text);
  case UnitRadioGroup1.ItemIndex of
    1: temp := ((temp - 32.0) * 5) / 9;
    2: temp := temp - 273.15;
  end;
  case UnitRadioGroup2.ItemIndex of
    1: temp := ((temp * 9) / 5) + 32.0;
    2: temp := temp + 273.15;
  end;
  ResultLabel.Caption := FloatToStr(temp);
end;
```

Zunächst wird der eingegebene Text gelesen und in eine Real-Zahl konvertiert (mit Hilfe der Delphi-Funktion *StrToFloat*), danach wird die eigentliche Umwandlung ausgeführt, und am Ende wird das Ergebnis wieder in einen Text umgewandelt (*FloatToStr*) und der *Label*-Komponente als Beschriftung zugeordnet.

Wir führen die Umwandlung hier in zwei Stufen durch. Zunächst konvertieren wir die eingegebene Zahl in Grad Celsius, danach wird von Grad Celsius in die benötigte Einheit umgewandelt. Dadurch vermeiden wir die Unterscheidung zu vieler Fälle; bei drei Einheiten gäbe es allein neun Möglichkeiten der Umrechnung (einschließlich der »Umwandlung« von Einheiten in sich selbst, was ja auch gewählt werden kann), und bei anderen Maßen gibt es noch mehr Einheiten. Die Eigenschaft *ItemIndex* der *RadioGroup*-Komponente gibt die nullbasierte Nummer des aktuell gewählten Optionsfeldes an. Der Fall, in dem dieser Index Null ist (für Grad Celsius), wird hier einfach übergangen, da in diesem Fall die Variable *temp* nicht geändert werden muß.

Jetzt müssen wir die Prozedur nur noch an den richtigen Stellen aufrufen. Die Berechnung soll immer dann neu erfolgen, wenn der Anwender eine andere Zahl eingibt oder eine andere Einheit wählt. Sie müssen also Behandlungsroutinen für die entsprechenden Ereignisse erstellen. Für die *RadioGroup*-Komponente ist dies das Ereignis *OnClick*, für die *Edit*-Komponente das *OnChange*-Ereignis. Markieren Sie also die *Edit*-Komponente, wechseln Sie mit F11 zum Objektinspektor, und doppelklicken Sie auf der Ereignisseite neben *OnChange*. In der so erstellten Methode können Sie dann unsere Prozedur aufrufen:

```
procedure TConversionForm.NumEditChange(Sender: TObject);
begin
  Convert;
end;
```

Abbildung 3.3: Die Liste der Ereignisse im Objektinspektor

Für die anderen beiden Ereignisse (die *OnClick*-Ereignisse der *RadioGroup*-Komponenten) müssen Sie dann keine eigenen Ereignisse mehr erstellen. Statt dessen ordnen Sie ihnen einfach die gleiche Ereignisbehandlung zu, indem Sie sie im Objektinspektor aus der Liste von Ereignissen auswählen, die in diesem Fall nur aus einer Prozedur besteht (siehe Abbildung 3.3).

Initialisierung

Jetzt sind Sie soweit, daß Sie die Anwendung das erste Mal ausprobieren können. Wenn Sie das tun und alles gemacht haben wie beschrieben, dann werden Sie schon sehr schnell die ersten Fehler sehen, die mit einer fehlenden Initialisierung zu tun haben. Am offensichtlichsten ist die Ausnahme, die hervorgerufen wird, weil sich im Editierfeld keine gültige Zahl befindet. Sie sollten also den Text dieses Feldes im Objektinspektor auf eine gültige Zahl (z.B. »1«) setzen. Ein zweiter Initialisierungsfehler führt zwar nicht zu einer Fehlermeldung, ist aber für den Anwender irritierend: Keines der Optionsfelder ist anfänglich ausgewählt, weil die Eigenschaft *ItemIndex* standardmäßig auf -1 gesetzt ist. Sie sollten diesen Wert für beide Komponenten im Objektinspektor auf 0 setzen.

Außerdem wird nach Programmstart auch das Ergebnis noch nicht angezeigt. Sie könnten jetzt die Eigenschaft *Caption* der *Label*-Komponente von Hand auf das richtige Ergebnis setzen; das sollten Sie aber normalerweise nicht tun, da hier leicht Fehler unterlaufen. Statt dessen rufen Sie einfach Ihre Konvertierungsprozedur gleich nach Programmstart einmal auf. Der günstige Platz dafür ist das *OnCreate*-Ereignis des Formulars, das gleich nach der Erstellung des Formulars erzeugt wird. Auch in diesem Fall brauchen Sie keine eigene Prozedur zu erstellen, sondern können die Ereignisbehandlung von *NumEdit* übernehmen.

Schließlich werden Sie beim Probieren feststellen, daß immer dann, wenn Sie einen ungültigen Wert eingeben, eine Ausnahme erzeugt wird. Besonders lästig ist das, wenn Sie alle Zeichen löschen, um einen neuen Wert einzugeben. Eine weniger störende Reaktion auf eine solche Eingabe wäre einfach die Anzeige einer entsprechenden Fehlermeldung statt eines Ergebnisses. Das können Sie mit Hilfe der *try-except*-Anweisung erreichen, indem Sie schreiben:

```
try
  temp := StrToFloat(NumEdit.Text);
  ...
  ResultLabel.Caption := FloatToStr(temp);
except
  ResultLabel.Caption := 'Zahl ungültig!';
end;
```

Dadurch werden die Ausnahmen, die bei der Umwandlung durch ungültige Eingaben entstehen, abgefangen. In Kapitel 9 werden wir näher auf die Ausnahmebehandlung eingehen.

Wenn Sie das Programm aus der Entwicklungsumgebung starten, tritt trotzdem eine Ausnahme auf, die in diesem Fall der Fehlersuche dient. Sie können dieses Verhalten abschalten, indem Sie in den Debugger-Optionen die Option »Bei Delphi-Exceptions stoppen« zeitweilig ausschalten (siehe auch Kapitel 9).

Nach dieser ersten Fehlerbereinigung sind wir praktisch am Ende der ersten Runde; Sie finden das Programm auf der CD-ROM unter KAPITEL 3\EINHEITEN\VERSION1. Allerdings haben wir die Fehlersuche an dieser Stelle recht schnell abgetan, da es sich um eine sehr einfache Anwendung handelt. In komplexeren Anwendungen ist die Fehlersuche dagegen eine sehr wichtige (und teilweise auch langwierige) Angelegenheit, weshalb wir ihr den nächsten Abschnitt widmen.

3.2 Die Stunde der Wahrheit: Fehlersuche

Ein sehr wichtiger Teil der Anwendungsentwicklung ist das Testen einer Anwendung. Dabei soll zum einen sichergestellt werden, daß die Anwendung unter allen möglichen Bedingungen wie vorgesehen funktioniert und keine unvorhergesehenen Fehler auftreten; zum anderen muß natürlich auch getestet werden, ob die potentiellen Anwender mit Bedienung und Benutzerführung klarkommen. Was den zweiten Teil betrifft, so werden Sie anfänglich damit kaum konfrontiert werden, jedenfalls, solange Sie selbst Anwender und Tester Ihrer Programme in einer Person sind. Mit dem ersten Teil, der Fehlersuche, werden Sie dagegen sehr viel zu tun haben.

Debugging

Die Fehlersuche in Anwendungen wird als Debugging oder neudeutsch Debuggen (*debug* = entwanzen) bezeichnet, die Fehler entsprechend als Bugs und das Werkzeug zur Fehlersuche als Debugger. Angeblich kommt die Bezeichnung daher, daß in den ersten Großcomputern verschiedene Insekten, die sich innerhalb des Computers befanden, Kurzschlüsse und Fehler verursachten (der erste gefundene Bug soll eine Motte gewesen sein, die 1945 aus einem Mark II-Rechner gezogen wurde).

Delphi erlaubt eine ganze Menge verschiedener Techniken, die Sie bei der Fehlersuche unterstützen. Die Kenntnis dieser Techniken ist nach der Kenntnis der Sprache wohl die wichtigste Voraussetzung für eine effektive Programmierung. Deshalb stellen wir Ihnen in diesem Abschnitt verschiedene Elemente dieser Techniken etwas genauer vor. Wir verwenden dabei zwar unsere Beispielanwendung zur Erklärung, haben aber nicht extra Feh-

ler in diese Anwendung eingebaut, die wir jetzt »suchen« wollen, sondern zeigen Ihnen einfach nur die verschiedenen Möglichkeiten der Fehlersuche, die Sie kennen sollten. Sie können später wieder auf diesen Absatz zurückkommen, wenn Sie in Ihren eigenen Übungen diese Techniken benötigen.

Achten Sie darauf, daß in den Compileroptionen im Abschnitt DEBUGGEN die Optionen »Debug-Informationen« und »Lokale Symbole« angeschaltet sind, bevor Sie das Programm kompilieren und testen.

R 8 Haltepunkte setzen und Programm schrittweise ausführen

Haltepunkte, oder *Breakpoints*, wie die englische Variante lautet, sind die Grundlage einer effektiven Fehlersuche. Ein Haltepunkt ist einfach eine Zeile in Ihrem Quelltext, an der die Ausführung des Programms gestoppt werden soll; das erlaubt es Ihnen, den Zustand des Programms zu diesem Zeitpunkt genauer zu untersuchen.

Das Setzen eines Haltepunktes ist sehr einfach: es reicht, links neben die entsprechende Zeile zu klicken (ab Delphi 3 in den Bundsteg neben dem Quelltext). Daraufhin erscheint dort ein roter Punkt, und die Quelltextzeile wird rot unterlegt. Sie können einen Haltepunkt in jede Zeile setzen, in der im Bundsteg ein kleiner blauer Punkt erscheint (seit Delphi 3); sind diese Punkte nicht vorhanden, dann müssen Sie das Programm noch einmal kompilieren, damit sie erscheinen. Das Entfernen eines Haltepunktes erfolgt auf die gleiche Art: Klicken Sie einfach darauf.

