Einstieg in die Objektorientierte Programmierung (OOP) mit Delphi

Warum steigst du in Delphi ein? Wahrscheinlich, weil dich einer der Vorzüge von Delphi reizt. Ich zähle ein paar auf:

- Mit Delphi erstellt man sehr schnell professionelle und effiziente Windows-Programme für alle denkbaren Zwecke.
- Delphi ist eine objektorientierte Programmiersprache, die dem Programmierer in der *Visual Component Library (VCL)* eine riesige Auswahl an Objektklassen („*Komponenten*“) zur Verfügung stellt. Im Internet findet man zusätzlich hunderte Komponenten.
- Die Sprachkonventionen beruhen auf denen von Objekt-Pascal. Die elementaren und selbstdefinierten Typen, die Syntax, Funktionen, Prozeduren: alles wie in Turbo-Pascal.
- Delphi kann dank Kylix auch plattformübergreifend unter Linux eingesetzt werden.
- Delphi ist **die** Programmiersprache zum Erstellen von Datenbanken.
- Delphi ist schneller als Java und etwas langsamer als C++, aber leichter zu erlernen als letzteres.

Am Ende dieser Einführung wirst du sicherer sein, ob du dich vertieft mit Delphi befassen willst. **Momentan setze ich voraus, dass du schon mal programmiert hast**, d.h. der Umgang mit Variablen, Typen, Schleifen, Verzweigungen, Funktionen, Prozeduren sollte dir nicht neu sein. Wenn dir darüber hinaus die Begriffe *Objekt* und *Klasse* geläufig sind, kannst du Kapitel 1 überspringen.

1. Objekte und Klassen

In der „realen“ Welt beschreiben wir gleichartige Objekte mit demselben Begriff: Stuhl, Auto, Schule, Fenster, Person. In der OOP nennt man solche Begriffe **Klassen**. *Stuhl* ist etwas anderes als *Auto* oder *Schule*. Die Klasse *Stuhl* hat (je nach Sichtweise) die Eigenschaften Material, Zahl der Beine, Höhe. Die **Klasse *Auto*** die Eigenschaften Modell, Farbe, PS-Zahl, Ausstattung. Die Klasse *Schule* die Eigenschaften Schulname, Schülerzahl, Schüler, Hausmeister und Direktor.

Die Klasse *Stuhl* ist noch kein reales Objekt, ebensowenig wie die Klasse *Auto*. Ich kann aber ins Autohaus gehen und einen Bestellschein ausfüllen, dann wird mein Traum sehr schnell wahr und steht als **Objekt *meinAuto*** vor der Tür.

Die Klasse ist der Bauplan für ein oder mehrere Objekte dieser Klasse.

Bestellschein für *meinAuto*

```
meinAuto : Auto
meinAuto.Modell := 'VW Golf'
meinAuto.PSZahl := 90
meinAuto.Ausstattung := (CD-Spieler,
                         Schiebedach)
meinAuto.Farbe := rot
```

Die Schreibweise meines Bestellscheins habe ich bereits der Schreibweise in den Programmiersprachen angepasst: *meinAuto* ist eine **Objekt-Variable**, die im Computerspeicher einen Zeiger auf ein Objekt vom Typ *Auto* darstellt. Sie ist dazu da, das Objekt anzusprechen. Den Punkt zwischen ‚meinAuto‘ und ‚Modell‘ nennt man Verweis-Operator. *Modell* ist eine **Eigenschaft** der Klasse *Auto*. In *meinAuto* wird dieser Eigenschaft *Modell* der **Eigenschafts-Wert** „VW Golf“ zugewiesen.

Einfache Objekt-Klassen kommen allein mit Eigenschaften (Attributen, Daten) aus, wie etwa die Klasse *Stuhl*. Objekte von solchen Klassen werden oft von anderen, komplexeren Objekten (z.B. Herr Hafner) benutzt, oder werden in andere Klassen eingefügt (Klasse *Wohnung*, Klasse *Schulzimmer* oder auch Klasse *Auto*). Bei komplexeren Klassen erwarten wir aber nicht nur das Vorhandensein von Eigenschaften, sondern auch, dass ein Objekt dieser Klasse eigene Operationen ausführen kann. Von einem Auto erwarten wir, dass es Operationen besitzt, mit denen es gestartet, beschleunigt, gelenkt und gebremst werden kann. Diese Operationen innerhalb einer Objekt-Klasse nennt man in der OOP **Methoden**.

Auto
<<Eigenschaften>>
Modell: string
Farbe: Farbtyp
PSZahl: integer
... (und viele andere)
<<Methoden>>
starten
bremsen
beschleunigen
lenken
... (und viele andere)

Klassen und Objekte haben also Eigenschaften und meist auch Methoden.