Haben Sie den Haltepunkt an der Sie interessierenden Stelle gesetzt (also meist an einer Stelle, die etwas mit einem auftretenden Fehler zu tun hat), dann können Sie das Programm starten, und es wird anhalten, sobald die Programmausführung an einem Haltepunkt angelangt ist (Sie können natürlich auch mehrere Haltepunkte setzen). Sie erkennen die aktuelle Zeile, also die Zeile, die als nächste ausgeführt werden soll, an dem kleinen grünen Pfeil im Bundsteg.

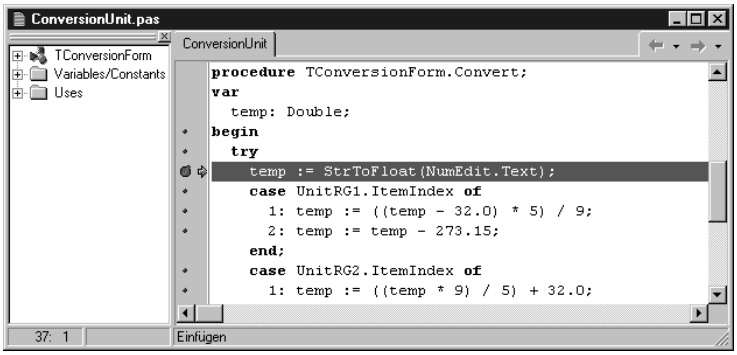

Abbildung 3.4: Am Haltepunkt angekommen

Sie können das in unserer Anwendung ausprobieren. Setzen Sie einen Haltepunkt in die erste Zeile der Prozedur *Convert*, und starten Sie das Programm. Das Programm wird gleich nach dem Start anhalten (wegen des Aufrufs im *OnCreate*-Ereignis), und Sie befinden sich im Quelltext an der Stelle, wo Sie den Haltepunkt gesetzt haben.

Jetzt beginnt die eigentliche Arbeit des Debuggens – die Untersuchung der Programmausführung. Sie können jetzt den aktuellen Zustand des Programms untersuchen (dazu kommen wir gleich) und dann in der Programmausführung fortfahren. Dabei haben Sie verschiedene Möglichkeiten:

- Sie können ganz normal mit der Programmausführung fortfahren, indem Sie auf den Schalter **Start** klicken oder F9 drücken. Damit kommen Sie bis zum nächsten Haltepunkt oder bis zum Programmende.

- Sie können durch einen Klick auf **Gesamte Routine** oder einen Tastendruck auf F8 nur die aktuelle Zeile ausführen lassen. Die nächste Zeile wird dann zur aktiven Zeile, was durch einen blauen Hintergrund angezeigt wird.

- Falls sich in der aktuellen Zeile eine Prozedur oder Funktion befindet, die Sie selbst erstellt haben, können Sie mit Hilfe von **Einzelne Anweisung** oder F7 auch in die Ausführung dieser Prozedur springen.

- Eine angenehme Neuerung in Delphi 5 ist die Möglichkeit, die Prozedur oder Funktion, in der Sie sich befinden, wieder zu verlassen, indem sie bis zum Ende ausführt und die Kontrolle an den Aufrufer zurückgegeben wird. Es passiert relativ oft, daß man versehentlich in einer Methode landet, die man gar nicht anschauen wollte und jetzt nicht schrittweise bis zum Ende ausführen möchte. Dann können Sie **Ausführung bis Rückgabe** klicken oder ⇧ F8 drücken.

- Sie können auch den Cursor auf eine Quelltextzeile setzen, die Sie als nächste untersuchen wollen, und F4 drücken oder auf **Zu Cursor gehen** klicken. Damit wird das Programm bis zu diesem Punkt ausgeführt (das entspricht dem Setzen eines Haltepunktes und F9).

- Recht nützlich kann auch der Menüpunkt START | ZEIGE AUSFÜHRUNGSPOSITION sein. Wenn Sie während des Debuggens im Quelltext »geblättert« haben, kann es manchmal schwer sein, den Punkt der Programmausführung wiederzufinden. Dieser Befehl macht eine lange Suche unnötig.

Abbildung 3.5: Eine angepaßte Werkzeugleiste

Das Icon, das zum letzten Punkt gezeigt wurde, ist (wie auch andere gezeigte Icons) standardmäßig nicht in der Werkzeugleiste enthalten. Sie können aber mit Hilfe der Menüoption ANPASSEN... im Kontextmenü der Werkzeugleiste beliebige Befehle ergänzen oder auch entfernen (in Abbildung 3.5 sehen Sie meine angepaßte Werkzeugleiste). Auch diese Kleinigkeiten sind wichtig für ein komfortables Arbeiten.

Nachdem Ihr Programm an einem Haltepunkt gestoppt hat, ist das erste, was Sie normalerweise interessiert, der Wert bestimmter Variablen oder Ausdrücke. Im folgenden Rezept zeigen wir Ihnen verschiedene Möglichkeiten, wie Sie diese Werte untersuchen können.

R 9 Variableninhalte prüfen

Da ein Programm üblicherweise zum großen Teil darin besteht, daß es Daten manipuliert und umwandelt, ist die Untersuchung dieser Daten eine der wichtigsten Möglichkeiten für das Debuggen. Sobald die Programmausführung gestoppt wurde, haben Sie dazu verschiedene Möglichkeiten.

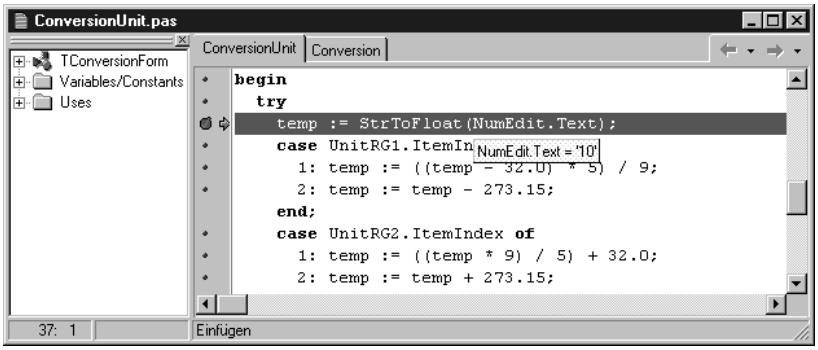

Abbildung 3.6: Auswertung durch Kurzhinweise

Kurzhinweise

Die einfachste und wohl am meisten genutzte Möglichkeit zur Anzeige von Variableninhalten ist wohl die Verwendung einer der Programmierhilfen: der Auswertung durch Kurzhinweise (seit Delphi 3). Plazieren Sie einfach den Mauscursor z. B. über `NumEdit.Text`, und warten Sie einen Moment. Es erscheint der Inhalt der Variablen in einem kleinen gelben Kästchen (siehe Abbildung 3.6). Wenn Sie etwas damit experimentieren, dann werden Sie sehen, daß auch der Inhalt von Variablen strukturierter Typen, wie Records und Arrays, angezeigt wird. Das gleiche gilt für einzelne Felder von Records – allerdings nur dann, wenn keine *with*-Anweisung verwendet wurde. Wenn Sie versuchen, sich den Inhalt von Feldern ohne Qualifizierer anzuschauen, dann werden Sie nichts sehen. Sollten Sie

innerhalb einer *with*-Anweisung doch den Inhalt eines Feldes sehen können, dann heißt das nur, daß eine solche Variable oder ein solches Feld auch außerhalb der *with*-Anweisung existiert.

Lokale Variablen

Sie haben aber noch andere Möglichkeiten, sich Variableninhalte anzuschauen. Seit Delphi 4 gibt es die automatische Anzeige lokaler Variablen in einem eigenen Fenster. Da der Inhalt zumindest einiger lokalen Variablen fast immer benötigt wird, ist diese Option sehr nützlich. Sie finden dieses und andere Fenster, die für das Debuggen verwendet werden können, unter ANSICHT | DEBUG-FENSTER. Da Sie in Delphi sowohl die Möglichkeit haben, diese Fenster an das Quelltextfenster anzudocken, als auch, sie gemeinsam mit anderen Fenstern so anzuzeigen, daß sie immer über Register anwählbar sind, ist auch die Plazierung solcher zusätzlicher Fenster kein Problem. Im übrigen haben Sie auch noch die Möglichkeit, im Kontextmenü solcher Fenster die Option IMMER IM VORDERGRUND einzuschalten, falls Sie sie über dem Quelltextfenster anzeigen lassen wollen.

Wenn Sie in der Methode *Convert* in unserem Beispiel einen Haltepunkt setzen und sich die lokalen Variablen anschauen, bekommen Sie die einzige lokale Variable *temp* sowie den *self*-Zeiger (den Zeiger auf das aktuelle Formular-Objekt) angezeigt.

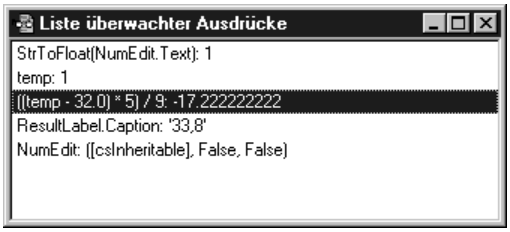

Abbildung 3.7: Verschiedene überwachte Ausdrücke

Liste überwachter Ausdrücke

Es gibt noch ein weiteres Fenster, das der Anzeige von Variableninhalten dient: die *Liste überwachter Ausdrücke*. Wie der Name schon andeutet, dient es nicht nur der Anzeige von Variablen, sondern kann auch Ausdrücke auswerten und das Ergebnis anzeigen (siehe Abbildung 3.7). Sie können jederzeit eine Variable zu diesem Fenster hinzufügen, indem Sie den Cursor auf die Variable setzen und Strg F5 drücken. Sie können auch über das Kontextmenü des Fensters beliebige Variablen und Ausdrücke zur Überwachung hinzufügen. Schließlich können Sie auch Ausdrücke per Drag and Drop aus dem Quelltextfenster und anderen Fenstern in die Liste ziehen (neu in Delphi 5). Diese Liste hat im Vergleich zu den gezeigten Möglichkeiten der Anzeige einige Vorteile:

- Es kann der Inhalt beliebiger Variablen angezeigt werden, egal, ob sie im Quelltextfenster gerade zu sehen sind oder nicht oder ob sie sich innerhalb einer *with*-Anweisung befinden.