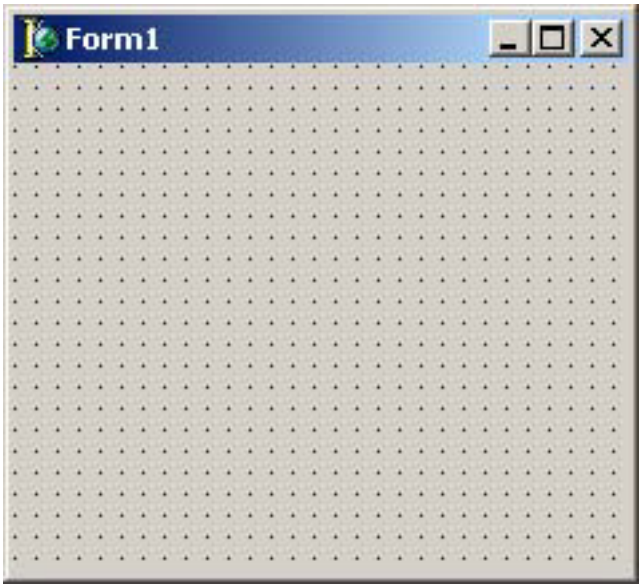
In einem **UML-Diagramm** würde unsere Klasse *Auto* etwa wie rechts dargestellt werden. (UML=Unified Modeling Language, eine standardisierte Modellierungs-Sprache)

Objekte einer Klasse besitzen die gleichen Methoden und Eigenschaften. Sie unterscheiden sich nur in den Eigenschaftswerten.

Ein wichtiges Grundkonzept der OOP ist die **Vererbung**. Wer schon mal einen Bauplan Klasse *Auto* hat, der braucht nicht wieder bei Adam und Eva anzufangen, wenn er einen Lieferwagen für einen Eislieferanten bauen soll. Er wird Klasse *Auto* verändern, indem er ein paar Eigenschaften und Methoden hinzufügt (z.B. *Ladefläche*, *Laderaum kühlen*), andere Eigenschaftswerte festlegt (z.B. *Heckspoiler* = nein), oder eine solche Eigenschaft versteckt (damit ein Kunde gar nicht auf die Idee kommt, einen Spoiler mitzubestellen). Eventuell muss er auch einige Methoden überschreiben (das Bremsen erfordert sicher den Einsatz eines größeren Bremskraftverstärkers). Ansonsten hat *Lieferwagen* alles, was *Auto* auch hat. Gleich bei deiner allerersten Delphi-Anwendung wirst du mit Vererbung konfrontiert werden.

Wenn nun eine Programmier-Umgebung wie Delphi dem Programmierer eine ganze Bibliothek von Klassen zur Verfügung stellt, geht das Programmieren von Objekten so schnell von der Hand wie das Bestellen von Autos: man muss sie nicht erst entwickeln, sondern man erzeugt sie einfach. Und die Klassen, die es noch nicht gibt, vererbt man meist aus vorhandenen.

2. Dein Delphi-Start



Wenn du die **Delphi-IDE** (integrierte Entwicklungsumgebung) startest mit Datei/Neu/Anwendung, zeigt sie dir ein leeres Formular namens „Form1“. Dieses Hauptfenster stellt bereits eine fertig funktionierende Anwendung dar. Wenn man sie startet, sieht das Fenster so aus wie vorher zur Entwicklungszeit, trägt die Überschrift „Form1“ und reagiert auf Mausereignisse so, wie man es von Fenstern erwartet: Man kann seine Lage und Größe verändern und man kann es schließen. Delphi hat den Programm-Code dazu selber erzeugt in mehreren Dateien, die alle zu diesem *Projekt1* gehören.

Das Hauptfenster stellt ein Objekt einer **Klasse *TForm*** dar. (Delphi-Klassen erkennt man immer an dem vorangestellten ‚T‘.) Die Typ-Deklaration dieses Hauptfensters steht in der **Unit1.pas**, auf die du von der Formularansicht mit der Taste F12 umschalten kannst. Du siehst, dass von einer Klasse *TForm* eine **Klasse *TForm1* vererbt** wird, deren Eigenschaften und Methoden wir im Weiteren verändern werden. Und wir werden ihr Objekte anderer Klassen hinzufügen. Damit von der Klasse *TForm1* beim Start des Programms ein Fenster-Objekt erzeugt werden kann, braucht man eine **Objekt-Variable**, das ist ein Zeiger, der hier *Form1* heißt (nicht zu verwechseln mit dem gleichnamigen Titel unseres Hauptfensters). Über diesen Zeiger hat man dann Zugriff auf die Eigenschaften und Methoden des Objekts.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs;
type
  TForm1 = class(TForm)
  private
   { Private-Deklarationen }
  public
   { Public-Deklarationen }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
end.
```

```
program Project1;
uses
  Forms, Unit1 in ‚Unit1.pas‘ {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Mit dem Hauptprogramm hat man beim Programmieren einfacherer Anwendungen eigentlich nie zu tun. Dieses steht deswegen in einer eigenen Projektdatei ***Projekt1.dpr***. Dort wird das Objekt *Form1* der Klasse *TForm1* erschaffen und das Programm ausgeführt.