- Es können Ausdrücke zur Laufzeit ausgewertet werden, was die Überprüfung von Ausdrücken im Quelltext viel einfacher macht.
- In Delphi 5 können auch Funktionen in diesem Fenster ausgewertet werden. Ein Beispiel dafür sehen Sie in der ersten Zeile von Abbildung 3.7. Voraussetzung dafür ist, daß Sie in den Debugger-Optionen die Option »Funktionsaufrufe in neuen überwachten Ausdrücken« ankreuzen.

NEU

- Die Liste ist permanent vorhanden, auch nach einem Neustart der Anwendung, und damit immer sichtbar.
- Ebenfalls neu in Delphi 5 ist das Kontextmenü des Fensters, das unter anderem die schnelle Änderung von Ausdrücken erlaubt.

Schließlich gibt es auch die Möglichkeit, den Inhalt von Variablen zur Laufzeit zu ändern. Zu diesem Zweck dient das Dialogfenster »Auswerten und Ändern«, das in Delphi 5 noch zusätzliche Optionen bekommen hat. Um das auszuprobieren, setzen Sie in unserem Beispielprogramm einen Haltepunkt auf die Zeile:

```
ResultLabel.Caption := FloatToStr(temp);
```

Wenn Sie beim Ausführen des Programms an diesem Haltepunkt angekommen sind, setzen Sie den Cursor auf die Variable *temp* und drücken Strg F7. In der erscheinenden Dialogbox können Sie den aktuellen Wert der Variablen *temp* sehen und in dem unteren Eingabefeld auch ändern (siehe Abbildung 3.8).

Wenn Sie jetzt den Variableninhalt ändern und die Anwendung weiterlaufen lassen, können Sie die Änderung am angezeigten Ergebnis sehen.

Soweit zu den Variableninhalten. Jetzt schauen wir uns etwas genauer die anderen Möglichkeiten an, die uns für das Debuggen zur Verfügung stehen.

Der *Aufruf-Stack* zeigt eine Liste der übergeordneten Prozeduren an, von denen die aktuelle Prozedur aufgerufen wurde, sowie deren Aufrufparameter. Sie können auch durch einen Doppelklick auf eine der aufrufenden Prozeduren mit dem Schreibcursor dorthin springen. Sie benötigen den Aufruf-Stack beispielsweise, wenn Sie einen Haltepunkt in einer selbstdefinierten Prozedur gesetzt haben und wissen wollen, von wo sie aufgerufen wurde. Auch bei rekursiven Funktionsaufrufen kann der Aufruf-Stack nützlich sein (siehe auch Kapitel 8).

Aufruf-Stack

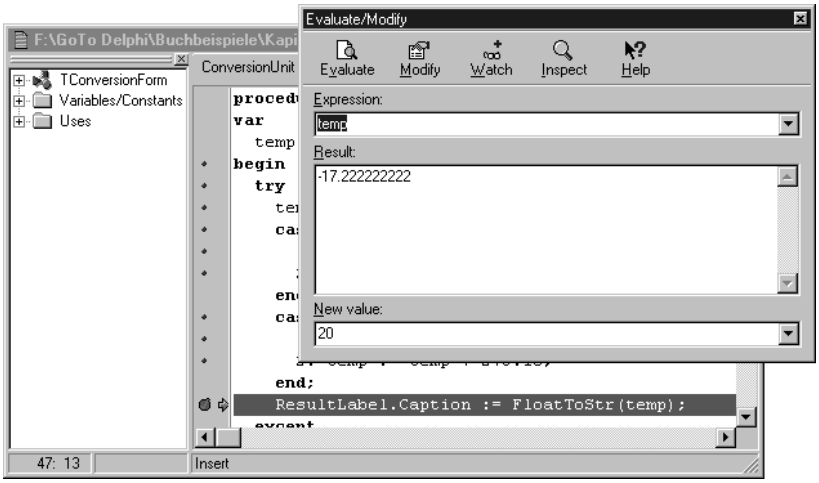

Abbildung 3.8: Der Dialog »Auswerten und Ändern«

Symbolinformation durch Kurzhinweis

Zu den Programmierhilfen gehört die sogenannte *Symbolinformation durch Kurzhinweis*, die in Delphi 4 eingeführt wurde (nur ab Profiversion). Dabei handelt es sich um die Anzeige der zugehörigen Deklaration zu einem Bezeichner (egal, ob es sich dabei um eine Variable, Prozedur oder Unit handelt). Angezeigt wird dabei der Typ des Bezeichners (*proc* für Prozedur, *prop* für Eigenschaft, *const* für Konstante usw.), der Name des Bezeichners mit Qualifizierer, soweit vorhanden, der Name der Quelltextdatei, in der sich die Deklaration befindet, und die Zeilennummer der Deklaration in dieser Datei (siehe Abbildung 3.9 links). Das ermöglicht es Ihnen, einen Bezeichner und seine Herkunft schneller zu identifizieren.

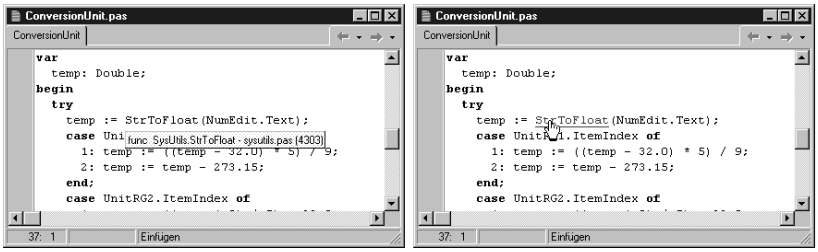

Abbildung 3.9: Symbolinformation durch Kurzhinweis und Quelltext-Browser

Quelltext-Browser

Damit im Zusammenhang steht der sogenannte *Quelltext-Browser*. Er wird aktiv, wenn Sie die Strg-Taste gedrückt halten und mit der Maus auf einen Bezeichner zeigen. Dieser erscheint dann unterstrichen, und die Form des Mauszeigers wandelt sich zu einer Hand (siehe Abbildung 3.9

rechts). Wenn Sie jetzt auf den Bezeichner klicken, dann springen Sie zu dem Teil des Quelltextes, in dem der Bezeichner deklariert wurde, so wie es mit Hilfe der Symbolinformation durch Kurzhinweis angezeigt wurde. Das funktioniert auch dann, wenn es sich um Quellcode der VCL handelt. Auch diese Option steht ab der Profiversion von Delphi 4 zur Verfügung. Dazu gehört auch, daß im Editor jede Position, an die Sie gesprungen sind, gespeichert wird, so daß Sie mit Hilfe der beiden Pfeiltasten rechts oben im Quelltexteditor an diese Positionen zurückkehren können.

Abbildung 3.10: Definition eines bedingten Haltepunktes

Bedingte Haltepunkte

Weitergehende Möglichkeiten, als wir Ihnen bisher gezeigt haben, bieten auch die Haltepunkte selbst. Stellen Sie sich vor, Sie haben eine *for*-Schleife programmiert, deren Zähler von 1 bis 1000 läuft. Sie sehen bei der Fehlersuche, daß offenbar am Ende der Schleifendurchläufe ein Fehler auftritt, der zu falschen Werten führt. Was machen Sie, wenn Sie das Schritt für Schritt im Debugger prüfen wollen? Sie können natürlich in der Schleife einen Haltepunkt setzen und dann 999 mal auf F9 drücken; einfacher ist es aber, wenn Sie einen bedingten Haltepunkt definieren. Dazu wählen Sie aus dem Kontextmenü des Bundstegs neben dem Haltepunkt die Option HALTEPUNKT-EIGENSCHAFTEN; hier können Sie eine Bedingung einsetzen, bei deren Auftreten der Haltepunkt aktiv wird (siehe Abbildung 3.10).

NEU

Ab Delphi 5 gibt es außerdem die Möglichkeit, einem Haltepunkt eine Aktion zuzuordnen. Interessant kann beispielsweise sein, bei Erreichen des Haltepunktes eine Meldung oder das Ergebnis eines Ausdrucks ins Ereignisprotokoll zu schreiben, statt das Programm zu stoppen. Neu ist ebenfalls, daß es jetzt auch einen Kurzhinweis für Haltepunkte gibt – wenn Sie den Cursor über einen Haltepunkt führen, sehen Sie die Zusatzinformationen, über die wir gerade geschrieben haben, in einem solchen kleinen Fenster.

Das Ereignisprotokoll (seit Delphi 4) wird benutzt, um Ereignisse verschiedener Herkunft zu protokollieren.

Seien Sie vorsichtig mit der Nutzung der Option zur Anzeige von Windows-Botschaften im Ereignisprotokoll (die in der mir vorliegenden Delphi 5-Version irrtümlicherweise »Meldungsfenster« statt »Fenstermeldungen« heißt). Auf meinem System führte der Versuch, diese Option zu benutzen, zum (reproduzierbaren) Totalabsturz – sowohl unter Delphi 4 als auch unter Delphi 5. Wenn Sie die Profiversion benutzen, dann haben Sie mit *WinSight* auch ein Werkzeug zur Verfügung, das dafür besser geeignet ist.

Aufgabe 1

Setzen Sie in unserem Beispiel einen Haltepunkt, der bei jeder durchgeführten Konvertierung eine Meldung im Ereignisprotokoll und das Ergebnis der Umwandlung ausgibt.

Außer den aufgezählten gibt es einige Möglichkeiten des Debuggens, die Sie zu Beginn kaum benötigen werden (zumindest, wenn Sie Programmieranfänger sind), die aber Profiprogrammierern eine sehr nützliche Hilfe sein können. Wir stellen diese Möglichkeiten hier nur kurz vor.