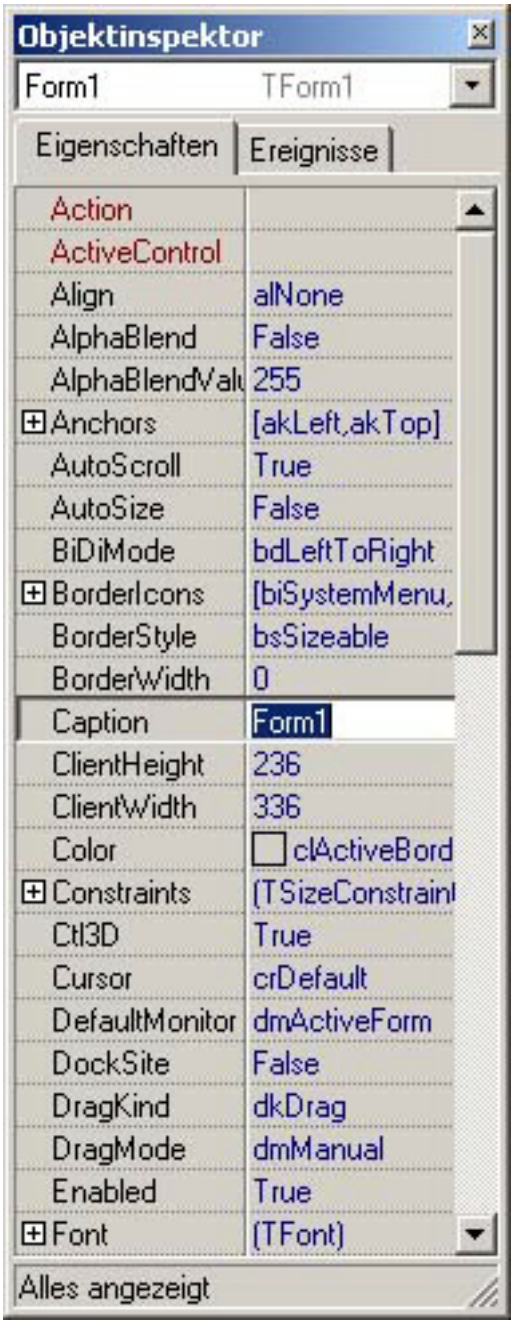
3. Unsere erste Delphi-Anwendung

Programmieren in Delphi bedeutet:

1. Hinzufügen von Komponenten (= Objekte von vorgegebenen Klassen) aus den Komponentenpaletten
2. Einstellen der Komponenten-Eigenschaften
3. Schreiben der Ereignis-Methoden (in Objekt-Pascal)
4. Starten der Anwendung (Test, Fehlersuche ...)

Erst in einem fortgeschrittenen Stadium wirst du selbst Klassen definieren und sie vielleicht sogar als Komponenten in die IDE integrieren. Ein Beispiel findest du am Ende dieses Lehrgangs.

TForm1: TForm
<<Eigenschaften>>
Caption: string
Color: Tcolor
Width: integer
... (und viele andere)
onclick (Ereignis)
... (und viele andere)
<<Methoden>>
create
repaint
... (und viele andere)



Bis jetzt enthält unsere Delphi-Anwendung nur ein Objekt: das Hauptformular *Form1*. Dieses besitzt bereits Eigenschaften, die wir verändern können, Methoden, die wir aufrufen können und Ereignisse (das sind spezielle Eigenschaften), für die wir eigene Ereignis-Methoden schreiben, sodass das Fenster auf diese Ereignisse nach unserem Wunsch reagiert. Der **Objekt-Inspektor** gibt uns Auskunft über den Inhalt der Eigenschaften zur Entwicklungszeit. Die Eigenschaft *Caption* etwa enthält den String ‚Form1', den wir durch z.B. „mein erstes Programm" ersetzen können. Der Eigenschaft *Color* können wir irgendeine Farbe zuweisen, die Fensterbreite können wir mit *Width* einstellen usw.. Vielleicht wundert dich, dass von dieser Änderung im Quellcode der *Unit1.pas* nichts zu sehen ist. Diese Voreinstellungen zur Entwicklungszeit speichert Delphi in einer eigenen Formular-Datei (*.dfm), die wir nie aktiv benötigen. Man hat dies ja viel übersichtlicher mit dem Objekt-Inspektor im Griff als mit geschriebenem Quellcode.

Delphi-Anwendungen sind ereignis-orientiert: Jeder wichtige Vorgang einer Windows-Anwendung muss durch etwas ausgelöst werden. Diese Auslöser heißen Ereignisse: ein Klick auf einen Button, das Drücken der Eingabe-Taste, das Auswählen eines Menü-Eintrags, das Schließen eines Fensters, ein Timer-Ereignis (mit dem z.B. regelmäßig eine Datei gespeichert wird) und viele andere. Unser *Form1* verfügt u.a. über ein ***onclick*-Ereignis**, das eintritt, wenn mit der Maus auf die Fensterfläche geklickt wird. Ein großer Vorteil von Delphi ist, dass man die Standard-Ereignisse, die ein Objekt auslöst, visuell im Objekt-Inspektor mit einer selbstgeschriebenen Methode verknüpfen kann. (Das erspart uns die Mühe, ständig neue Klassen zu

vererben und virtuelle Methoden zu überschreiben.) Und das wollen wir gleich ausprobieren: Klicke im Inspektor auf Ereignisse, dann auf *onclick*, mache einen Doppelklick in das Eingabefeld daneben - und staune: Delphi hat in der *Unit1.pas* der Klasse *TForm1* eine Methode hinzugefügt:

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

und das sowohl im *Interface*-Teil, wo *TForm1* deklariert wird, als auch im *Implementation*-Teil, wo der eigentliche Code steht. Das einzige, was du noch dazuschreiben musst, ist das, was der Mausklick bewirken soll: z.B. soll das Fenster seine Farbe ändern:

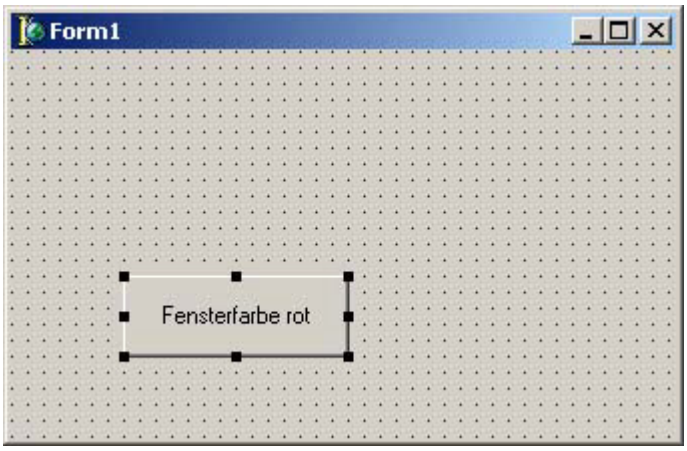
```
Form1.Color : = clred;
```

Eine weitere große Erleichterung hast du vielleicht gerade selbst bemerkt: Sobald du beim Tippen der Objektvariablen „Form1." an dem Punkt (dem Verweis-Operator) angekommen bist, klappt ein **PulldownMenü** auf, das dir alle sichtbaren (→ public, sh. Kap. 9) Eigenschaften und Methoden zur Auswahl anbietet. Man braucht sich also die Eigenschaften der Objekte nicht auswendig zu merken. Dieses Kontextmenü klappt nur dann auf, wenn bis zu dieser Stelle alles korrekt geschrieben ist. Darüber hinaus brauchst du noch die **Delphi-Hilfe**: Wie waren noch mal die Farben definiert? Kein Problem: setze die Schreibmarke auf *clred*, oder klicke im Objekt-Inspektor auf die Eigenschaft *Color* und drücke F1.

TForm1: TForm
<<Eigenschaften>>
Caption: string
Color: Tcolor
Width: integer
... (und viele andere)
onclick (Ereignis)
... (und viele andere)
<<Methoden>>
create
repaint
formclick
... (und viele andere)

Unser Objekt besitzt jetzt eine zusätzliche Methode *FormClick*, die dann automatisch aufgerufen wird, wenn das *onclick*-Ereignis eintritt. Im Inspektor kannst du nämlich sehen, mit welcher Methode das Ereignis *onclick* verknüpft ist: *formclick*. Und wie du auch siehst, unterscheidet man Ereignisse von „gewöhnlichen" Eigenschaften durch die Vorsilbe *on*. Noch ein Wort zu den Methoden des Objekts: *create* wird beim Start der Anwendung von der Methode createform von *TApplication* aufgerufen (s.h. Kasten program Project1 auf S.3), *repaint* wird von *Form1* selber bei vielen Gelegenheiten aufgerufen, z.B. beim Setzen der Eigenschaft *Color*, ebenfalls unsere Methode *FormClick* bei Eintritt des Ereignisses *onclick*. Du siehst an diesen Beispielen: Wenn es nicht einen besonderen Grund gibt, überlassen wir den Aufruf der Methoden den Objekten selber.

Hinzufügen eines Objekts der Klasse *TButton*



Unsere Delphi-Anwendung kann noch nicht viel Neues, außer einen Anwender erschrecken durch die unerwartete Änderung der Fensterfarbe. Dieser würde doch lieber auf einen Button „Fensterfarbe rot" klicken, wenn überhaupt. Aus der Komponentenpalette „Standard" holen wir einen Button ins Formular. In diesem Moment hat Delphi für uns in der *Unit1.pas* der Klasse *TForm1* eine Objektvariable *Button1* für ein Objekt der Klasse *TButton* hinzugefügt. Zwei Dinge bleiben dir zu tun: Setze die Eigenschaft *Caption* auf „Fensterfarbe rot" und verbinde das *onclick*-Ereignis von *Button1* mit einer Methode: Wenn der Button in deinem Formular markiert ist, zeigt der Inspektor die Eigenschaften von *Button1* an (ansonsten die von *Form1*). Ein Doppelklick in das Feld bei *onclick* erzeugt die Methode:

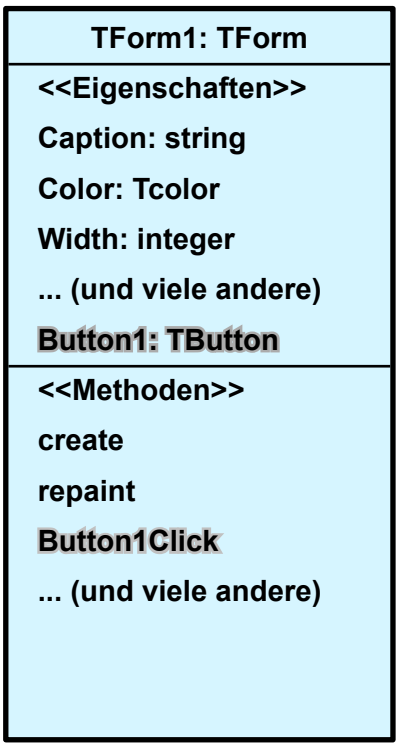
procedure TForm1.Button1Click(Sender : TObject);

Ergänze sie zwischen *begin* und *end* mit

Form1.Color := clred;

Die alte Methode *FormClick* ist dann überflüssig. Lösche nur die Anweisung zwischen *begin* und *end*. Das ganze Gerüst der Methode entfernt Delphi beim nächsten Programm-Start.