R 10 Fortgeschrittene Möglichkeiten der Fehlersuche

Seit Delphi 4 können Sie auch Haltepunkte definieren, die bei der Änderung einer bestimmten Adresse aktiv werden. Dabei kann es sich um eine Variable, ein Feld einer Struktur oder auch eine reine Adresse handeln. Diese Option ermöglicht es beispielsweise, herauszufinden, wo und wann bestimmte Variableninhalte geändert werden – eine sehr nützliche Funktion besonders für komplexere Programme, die viel Zeit bei der Fehlersuche einsparen hilft. Sie können einen solchen Haltepunkt nur zur Laufzeit des Programms definieren.

CPU-Fenster

Seit Delphi 4 gibt es direkt in Delphi integriert ein CPU-Fenster – ein Merkmal, das vorher externen Debuggern, wie dem Turbo Debugger, vorbehalten war. Dieses Fenster ermöglicht Ihnen das Debuggen auf der niedrigsten Ebene – Sie können hier einzelne Maschinenbefehle verfolgen, die Inhalte der Prozessorregister auslesen und sich Prozessorstack und Arbeitsspeicher anschauen. Sie haben auch alle Möglichkeiten zur Änderung dieser Inhalte, zur Veränderung der Anzeige und zum Navigieren zur Verfügung, die sonst nur in speziellen Debuggern zur Verfügung stehen. Damit ist auch in diesem Bereich eine Lücke zu anderen Sprachen geschlossen, die traditionell auf niedriger Ebene arbeiten können. In Delphi 5 haben Sie außerdem die Möglichkeit, sich die Register der FPU (*Floating Point Unit*) anzuschauen, also des mathematischen Coprozessors (ab der Profiversion), inclusive der MMX-Informationen.

Thread-Fenster und Modul-Fenster

Ebenfalls eher für Profiprogrammierer gedacht sind das Thread-Fenster, das einzelne Threads (unabhängig voneinander ausgeführte Teilprozesse) anzeigt, und das Modul-Fenster, das eine detaillierte Anzeige der aktiven Windows-Module erlaubt. Ab Delphi 4 können hier alle Quelltexte der verwendeten Units angezeigt werden (wenn Sie die Profiversion besitzen) und sogar die Einsprungpunkte aller Funktionen der verwendeten Windows-Bibliotheken, also das Assemblerlisting dieser Funktionen, eingesehen werden. Beide Fenster unterstützen jetzt auch die Anzeige mehrerer Prozesse. Auch das Debuggen mehrerer Prozesse gleichzeitig wird seit Delphi 4 unterstützt.

NEU

Eine wichtige Neuerung für Profis ist in Delphi 5 die Möglichkeit, außerhalb der IDE laufende Prozesse zu debuggen, indem der Debugger an einen beliebigen laufenden Prozess nachträglich »angehängt« wird.

Externe Fehlersuche

Schließlich wird seit Delphi 4 auch das Debuggen von Prozessen unterstützt, die auf einem entfernten PC ausgeführt werden (*Remote debugging*). Der Prozeß läuft dabei auf dem entfernten Rechner mit Hilfe eines sogenannten Debug-Servers, während die normale Delphi-Umgebung auf dem lokalen Rechner über das Netzwerk mit dem Prozeß kommuniziert. Auch das ist eine wichtige Option, die für die Erstellung professioneller Anwendungen wichtig sein kann.

3.3 In die nächste Runde

Nachdem wir Ihnen die wichtigsten Techniken zur Fehlersuche vorgestellt haben, kommen wir wieder zu unserem Beispiel zurück. Wir haben einen funktionsfähigen Prototypen geschaffen, den wir jetzt zu der Anwendung weiterentwickeln wollen, die wir geplant haben. Bevor wir mit der Programmierung weitermachen, kommen wir aber noch einmal auf die Gestaltung der Oberfläche zurück, die wir in der ersten Runde ziemlich vernachlässigt haben.

3.3.1 Kleider machen Leute – Die Programmoberfläche

Beim Entwurf der Oberfläche einer Anwendung gibt es eine ganze Reihe Dinge zu beachten, um sie für den Anwender möglichst einfach bedienbar und ergonomisch zu machen. Über Softwareergonomie allgemein und Ergonomie der Oberfläche speziell sind viele Bücher geschrieben worden, in denen Sie Tonnen guter und weniger guter Ratschläge finden. Auch Microsoft selbst hat Richtlinien für die Entwicklung der Anwenderschnittstelle erstellt. Wenn Sie selbst professionelle Software erstellen wollen, dann werden Sie nicht umhinkönnen, sich genauer damit auseinanderzusetzen.

An dieser Stelle geben wir Ihnen eine Reihe von Ratschlägen, die unserer Meinung nach nützlich sein können. Im Laufe dieses und der weiteren Kapitel werden wir dann von Zeit zu Zeit einzelne Aspekte der Oberflächengestaltung kurz anreißen. Die folgenden Regeln sind auch nur als Anhaltspunkt gedacht; Sie müssen selbst entscheiden, ob Sie Ihnen folgen oder nicht.

R 11 Regeln für die Oberflächengestaltung

R 11

- Bemühen Sie sich, Ihre Anwendungen so zu gestalten, daß sie den allgemein bekannten und akzeptierten *Standards* entsprechen, auch wenn Sie vielleicht selbst nicht von diesen Standards begeistert sind (es sei denn, Sie heißen Kai Krause oder sind ein ähnlich begnadeter Programmierer). Die Anwender haben im allgemeinen nicht die Geduld, sich in Anwendungen zurechtzufinden, die nicht diesen Standards entsprechen. Standard heißt in diesem Fall Microsoft-Standard, also das »Look and Feel« der Windows-Oberfläche und der Office-Produkte; dazu gehören solche Elemente wie Menüleiste, Werkzeugleiste, Statuszeile und Kontextmenüs, wie Sie sie teilweise bereits in Kapitel 1 kennengelernt haben.
- Bemühen Sie sich um *Übersichtlichkeit*. Dazu gehört, daß Sie die Elemente in den Fenstern nach Zusammengehörigkeit gruppieren, nicht zu viele Elemente auf einmal anzeigen, unnötige Informationen weglassen. Der Anwender soll auf den ersten Blick erkennen können, was er tun kann, und nur solche Informationen zur Verfügung haben, mit denen er etwas anfangen kann.
- Zwingen Sie dem Anwender nicht Ihren Stil auf. Ein typischer Anfängerfehler ist die Verwendung möglichst bunter Bilder, großer Schalter, vieler »Verschönerungen« usw., die nicht abgeschaltet oder geändert werden können. Das kann in bestimmten Bereichen seine Berechtigung haben, im allgemeinen reizt es auf die Dauer die Augen und führt eher zur Verwirrung. Lassen Sie den *Anwender selbst entscheiden*, welche Farbkombinationen er bevorzugt, indem Sie für Fensterhintergründe u.ä. die Standardfarben verwenden, die der Anwender in den Bildschirmeinstellungen jederzeit ändern kann. Sie können natürlich auch selbst die Möglichkeit der Farbauswahl für verschiedene Elemente und weitere Möglichkeiten der Anwenderkontrolle einbauen.
- Denken Sie daran, daß es verschiedene Gruppen von Anwendern gibt: vom Anfänger bis zum Profi. Stellen Sie *verschiedene Möglichkeiten der Bedienung* bereit, und geben Sie dem Anwender damit die Möglichkeit, die für ihn passende Möglichkeit auszuwählen. In Kapitel 4 lernen Sie, wie Sie eine Aktion verschiedenen Oberflächenelementen zuweisen können. Vergessen Sie nicht, Ihre Anwendungen auch zu testen, indem Sie sie beispielsweise ohne Hilfe der Maus bedienen.

Das sind nur einige wenige allgemeine Regeln, die unserer Meinung nach wichtig sind; die konkrete Implementierung dieser Regeln kann nicht in einigen Sätzen beschrieben werden.

Schauen wir uns nun die Oberfläche unseres »Prototypen« etwas genauer an. Die Forderung, allgemeine Standards einzuhalten, ist hier nur rudimentär erfüllt. Allerdings haben wir hier auch eine Anwendung, die eine sehr rudimentäre Funktionalität bereitstellt, so daß wir hier auf eine Werkzeugleiste verzichten können. Ein Menü, wenn auch vorläufig mit dem einzigen Menüpunkt DATEI | BEENDEN, wollen wir der Anwendung aber trotzdem spendieren.

Zusätzlich können Sie noch eine Schaltfläche zum Schließen der Anwendung bereitstellen.

Für den Schließknopf können Sie die *Button*-Komponente aus dem Register STANDARD verwenden, wie Sie das bereits in Kapitel 1 getan haben, Sie können aber auch die etwas buntere Variante der *BitBtn*-Komponente aus dem Register ZUSÄTZLICH benutzen.

R 12 Die BitBtn-Komponente verwenden

Die *BitBtn*-Komponente stellt die gleiche Funktionalität wie die *Button*-Komponente zur Verfügung, da sie von dieser abgeleitet ist. Während aber die *Button*-Komponente nur Text enthalten kann, kann die *BitBtn*-Komponente zusätzlich eine kleine Grafik anzeigen. Sie können diese Grafik selbst definieren (darauf kommen wir in Kapitel 6 noch einmal zurück), oder Sie benutzen einen der vordefinierten Schalter, die Ihnen die Komponente über die Eigenschaft *Kind* zur Verfügung stellt. Jeder dieser Schalter hat ein kleines Bild und eine Beschriftung. In Abbildung 3.11 sehen Sie alle Möglichkeiten, die Sie dafür haben sowie den zugehörigen Wert der *Kind*-Eigenschaft.

Abbildung 3.11: Die verschiedenen Arten von BitBtn-Komponenten

Setzen Sie *Kind* auf *bkClose*, dann wird ein stilisierter Ausgang dargestellt und die Beschriftung auf »Schließen« gesetzt, aber auch die eigentliche Funktionalität des Schließens der Anwendung (eigentlich des Fensters) ist bereits eingebaut, so daß Sie sich nicht mehr darum kümmern müssen. Das ist auch die Variante, die wir in unserem Beispiel nutzen wollen.