Button1Click ist eine Methode des Objekts *Form1* (nicht von *Button1*), aber sie ist verknüpft mit dem Ereignis *onclick* von *Button1*. Die Methode *Form1.Button1Click* kann darauf reagieren, weil *Form1* **Besitzer** des Objekts *Button1* ist. UML-Diagramm und Quellcode verdeutlichen dir die Verhältnisse nochmals.



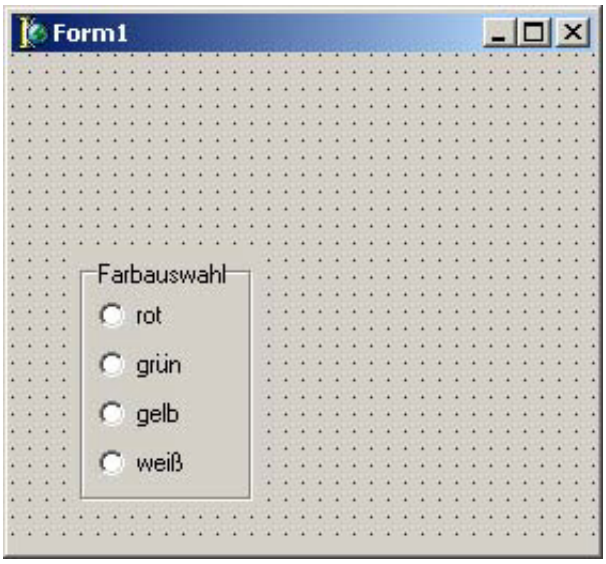

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants,
  Classes, Graphics, Controls, Forms,  Dia-
  logs;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TOb-
      ject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender:
 TObject);
begin
  Form1.Color:=clred;
end;
end.
```

Was Delphi-Anfänger immer verwirrt:

Was muss ich speichern?

Beim ersten Erstellen einer neuen Anwendung wähle „Projekt speichern unter ...“. Daraufhin öffnen sich zwei Speichern-Dialogfenster: Als erstes wird die *Unit1.pas* gespeichert. Wenn du hier einen anderen Namen für die Unit wählen willst, musst du vorher in der ersten Zeile der Unit auch den gleichen Namen verwenden. Anschließend wirst du nach einem Namen für das Delphi-Projekt gefragt. Belasse ihn bei „Projekt1“ oder gib ihm einen griffigeren Namen, aber nicht denselben wie der Unit. Zum Beispiel für die Unit: „Test.pas“ und für die Projekt-Datei: „TestPr.dpr“, und achte darauf, für jede neue Anwendung einen neuen Ordner zu verwenden, sonst gibt es zumindest optisch ein heilloses Durcheinander, da Delphi für eine Anwendung 9 Dateien speichert einschließlich der fertigen exe-Datei, die automatisch bei jedem Compilieren gespeichert wird.

Aufgaben:



1. Benutze anstelle eines *TButton* eine ***TRadioGroup*** aus der Palette „Standard“, um aus mehreren Farben auszuwählen. Verwende speziell ihre Eigenschaften *Items* und *ItemIndex* und schreibe eine Ereignis-Methode für das *onclick*-Ereignis von *RadioGroup1*. Hilfreich könnte dabei ein Array folgender Art sein:

   const Farbe: array[0..3] of TColor = (clred, clgreen, clyellow, clwhite);

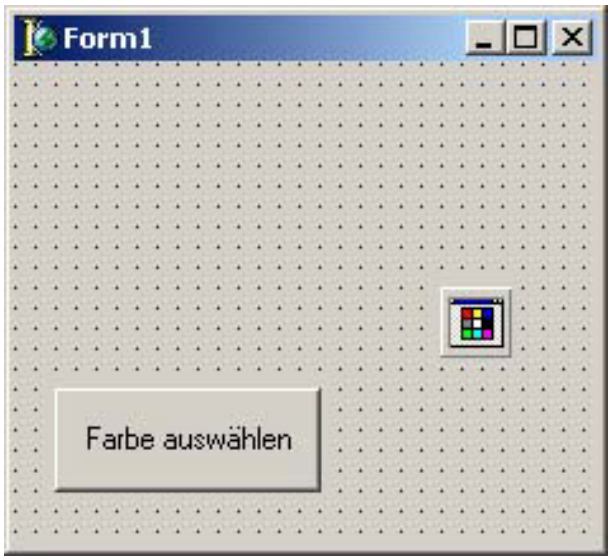
2. Für Aktionen, die zeitgesteuert ablaufen, benutzt man die (nicht-visuelle) Komponente ***TTimer*** aus der Palette „System“. Ergänze deine Anwendung aus Aufgabe 1 so, dass unabhängig von deinen Mausklicks die Farbe alle 5 Sekunden weitergestellt wird. Dieser Vorgang soll durch einen Button ein- und ausgeschaltet werden können. Füge einen Timer in dein Formular ein. Er ist als kleines Symbol zur Entwurfszeit sichtbar. *TTimer* besitzt die Eigenschaft *interval*, die du auf den Wert 5000 (Millisekunden) setzt. Nach dieser Zeit löst *TTimer* jeweils das Ereignis *ontimer* aus, für das du nebenstehende Methode schreiben kannst. Dann brauchst du noch einen Button mit der Beschriftung „Timer start / stop“. In der Methode für sein *onclick*-Ereignis sorgst du dafür, dass der Timer je nach Zustand ein- oder ausgeschaltet wird (Eigenschaft *enabled* von *TTimer*).

```
procedure TForm1.Timer1Timer (Sender: TObject);
var
  i:integer;
begin
  i := RadioGroup1.ItemIndex;
  if i<3 then i := i+1 else i := 0;
  RadioGroup1.ItemIndex := i;
  Form1.RadioGroup1Click(Sender);
end;
```

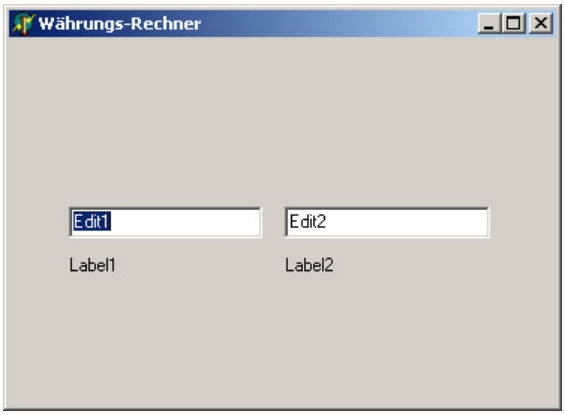
3. Benutze das Windows-Standardfenster zur Farbauswahl. In der Palette „Dialoge“ findest du die dazu nötige Komponente ***TColorDialog***. Diese Komponente ist wie *TTimer* zur Entwurfszeit nur als Symbol sichtbar, zur Laufzeit kannst du das Farbauswahl-Fenster z.B. durch einen Buttonklick anzeigen. Wenn der Benutzer das Fenster schließt, wird die gewählte Farbe übernommen. Die Methode für das *onclick*-Ereignis des Button muss dann so lauten:

   if ColorDialog1.Execute = true then
   Form1. Color : = ColorDialog1.Color;

   Die Methode *TColorDialog.Execute* öffnet das Dialogfeld zur Farbauswahl. Sie ist als Funktion vom Typ *boolean* definiert, so dass sie ein Ergebnis zurück gibt: Wenn der Benutzer eine Farbe auswählt und auf *OK* klickt, ergibt *execute = true*. Bricht der Benutzer ab, wird *false* zurückgegeben. Aus der Eigenschaft *ColorDialog1.Color* kann die Methode *TForm1.Button1Click* die gewählte Farbe in die Eigenschaft *Form1.Color* übernehmen.



4. Ein einfaches Rechenprogramm



Wir nehmen uns jetzt ein Beispiel vor, wo zwei Eingabe-Fenster zusammenarbeiten: Ein Programm zur Umrechnung von Währungen. Objekte der Klasse ***TEdit*** stellen einzeilige Eingabe-Fenster dar, in die der Benutzer Text eingibt. Wir wollen in eines einen Euro-Betrag eingeben. Das andere soll dann den Dollar-Wert anzeigen und umgekehrt. Für die Beschriftung der beiden Felder benutze ***TLabel***-Objekte. Ein *TLabel* hat nur die Aufgabe, einen Text handlich auf einem Formular auszugeben. Beide Komponenten findest du in der Palette „Standard“. Gib der Eigenschaft *Caption* von *Label1* den Wert „Euro“ und von *Label2* den Wert „US-Dollar“. Und lösche die Eigenschaft *Text* der beiden Edits, damit die Beschriftung verschwindet. *Text* ist die Eigenschaft, die den Inhalt des Fensters als *String* enthält. Deinem Formular kannst du noch die *Caption* „Währungs-Rechner“ geben.

Nun müssen wir eine **Ereignis-Methode** schreiben, die nach Eingabe eines Euro-Betrags in *Edit1* den Dollar-Betrag berechnet und in *Edit2* anzeigt. Wie du im Objekt-Inspektor siehst, kennt *TEdit* viele Ereignisse, aber keines, das ausgelöst wird bei Beenden der Eingabe mit der Eingabe-Taste. Am ehesten könnte zutreffen ***onKeyUp*** (bei Loslassen einer Taste). Verknüpfe also mit diesem Ereignis eine Methode: Delphi gibt ihr den Kopf

```
procedure TForm1.Edit1KeyUp(Sender: TObject; var Key: Word;  Shift: TShiftState);
```

Bevor du die Umrechnung programmierst, kannst du ausprobieren, ob diese Methode arbeitet wie erhofft: schreibe zwischen *begin* und *end* die Anweisung