Wenn Sie einen der Schalter in einem zweiten Formular benutzen, das Sie mit Hilfe von *ShowModal* anzeigen, bekommen Sie als Rückgabewert den Wert von *ModalResult*. Diese Eigenschaft ist für jede der gezeigten Arten auf einen anderen Wert eingestellt. Außerdem wird (außer beim Hilfe-Schalter) das Formular beim Anklicken eines Schalters geschlossen. Auf diese Weise können Sie nach dem Schließen des zweiten Formulars bestimmen, welcher der Schalter angeklickt wurde, ohne dabei auf das Formular selbst zugreifen zu müssen.

Kommen wir wieder zu unserem Beispiel. Im zweiten Punkt unserer Regeln haben wir unter anderem gefordert, daß keine überflüssigen Elemente angezeigt werden sollen. Diese Forderung ist in unserer schlichten Anwendung auf den ersten Blick eher übererfüllt; trotzdem gibt es auch hier einen Punkt, der geändert werden sollte. Wenn Sie die Anwendung starten, dann sehen Sie, daß Sie die Größe des Hauptfensters verändern können. Da wir aber eine Oberfläche haben, deren Elemente in der Größe nicht veränderlich und auch nicht verschiebbar sind, ist das eine überflüssige Option; das gleiche gilt für die Option »Vollbild«.

R 13 Fensterart einstellen

R 13

Für die Art des dargestellten Fensters (Fensterrand und Titelleiste) sind zwei Eigenschaften verantwortlich: *BorderStyle* und *BorderIcons*. In der Voreinstellung steht *BorderStyle* auf *bsSizeable*. Wird die Eigenschaft auf *bsSingle* gesetzt, kann die Größe des Fensters nicht mehr verändert werden. Das gleiche gilt für die Werte *bsDialog*, *bsToolWindow* und *bsNone*:

- Die Einstellung *bsNone* führt dazu, daß weder ein Rahmen noch eine Titelleiste vorhanden ist; ein Beispiel für die Verwendung eines solchen Fensterstils finden Sie in Kapitel 5.
- Für zeitweilig angezeigte Dialogfenster wird *bsDialog* verwendet, das nur einen Schließknopf in der Titelleiste anzeigt.
- Für Werkzeugpaletten und ähnliches wird *bsToolWindow* verwendet, das bsDialog entspricht, aber eine dünnere Titelleiste anzeigt. Eine Variante davon, die in der Größe veränderlich ist, wird durch den Wert *bsSizeToolWindow* erzeugt.

Die Schaltfläche für die Vollbilddarstellung können Sie deaktivieren, indem Sie die Untereigenschaft *biMaximize* der Eigenschaft *BorderIcons* auf *False* setzen. Ganz verstecken können Sie die Schaltfläche aber nicht – das funktioniert nur, wenn zugleich auch die Schaltfläche zum Minimieren versteckt wird. Die beiden Untereigenschaften *biMaximize* und *biMinimize* werden nur für »normale« Fenster (*BorderStyle bsSizeable* oder *bsSingle*) genutzt – bei allen anderen Fensterarten werden die beiden Schalter nie angezeigt. Die Untereigenschaft *biHelp* wird dagegen nur für alle Fenster außer Dialogfenster (*BorderStyle* ist *bsDialog*) ignoriert – sie ist für das kleine Fragezeichen in der Titelleiste verantwortlich.

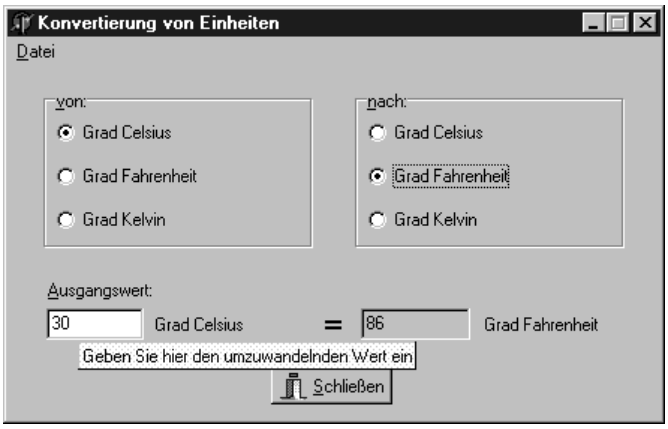

Abbildung 3.12: Die verbesserte Version zur Laufzeit

Was die nötigen Informationen in der Oberfläche unserer Anwendung betrifft, so sind sie etwas spartanisch ausgefallen. Der Anwender sollte zumindest einen Hinweis darauf erhalten, was er in das Editierfeld eingeben kann und was das Ergebnis bedeutet. Eine einfache Lösung sehen Sie in Abbildung 3.12. Hierbei wurden einfach zwei *Label*-Komponenten ergänzt, die die aktuell ausgewählte Einheit anzeigen:

```
UnitLabel1.Caption :=
  UnitRadioGroup1.Items[UnitRadioGroup1.ItemIndex];
UnitLabel2.Caption :=
  UnitRadioGroup2.Items[UnitRadioGroup2.ItemIndex];
```

Beachten Sie, daß außerdem aus Gründen der Symmetrie für die Ergebnisanzeige statt einer *Label*-Komponente eine *Edit*-Komponente verwendet wurde. Die Eigenschaft *ReadOnly* dieser Komponente wurde auf *True* gesetzt, um ein Editieren zu verhindern, und zusätzlich wurde der Hinter-

grund grau gefärbt (*Color* wurde auf *clButtonFace* gesetzt), um das auch dem Anwender anzuzeigen. Es ist wichtig, daß der Anwender immer Informationen darüber hat, ob ein Element editierbar ist oder nicht.

Tooltips

Schließlich können Sie in der Abbildung ein weiteres Element der Benutzeroberfläche sehen, das mittlerweile zum Standard gehört: die Kurzhinweise (*Tooltips*). Sie können jeder Komponente einen eigenen Text in der Eigenschaft *Hint* zuweisen, der zur Laufzeit angezeigt werden soll, falls der Mauscursor kurze Zeit über der Komponente verweilt. Der angezeigte Kurzhinweis erscheint, wenn der Cursor über das Editierfeld plaziert wird. Um diese Anzeige zu aktivieren, müssen Sie außerdem die Eigenschaft *ShowHint* der jeweiligen Komponente auf *True* setzen. Mehr über verschiedene Möglichkeiten der Hilfeerstellung für den Anwender erfahren Sie auch in Kapitel 12.

Der nächste Punkt, den wir uns anschauen wollen, ist die Bedienung per Maus oder Tastatur. Die Bedienung per Maus ist offensichtlich kein Problem.

R 14 Tastaturbedienung ermöglichen

Um ein Bedienelement per Tastatur zu erreichen, kann der Anwender hier die ⇆-Taste benutzen, da es sich nur um vier Elemente handelt; trotzdem ist es immer gut, dem Anwender die Möglichkeit zu geben, ein Steuerelement auch direkt mit Hilfe einer Tastenkombination (eines Tastaturkürzels) zu erreichen. Dafür müssen Sie nur in der Beschriftung der einzelnen Elemente ein kaufmännisches Und (&) vor den Buchstaben setzen, der in Verbindung mit der Alt-Taste als Tastaturkürzel fungieren soll. Dieser Buchstabe wird zur Laufzeit unterstrichen dargestellt, genauso, wie Sie das bereits von den Menüoptionen kennen.

Für die Elemente, die eine Beschriftung haben, ist das kein Problem. Für Elemente ohne Beschriftung, wie das Editierfeld, gibt es ebenfalls eine einfache Möglichkeit, ein solches Tastaturkürzel zu definieren. Dazu ordnen Sie der Komponente eine *Label*-Komponente zu, die eine Beschreibung samt Tastaturkürzel als Beschriftung enthält (in unserem Fall ist das *&Ausgangswert*). Wenn Sie sich jetzt die Eigenschaft *FocusControl* der *Label*-Komponente anschauen, dann sehen Sie, daß Sie hier ein Element aus der Liste aller Komponenten des Formulars auswählen können; das ausgewählte Element (in diesem Fall das Editierfeld) wird dann automatisch aktiviert, sobald das Tastenkürzel der *Label*-Komponente betätigt wird.

Tabulatorreihenfolge

Für den Fall, daß der Anwender die ⇆-Taste zum Navigieren benutzt, ist es außerdem wichtig, daß die Tabulatorreihenfolge (also die Reihenfolge, in der die einzelnen Bedienelemente mit der ⇆-Taste aktiviert werden) der zu erwartenden Reihenfolge entspricht. Das heißt generell, daß die Reihenfolge der Aktivierung von links nach rechts und von oben

nach unten verläuft. Die konkrete Reihenfolge (zuerst vertikal oder zuerst horizontal) hängt dabei vom logischen Zusammenhang der Elemente ab. In unserem Fall heißt dies, daß nacheinander linke und rechte Optionsfelder angewählt werden (und nicht etwa umgekehrt), und danach zum Editierfeld und zur Schaltfläche SCHLIESSEN gesprungen wird. Sie können die Tabulatorreihenfolge ändern, indem Sie im Kontextmenü des Formulars die Option TABULATORREIHENFOLGE... aufrufen und in dem erscheinenden Listenfeld die Komponenten in der gewünschten Reihenfolge anordnen (siehe Abbildung 3.13). Achten Sie auch darauf, daß Sie die Eigenschaft *TabStop* des Nur-Lese-Editierfeldes auf *False* setzen, da es keinen Sinn hätte, dieses Element anzuwählen.

Abbildung 3.13: Die richtige Tabulatorreihenfolge

Sie finden das Beispiel mit der verbesserten Oberfläche unter KAPITEL 3\EINHEITEN\VERSION2. Jetzt wollen wir uns der Erstellung der vollständigen Anwendung mit verschiedenen Möglichkeiten der Umwandlung von Einheiten zuwenden.

3.3.2 Ordnung schaffen

Zunächst ist es klar, daß wir verschiedene Gruppen von Umwandlungen (also z.B. Längen oder Temperaturen) getrennt voneinander darstellen sollten. Da die Anwendung dazu dienen soll, eine gerade benötigte Umwandlung vorzunehmen, und die verschiedenen Arten von Umwandlungen nichts miteinander zu tun haben, sollten sie auch unabhängig voneinander dargestellt werden, so daß der Anwender immer nur die Informationen sieht, die er gerade benötigt (gemäß unseren Regeln für die Oberflächengestaltung). Es soll also für jede Art von Anwendung ein eigenes Fenster oder ein eigener Bereich erstellt werden, der jeweils separat

angezeigt wird. Um das zu bewerkstelligen, haben Sie mehrere Möglichkeiten, aus denen Sie diejenige heraussuchen müssen, die am besten zu Ihrer Anwendung paßt und die Ihnen am meisten zusagt; letztendlich ist die Entscheidung für ein bestimmtes Design immer auch eine Entscheidung des persönlichen Geschmacks.

R 15 Platz auf dem Bildschirm schaffen

Wir wollen Ihnen zunächst einige Möglichkeiten vorstellen, die Ihnen helfen, Platz auf dem Bildschirm zu schaffen, indem die gesamte Funktionalität in mehrere Einzeldialoge aufgeteilt wird.

Die nächstliegende (aber nicht immer die beste) Möglichkeit ist die Erstellung eines eigenen Formulars für jeden Dialog. Das ist insbesondere dann sinnvoll, wenn manchmal mehrere Dialoge gleichzeitig angezeigt werden sollen. Auch, wenn sich die Dialoge inhaltlich stark unterscheiden, können sie auf diese Art getrennt werden. Dabei werden alle Dialoge (bzw. Formulare) beim Start der Anwendung erstellt, aber immer nur diejenigen angezeigt, die gerade benötigt werden. Die Auswahl der benötigten Dialoge kann über Schaltflächen und/oder Menüoptionen erfolgen. Da von den Dialogen in unserem Beispiel immer nur einer angezeigt werden muß, ist diese Option für uns weniger geeignet. Sie werden aber noch Anwendungen kennenlernen, die mehrere Formulare benutzen.

Für die Anzeige mehrerer Dialoge innerhalb eines Formulars existieren mehrere Möglichkeiten. Vier verschiedene Komponenten sind auf diese Aufgabe spezialisiert, wobei zwei von ihnen (*TTabbedNotebook* und *TTabSet* aus dem Register WIN 3.1) nur der Abwärtskompatibilität mit Delphi1-Anwendungen dienen und für neue Anwendungen nicht verwendet werden sollten. Die gleiche oder eine ähnliche Funktionalität wird unter Windows 95/NT durch eigene Windows-Steuerelemente gewährleistet, die wie gewohnt durch Delphi-Komponenten gekapselt werden.

Dabei handelt es sich um die Komponenten *TTabControl* und *TPageControl*. Beide Komponenten ermöglichen die Erstellung von Formularen mit mehreren Seiten, die über Register erreichbar sind, also einer Art Registerkarten. Dabei arbeitet die *PageControl*-Komponente, die das entsprechende Windows-Steuerelement kapselt, mit mehreren *TabSheet*-Komponenten zusammen, die die einzelnen Seiten darstellen. Jede Seite ist also ein eigener Dialog. Diese Komponente ist insbesondere dann sinnvoll, wenn die verschiedenen Dialoge einen unterschiedlichen Aufbau und unterschiedliche Elemente haben. *PageControl*-Komponenten werden im allgemeinen für die Anzeige und Änderung von Eigenschaften und Parametern verschiedener Elemente verwendet. Sie finden in Windows selbst eine Menge Beispiele für die Verwendung des entsprechenden Steuerelements, beispielsweise den Eigenschaftsdialog im Explorer (siehe Abbildung 3.14).

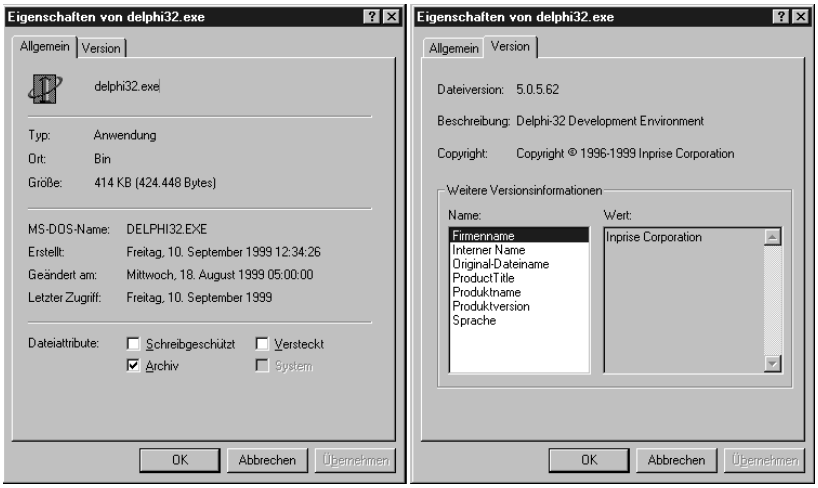

Abbildung 3.14: Die beiden Seiten eines PageControl-Steuerelements

Die zweite Komponente, die Sie verwenden können, ist die *TabControl*-Komponente. Im Gegensatz zur *PageControl*-Komponente hat diese Komponente immer nur eine Seite; der Programmierer ist dafür verantwortlich, daß beim Wechsel des Registers der Inhalt dieser Seite entsprechend angepaßt wird. Dies wird im allgemeinen dadurch geschehen, daß bestimmte Komponenten versteckt oder angezeigt werden und eventuell Beschriftungen der Elemente geändert werden. Das ist insbesondere dann sinnvoll, wenn sich die verschiedenen Dialoge wenig voneinander unterscheiden, so daß die gleichen Komponenten für verschiedene Dialoge verwendet werden können.

Da die *TTabControl* im übrigen ebenso wie *TPageControl* von *TCustomTabControl* abgeleitet ist, werden beide Komponenten relativ ähnlich behandelt, insbesondere was die Darstellung und Verwendung der Register betrifft.

Um einen solchen Fall handelt es sich offensichtlich in unserem Beispiel. Die Komponenten, die wir bereits erstellt haben, sind nicht spezifisch für die Umwandlung von Temperaturen; die *RadioGroup*-Komponenten müssen für jede Option nur entsprechend angepaßt werden. Diese Anpassung muß zur Laufzeit geschehen, wenn der Anwender das Register wechselt.

Zunächst plazieren Sie eine *TabControl*-Komponente auf dem Formular. Jetzt müssen Sie die Elemente, die Sie bisher direkt auf dem Formular plaziert hatten, auf diese Containerkomponente verschieben. Ausnehmen davon wollen wir die Schaltfläche zum Schließen. Sie ist unabhängig von der Art der Umwandlung und sollte deshalb auch außerhalb der »Register-

karten« dargestellt werden. Da Sie die Komponenten nicht direkt auf eine Containerkomponente verschieben können, wie Sie aus Rezept 1 wissen, müssen Sie die Komponenten zunächst markieren (am einfachsten geht das, indem Sie mit der Maus einen Rahmen um die Komponenten aufziehen), dann ausschneiden (Strg X) und auf der *TabControl*-Komponente wieder einfügen (Strg V).

Abbildung 3.15: Die TabControl-Komponente zur Entwurfszeit

Align Setzen Sie die Eigenschaft *Align* der *TabControl*-Komponente auf *alTop*; dadurch wird sie automatisch am oberen Rand des Formulars plaziert, und die linken und rechten Seiten werden ebenfalls an das Formular angepaßt. Die einzelnen Register fügen Sie hinzu, indem Sie die Eigenschaft *Tabs* editieren. Wie bei der *RadioGroup*-Komponente verwenden Sie dazu einen Stringlisten-Editor, wo Sie die einzelnen Registerbezeichnungen als Zeilen einfügen. Sie sehen das Ergebnis auch gleich auf dem Formular. Wenn Sie wollen, daß die einzelnen Register hervorgehoben dargestellt werden, wenn sich der Mauscursor darüber befindet, dann setzen Sie die Eigenschaft *HotTrack* auf *True*. Außerdem können Sie in den beiden *RadioGroup*-Komponenten alle Optionsfelder entfernen, da Sie diese ja jetzt erst zur Laufzeit erstellen wollen. Das Ergebnis sehen Sie in Abbildung 3.15, wobei Sie natürlich Ihre eigenen Kategorien für die Einheiten verwenden können.

Nachdem wir die Organisation der Oberfläche geklärt haben, kommen wir wieder zur Schreibarbeit: Wir müssen die Funktionalität entsprechend anpassen. Dazu erstellen wir zunächst eine eigene Prozedur, die für die Aktualisierung der Dialogseite verantwortlich ist. In dieser Prozedur (wir haben sie *UpdatePage* genannt) müssen wir die einzelnen Optionsfelder für die beiden *RadioGroup*-Komponenten erstellen. Das erledigen wir in einer langen *case*-Anweisung. Um aber innerhalb der *case*-Anweisung die Übersicht zu behandeln, verwenden wir anstatt der Zahlenwerte für die gewählte Kategorie Konstanten mit aussagekräftigeren Bezeichnungen. Am einfachsten ist das, wenn wir einen eigenen Aufzählungstypen dafür definieren:

```
TUnitKind = (ukTemp, ukLength1, ukLength2, ukWeight);
```

Jetzt müssen wir den Wert für das ausgewählte Register noch in diesen Typ konvertieren, um die Konstanten verwenden zu können:

```
with UnitRadioGroup1.Items do
begin
  Clear;
  case TUnitKind(TabControl.TabIndex) of
    ukTemp:
    begin
      Add('Grad Celsius');
      Add('Grad Fahrenheit');
      Add('Grad Kelvin');
      Add('Grad Réaumur');
    end;
    ukLength1:
    begin
      Add('Millimeter');
      Add('Zentimeter');
      Add('Zoll');
      ...