```
Edit2.Text := 'Probetext';
```

und beobachte die Wirkung. Starte die Anwendung, klicke mit der Maus in *Edit1* und betätige eine Taste. In dem Moment, wo du sie loslässt, erscheint in *Edit2* dein „Probetext“. Das ist nicht schlecht, aber wir wollen die Umrechnung erst, wenn der Benutzer die Eingabe-Taste benutzt. Du siehst im Prozedur-Kopf, dass die **Variable *Key*** übergeben wird. Damit können wir feststellen, ob es die Eingabe-Taste war, und entsprechend reagieren. Leider ist an dieser Stelle die Delphi-Hilfe nicht optimal. Erst unter dem Stichwort „virtuelle Tastencodes“ findest du vernünftige Auskunft. Zwei mögliche Lösungen:

```
if Key = VK_RETURN then  Edit2.Text := 'Probetext';
```

oder

```
if ord(Key) = 13 then . . .
```

Für die Umrechnung rechts im Kasten brauchen wir nur noch eine Konstante *Kurs*, die du entweder im *private*- oder *public*-Teil von *TForm1* deklarieren kannst oder global bei dem *var*-Teil vor *implementation* oder aber auch nur ganz lokal innerhalb der *procedure TForm1.Edit1KeyUp*. Ich entscheide mich für eine globale Konstante:

```
var
  Form1: TForm1;
const
```

```
procedure TForm1.Edit1KeyUp(Sender: TObject;
 var Key: Word;  Shift: TShiftState);
var
  a:double;
begin
  if Key = VK_RETURN  then  begin
    a := StrToFloat(Edit1.Text); //Typ-Umwandlung
    a := a * Kurs;
    a := round(a*100) / 100;       //auf 2 Stellen runden
    Edit2.Text := FloatToStr(a); //Typ-Umwandlung
  end;
```

```
Kurs=1.18;   // Dollarkurs in Euro
```

Wie du siehst, ist unsere Anwendung mit diesen paar Zeilen bereits recht professionell. Für die Umrechnung von Dollar in Euro könntest du analog eine Methode für das *onKeyUp*-Ereignis von *Edit2* erstellen, den Code kopieren, darin die Edits vertauschen und durch den Kurs dividieren. Ich möchte dir aber gleich noch was anderes zeigen:

```
procedure TForm1.Edit1KeyUp(Sender: TOb-
 ject; var Key: Word;  Shift: TShiftState);
var
  a:double;
begin
  if Key = VK_RETURN  then  begin
    if Sender = Edit1 then begin
      a := StrToFloat(Edit1.Text);
      a := a * Kurs;
      a := round(a*100) / 100;
      Edit2.Text := FloatToStr(a);
    end else begin
      a := StrToFloat(Edit2.Text);
      a := a / Kurs;
      a := round(a*100) / 100;
      Edit1.Text := FloatToStr(a);
    end;
  end;
end;
```

Alle Ereignis-Methoden übergeben als Parameter den ***Sender*** des Ereignisses (als Zeiger auf das Objekt). Wir können die gesamte Kurs-Umrechnung in eine einzige Prozedur geben und in dieser abfragen, wer die Umrechnung veranlasst hat: *Edit1* oder *Edit2*. Damit das funktioniert, muss natürlich auch *onKeyUp* von *Edit2* mit derselben Methode verknüpft werden wie *Edit1*. Das geht einfach: Markiere *Edit2*, klicke im Objekt-Inspektor auf das *onKeyUp*-Ereignis und rechts daneben auf die DropDown-Liste: dort erscheinen alle Methoden von *Form1*. Wähle die Methode *Edit1KeyUp* aus. Den Code der Methode kannst du dann wie nebenan vervollständigen.

Du siehst, wie effektiv Programmieren in Delphi ist: Für diese Anwendung brauchen wir nur eine einzige kurze Prozedur zu schreiben, die das beinhaltet, was die Anwendung ausmacht. Die gesamte restliche Arbeit nimmt uns Delphi ab.

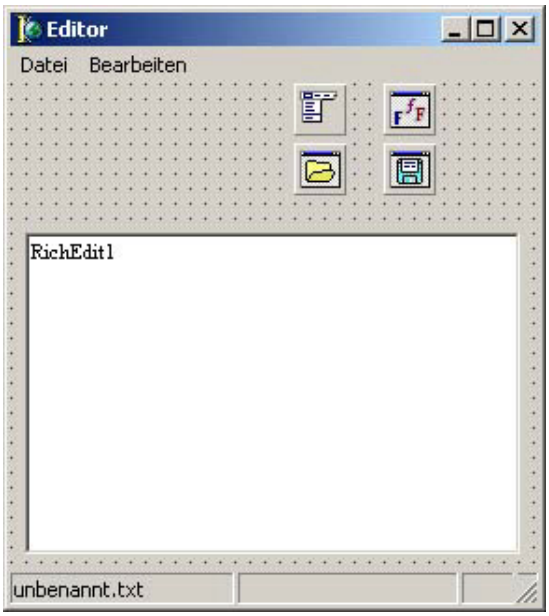
Aufgabe:

Der Anwender soll aus einer längeren Liste von Währungen eine auswählen können. Benutze zwei konstante Arrays für eine Währungstabelle und eine Kurstabelle. Zeige die erste in einer ***TComboBox*** an, damit der Benutzer eine Währung auswählen kann. Diese ComboBox setzt du an die Stelle von *Label2*, das bisher „Dollar“ angezeigt hat. Die Währung Euro kannst du zunächst festlassen. Den *Kurs* musst du jetzt als Variable definieren. Verknüpfe *oncreate* von *TForm1* mit einer Methode, die beim Start der Anwendung aufgerufen wird. Dort kannst du die *ComboBox1* mit der Währungstabelle beschreiben. Studiere die Eigenschaft *ItemIndex* und die Methoden *Clear* und *AddItem*. Zeige den Kurs der gewählten Währung auf einem eigenen Label an.