```

Die Eigenschaft *Items* ist vom Typ *TStrings*, der eine Stringliste verwaltet; wir benennen hier nur die verwendeten Methoden, mehr dazu werden Sie im nächsten Kapitel lesen. Zu Beginn werden mit Hilfe der Methode *Clear* alle Elemente gelöscht; danach werden, abhängig vom gerade aktiven Register, alle gewünschten Optionsfelder mit Hilfe der Methode *Add* erstellt. Die lange *case*-Anweisung ist vielleicht nicht sehr elegant, erfüllt aber ihren Zweck. Für die zweite *RadioGroup*-Komponente werden dann einfach alle Elemente der Eigenschaft *Items* von der ersten übernommen:

```
  UnitRadioGroup2.Items.Clear;
  for i := 0 to UnitRadioGroup1.Items.Count - 1 do
  UnitRadioGroup2.Items.Add(UnitRadioGroup1.Items[i]);
```

Dadurch vermeiden wir eine zweite *case*-Anweisung. Schließlich müssen wir auch noch die Eigenschaft *ItemIndex* der beiden Komponenten wieder auf Null setzen, wie wir das vorher mit Hilfe des Objektinspektors gemacht haben:

```
  UnitRadioGroup1.ItemIndex := 0;
  UnitRadioGroup2.ItemIndex := 0;
```

Jetzt müssen wir unsere Prozedur noch an den richtigen Stellen aufrufen: zu Programmbeginn und beim Wechsel des Registers. Sie wissen bereits, daß für die Darstellung zu Programmbeginn das Ereignis *OnCreate* des Formulars am besten geeignet ist. Da wir die Ereignisbehandlung dafür auf eine andere Prozedur gesetzt haben (auf *NumEditChange*), müssen wir zunächst diese Verbindung wieder lösen. Das geschieht einfach durch Löschen des entsprechenden Eintrags im Objektinspektor. Danach können Sie mit einem Doppelklick in die jetzt leere Spalte wieder ein neues Ereignis erzeugen:

```
procedure TConversionForm.FormCreate(Sender: TObject);
begin
  UpdatePage;
  Convert;
end;
```

Den Aufruf von *Convert*, der in *NumEditChange* steht, müssen Sie jetzt hier ebenfalls ergänzen. Schließlich gibt es auch ein Ereignis der *TabControl*-Komponente, das nach jedem Wechseln des Registers aufgerufen wird: *OnChange*. Auch hierfür müssen Sie eine Ereignisbehandlung erstellen (dafür reicht ein Doppelklick auf die Komponente) und die Prozedur *UpdatePage* eintragen.

Damit ist die Aktualisierung der Darstellung abgeschlossen. Jetzt bleibt noch die eigentliche Umwandlung: die Prozedur *Convert*. Auch hier verwenden wir eine *case*-Anweisung für die Unterscheidung der einzelnen Kategorien. Außerdem benutzen wir innerhalb jedes *case*-Zweigs zwei weitere *case*-Anweisungen, um, wie für die Temperatur bereits gezeigt, den Wert zunächst in eine Zwischeneinheit (wir verwenden dafür immer die erste angezeigte Einheit) und von da in die gewünschte Einheit umzuwandeln. Hier ist ein Ausschnitt aus dieser jetzt recht umfangreichen Prozedur:

```
value := StrToFloat(NumEdit.Text);
case TUnitKind(TabControl.TabIndex) of
  ukTemp:
  begin
    case UnitRadioGroup1.ItemIndex of
      1: value := ((value - 32.0) * 5) / 9; // Fahrenheit
      2: value := value - 273.15;           // Kelvin
      3: value := 1.25 * value;             // Réaumur
    end;
    case UnitRadioGroup2.ItemIndex of
      ...
    end;
  ukLength1:
  begin
    case UnitRadioGroup1.ItemIndex of
      1: value := 10 * value;          // Zentimeter
      2: value := 25.4 * value;        // Zoll
      3: value := 304.8 * value;       // Fuß
      ...
```

Wie Sie sehen, haben wir in den inneren *case*-Anweisungen keine Konstanten verwendet; in diesem Fall ist aber eine entsprechende Kommentierung unerläßlich.

Wenn Sie den Code richtig erstellt haben, ist Ihre Anwendung fast fertig. Wenn Sie jetzt allerdings starten, bekommen Sie gleich eine Fehlermeldung über einen Index außerhalb des Wertebereichs. Jetzt haben Sie die Gelegenheit, das anzuwenden, was Sie über das Debuggen gelernt haben.

Aufgabe 2

Finden Sie mit Hilfe von Haltepunkten, des Aufruf-Stacks und der Untersuchung von Variableninhalten heraus, was passiert ist.

In Kapitel 9, wo es um die Fehlerbehandlung geht, kommen wir noch einmal darauf zurück; hier nur die Lösung des Problems zu Beginn der Prozedur *Convert*:

```
if (UnitRadioGroup1.ItemIndex < 0) or
   (UnitRadioGroup2.ItemIndex < 0) then Exit;
```

Jetzt sollte Ihre Anwendung anstandslos funktionieren.

3.3.3 Weitere Komponenten

Zum Schluß wollen wir noch eine kleine Ergänzung anfügen, die der Konfiguration des Programms der Anwendung, in erster Linie allerdings der Einführung einer weiteren einfachen Komponente dient: der *CheckBox*-Komponente. Wir wollen dem Anwender die Möglichkeit geben, die Kurzhinweise an- und abzuschalten, die ja vielleicht störend wirken könnten (da es in der Anwendung sonst nicht allzuviel zu konfigurieren gibt).

Dazu plazieren Sie eine *CheckBox*-Komponente aus dem Register STANDARD auf dem Formular und geben ihr eine entsprechende Beschriftung. Die Komponente kapselt, ähnlich wie *TRadioButton*, ein einfaches Windows-Steuerelement: ein Kontrollfeld, das markiert sein kann. Im Gegensatz zur *RadioButton*-Komponente ist jedes Kontrollfeld eigenständig und hängt nicht vom Zustand anderer Kontrollfelder ab. Die Behandlung solcher Kontrollfelder ist recht einfach: Sie müssen meist nur das Ereignis *OnClick* behandeln und dann in Abhängigkeit vom aktuellen Zustand des Kontrollfeldes die entsprechende Aktion ausführen. Hier ist die Ereignisbehandlung für unser Beispiel:

```
procedure TConversionForm.HintCheckBoxClick(Sender: TObject);
var
  show: Boolean;
begin
  show := HintCheckBox.Checked;
  UnitRadioGroup1.ShowHint := show;
  UnitRadioGroup2.ShowHint := show;
  NumEdit.ShowHint := show;
  ResultEdit.ShowHint := show;
end;
```

Wir benutzen hier die Hilfsvariable *show*, um nicht jedesmal die Eigenschaft *Checked* abfragen zu müssen. Hier wird vorausgesetzt, daß im Anfangszustand (also zur Entwurfszeit) das Kontrollfeld markiert ist, da die Kurzhinweise eingeschaltet sind. Wenn Sie viele Elemente mit Kurzhinweis in Ihrem Formular haben, ist die gezeigte Methode allerdings etwas unbequem, da Sie jede einzelne Komponente hier aufführen müssen. Eine Möglichkeit, sämtliche Komponenten des Formulars zu durchlaufen, werden Sie in Rezept 32 kennenlernen; auf die gleiche Art könnten Sie auch dieses Beispiel verkürzen.

Schließlich wollen wir das Kontrollfeld, das der Konfiguration der Anwendung dient, auch optisch von dem inhaltlichen Teil abgrenzen. Die einfachste Möglichkeit dafür ist die Verwendung der Komponente *Bevel* aus dem Register ZUSÄTZLICH; hierbei handelt es sich um ein einfaches

grafisches Element, das nur der optischen Verschönerung dient. In Abhängigkeit von der Eigenschaft *Shape* kann es sich dabei um eine einzelne Trennlinie oder um verschieden gestaltete Rahmen handeln. Die Eigenschaft *Style* gibt außerdem an, ob die Fläche oder die Linie als abgesenkt (*bsLowered*) oder herausgehoben (*bsRaised*) erscheint. In Abbildung 3.16 sehen Sie den Effekt, der wie eine abgesenkte Fläche wirkt.

Abbildung 3.16: Die fertige Anwendung zur Laufzeit

Sie können statt der *Bevel*-Komponente auch die *GroupBox*-Komponente verwenden, die das entsprechende Windows-Steuerelement kapselt; das ist aber im allgemeinen nicht notwendig. Die *GroupBox*-Komponente ist im Gegensatz zur *Bevel*-Komponente ein echtes Windows-Fenster und wirkt als Containerkomponente, was z.B. zum Zusammenfassen von *RadioButton*-Komponenten nützlich sein kann, falls Sie aus irgendeinem Grund nicht die *RadioGroup*-Komponente nutzen können. Wenn Sie die Wahl haben, sollten Sie aber immer die *Bevel*-Komponente verwenden, da sie flexibler ist und weniger Windows-Ressourcen benötigt.

Aufgabe 3

Modifizieren Sie die Anwendung so, daß die Register links (mit der Schrift um 90 Grad gedreht) angezeigt werden.

Aufgabe 4

Modifizieren Sie die Anwendung so, daß die Auswahl nicht über eine *TabControl*-Komponente, sondern über ein lokales Menü (*PopupMenu*-Komponente) erfolgt (das ist eine fortgeschrittene Aufgabe, die eventuell einiges Probieren und Studium der Online-Hilfe erfordert).

Damit wollen wir unsere kleine Beispielanwendung abschließen. Sie können die Anwendung bei Bedarf natürlich für beliebige Einheiten erweitern. Sie finden die Anwendung in der Endversion auf der CD-ROM im Verzeichnis KAPITEL 3\EINHEITEN\VERSION3. Bevor wir dieses Kapitel abschließen, wollen wir noch ein Rezept anfügen, das Ihnen eine Art Checkliste für den Abschluß einer größeren Anwendung liefert.

R 16 Erstellen der Endversion

Bisher ging es vor allem darum, ein Programm zum Laufen zu bringen und zu testen. Irgendwann kommt allerdings einmal der Punkt, wo das Programm als fertig anzusehen ist und zur Verwendung »freigegeben« wird – sei es nur zur eigenen Verwendung oder als Freeware- oder Shareware-Programm im Internet. Wir schreiben hier absichtlich »als fertig anzusehen ist«, weil es wohl keine nicht-triviale Anwendung gibt, die völlig fehlerfrei ist. Die Frage ist nicht, ob die Anwendung Fehler enthält, sondern, wie viele Fehler sie enthält. Eine brauchbare Anwendung sollte nur unter sehr seltenen Umständen ein fehlerhaftes Verhalten zeigen (und nicht den Anwender zum Beta-Tester machen, wie es leider oft genug geschieht).

Bevor Sie eine fertige Version verwenden oder weitergeben, sollten Sie darauf achten, daß die Anwendung nichts mehr enthält, was nur zum Testen der Anwendung notwendig war. Das ist unter Delphi nicht besonders schwer: Sie müssen eventuell einige Optionen ändern und das Projekt einfach neu kompilieren. Folgende Punkte sollten Sie dabei beachten:

- Stellen Sie die Optimierung an. Während beim Testen die Optimierung oft störend wirkt, da sie z.B. die Anzeige »wegoptimierter« Variablen verhindert, ist die Optimierung in der Endversion ein Muß, um Ausführungszeit und -code zu sparen. Sie finden die Option im Dialog PROJEKTOPTIONEN (PROJEKT | OPTIONEN...) auf der Seite COMPILER bei CODEERZEUGUNG.
- Auf der gleichen Seite der Projektoptionen finden Sie auch die Rubrik DEBUGGEN; diese Rubrik enthält Optionen, die nur beim Debuggen notwendig sind. Die ersten drei Punkte sorgen dafür, daß Sie während des Testens Zugriff auf alle benötigten Informationen (Symbolnamen, Zeilennummern usw.) haben; diese Informationen sind in der Endversion natürlich überflüssig. Das gleiche gilt für die Option »Assertion«; sie bezieht sich auf die *Assert*-Funktion, mit deren Hilfe zusätzliche Prüfungen vorgenommen werden können (siehe Kapitel 9).

- Schließlich finden Sie, ebenfalls auf dieser Seite, die Rubrik LAUFZEITFEHLER. Hier wird angekreuzt, welche Prüfungen zur Laufzeit vorgenommen werden sollen. Sie sollten die beiden Optionen »Bereichsüberprüfung« und »Überlaufprüfung« deaktivieren. Diese Optionen bewirken, daß Kontrollen auf bestimmte Bereichsüberschreitungen durchgeführt werden. Solche Bereichsüberschreitungen deuten auf Programmierfehler hin, die in der Endversion nicht mehr vorkommen sollten. Die Option »I/O-Prüfung« sollten Sie dagegen eingeschaltet lassen, da Eingabe-/Ausgabefehler auch in der Endversion immer vorkommen können (z.B. durch nicht eingelegte Disketten, volle Festplatten oder schreibgeschützte Dateien).
- Auf der Seite LINKER sollte die Option MAP-DATEI auf »Aus« stehen; außerdem sollte unbedingt die Option »Mit Debug-Info für TD32« ausgeschaltet sein. Beide Einstellungen sind Standard, so daß Sie üblicherweise hier nichts ändern müssen.

Soviel zu den verschiedenen Optionen beim Kompilieren. Für den Fall, daß Sie Ihre Anwendung weitergeben wollen, hier noch ein paar weitere Hinweise:

- Vergessen Sie nicht, eine Hilfe mitzugeben. Das absolute Mindestmaß an Hilfe ist eine mitgelieferte README-Datei, besser ist eine eigene Hilfedatei (siehe Kapitel 12). Denken Sie daran, daß Sie auch eine englische Hilfe mitliefern, falls Sie die Datei ins Internet stellen (und wenn es der Hinweis ist, daß die Anwendung nur für deutsche Anwender geeignet ist, was z.B. bei bestimmten Finanzprogrammen der Fall sein kann).
- Fügen Sie eine Versionsinformation in die Datei ein. Dazu kreuzen Sie auf der Seite VERSIONSINFO der Projektoptionen die Option »Versionsinfo ins Projekt übernehmen« an und füllen die entsprechenden Felder aus. Die Information wird z.B. bei der Schnellansicht der ausführbaren Datei angezeigt; außerdem kann sie später der Versionsüberprüfung dienen, wenn eine neuere Version mit Hilfe eines Installationsprogramms installiert werden soll.
- Geben Sie der Anwendung einen eigenen »Stempel«. Geben Sie der ausführbaren Datei einen einprägsamen Namen (verwenden Sie auf keinen Fall den Vorgabenamen PROJECT1.EXE!), und erstellen Sie ein eigenes Icon für die Anwendung. Das Icon können Sie in den Projektoptionen auf der Seite ANWENDUNG ändern. Übliche Praxis ist es auch, die Zeit der Dateierstellung auf die aktuelle Versionsnummer zu setzen; Sie können dazu z.B. die Anwendung im Verzeichnis KAPITEL 8\ZEITSTEMPEL auf der CD-ROM benutzen, die am Ende von Kapitel 8 noch näher erläutert wird.

- Überzeugen Sie sich, daß Ihre Anwendung auch auf einem »jungfräulichen« Computer (also ohne installiertes Delphi, möglichst auch ohne andere Programme) ordnungsgemäß läuft. Achten Sie darauf, alle notwendigen Dateien mitzuliefern (inklusive Packages – siehe Kapitel 11). Falls möglich, erstellen Sie mit Hilfe des in der Profiversion mitgelieferten *InstallShield* eine Installationsroutine für Ihre Anwendung.

Zum Abschluß noch eine Bemerkung zur »großen« Softwareentwicklung. Es gibt in der professionellen Softwareentwicklung mehr oder weniger etablierte Methoden und Ansätze zur Ablaufsteuerung und -kontrolle.

CASE-Tools

Bei großen Projekten spielen solche Aspekte wie die Kontrollierbarkeit der Ergebnisse einzelner Entwicklungsphasen, die reibungslose Zusammenarbeit größerer Entwicklungsteams oder eine einfach zu handhabende Versionsverwaltung eine Rolle. Dafür existieren spezielle Werkzeuge, sogenannte CASE-Tools (von *Computer Aided Software Engineering*), die es für die strukturierte und seit einiger Zeit auch für die objektorientierte Programmentwicklung gibt. Delphi stellt in der Enterprise-Version von Delphi ebenfalls Werkzeuge dafür zur Verfügung – insbesondere *Teamsource*, das die Softwareentwicklung in größeren Gruppen unterstützt.

Das alles brauchen Sie für die Entwicklung kleiner und mittlerer Ein-Mann-Delphi-Anwendungen nicht. Sie sollten aber trotzdem auch in Ihren kleinen Programmen darauf achten, daß Sie nicht einfach wild drauflosprogrammieren, sondern sich zuerst wenigstens ein einfaches Konzept machen (möglichst nicht nur im Kopf) und das Hauptziel nicht aus den Augen verlieren, das Sie mit einer Anwendung erreichen wollen.

3.4 Lösungen

Aufgabe 1

Sie setzen den Haltepunkt in die Zeile:

```
ResultLabel.Caption := FloatToStr(temp);
```

Dann lassen Sie sich im Kontextmenü des Haltepunktes dessen Eigenschaften anzeigen und klicken auf WEITERE.... Dort wählen Sie die Aktion »Anhalten« ab, schreiben Ihre Meldung in das Editierfeld »Meldung protokollieren« und die Variable `temp` in das Feld »Eval-Ausdruck«. Dann kreuzen Sie noch die Option »Ergebnis protokollieren« an, falls sie nicht bereits gewählt ist, und lassen die Anwendung laufen. Wenn Sie ANSICHT | DEBUG-FENSTER | EREIGNISPROTOKOLL wählen, sehen Sie das Ergebnis.

Aufgabe 2

Das Problem tritt in der Methode *Convert* auf. Wenn Sie den Aufruf-Stack nutzen, sehen Sie, daß dieser Aufruf von der Methode *NumEditChange* kommt, noch bevor der Aufruf von *Convert* in *FormCreate* erfolgt. Sie können jetzt auf verschiedene Art vorgehen. Wenn Sie weitere Haltepunkte setzen, werden Sie sehen, daß der Aufruf innerhalb von *UpdatePage* erfolgt. Durch schrittweise Ausführung dieser Methode finden Sie, daß die Zeile

```
UnitRadioGroup1.ItemIndex := 0;
```

diesen Aufruf hervorruft. Mehr dazu in Kapitel 9, Rezept 68, im Abschnitt *Seiteneffekte durch Setzen von Eigenschaften*.

Aufgabe 3

Die Lösung ist im Prinzip simpel: Sie setzen die Eigenschaft *TabPosition* der *TabControl*-Komponente auf *tpLeft*. Allerdings wird in diesem Fall die Schrift nicht richtig angezeigt. Die Lösung finden Sie in der Online-Hilfe: Sie müssen eine True Type-Schriftart für die Komponente (also z.B. »Arial«) benutzen, damit es funktioniert.

Aufgabe 4

Um nach dem Entfernen der *TabControl*-Komponente nicht alle Komponenten neu positionieren zu müssen, markieren Sie am besten alle Komponenten, die sich darauf befinden, und kopieren Sie in die Zwischenablage. Danach können Sie die *TabControl*-Komponente entfernen und die Komponenten aus der Zwischenablage wieder einfügen und dadurch direkt auf dem Formular plazieren. Die *PopupMenu*-Komponente passen Sie im Menü-Designer an, indem Sie die verschiedenen Kategorien von Einheiten eintragen und die Eigenschaft *RadioItem* aller Einträge auf *True* setzen. Um die Ereignisbehandlung zu erstellen, markieren Sie am einfachsten alle Menüeinträge und doppelklicken dann auf der Ereignisseite des Objektinspektors. Sie können dann von dort die Methode *UpdatePage* aufrufen, die Sie entsprechend anpassen müssen. Für den Code verweisen wir Sie auf den Quelltext auf der CD-ROM.