```
var
  Form1: TForm1;
  Kurs:double;
const
  Anzahl = 3;              //Länge der Tabelle
  waehrungTab: array[0. . Anzahl] of string = ('US-Dollar',
                    'engl. Pfund','jap. Yen','nor. Krone');
  kursTab: array[0 . . Anzahl] of double =
                                  (1.1865, 0.718, 140.43, 7.8831);
. . .
procedure TForm1.FormCreate(Sender: TObject);
var
  i:integer;
begin
  ComboBox1.Clear;          //eventuelle Einträge löschen
  for i:=0 to Anzahl do
       ComboBox1.AddItem(waehrungTab[i],nil);
  ComboBox1.ItemIndex:=0;   //auf ersten Eintrag stellen
  Kurs := kursTab[0];
end;
```

5. Die Entwicklung eines Texteditors

Jetzt wollen wir uns einer etwas größeren Aufgabe stellen und einen Texteditor programmieren mit allem, was so dazugehört: Menü, Statusleiste, Kopieren und Einfügen von Text, Schriftauswahl-Dialog und Methoden zum Öffnen und Speichern. Gehe dazu nach folgenden Schritten vor:



Eine neue Anwendung beginnen

Erstelle für das Projekt ein eigenes Verzeichnis, starte Delphi mit Datei/Neu/Anwendung, ändere die Eigenschaft *Form1.Caption* in „Editor“ und speichere das Projekt mit Datei/Projekt speichern unter...

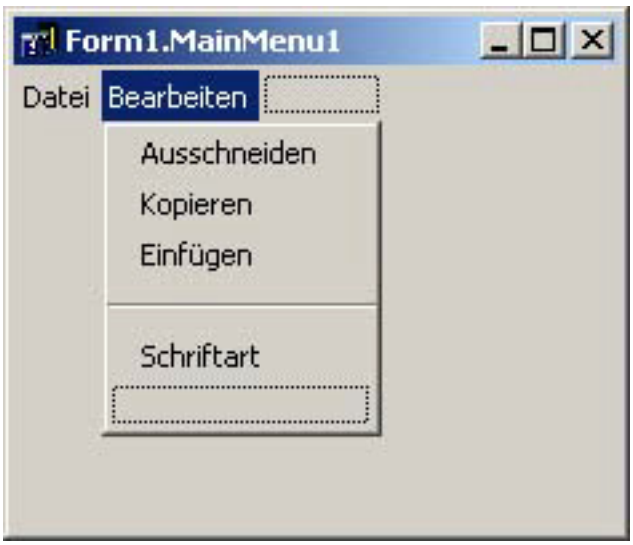
Komponenten ins Formular holen

Nach unserer Aufgabenstellung benötigen wir ein Textfenster (*TRichEdit* in der Palette „Win32“), eine Statusleiste (*TStatusBar* ebenfalls in „Win32“), ein Menü (*TMainMenu* in der Palette „Standard“) und aus der Palette „Dialoge“ ein *TFontDialog*, ein *TOpenDialog* und ein *TSaveDialog*.

Eigenschaften einstellen

Gib dem ***RichEdit1*** die gewünschte Größe und Position (experimentiere mit seiner Eigenschaft *Align*) und setze die Symbole der vier Dialoge irgendwohin, wo sie nicht stören. Für ein besseres Verständnis von *TRichEdit* solltest du in der Hilfe nachlesen, auch über seine Eigenschaft *Lines* von der Klasse *TStrings*. Objekte der Klasse *TStrings* besitzen unter anderem die Methoden *SaveToFile* und *LoadFromFile*, die wir weiter unten benötigen.

Doppelklicke auf die *StatusBar1*, um ihre Eigenschaft *Panels* zu bearbeiten. Füge drei oder vier neue *Panels* hinzu, markiere *Panels[0]* und setze seine Eigenschaft *Text* auf „unbenannt.txt“, und seine Eigenschaft *Width* etwa auf 120. Passe die Breite der anderen *Panels* (für den späteren Eintrag der Schriftart und ...) entsprechend an.



Doppelklicke auf das Symbol von ***MainMenu1***. Es öffnet sich ein Fenster, indem du dein Menü wie gewünscht erstellen kannst. Die Stichworte „Datei“, „Neu“, „Öffnen...“, „Speichern unter...“, „Beenden“ usw. schreibst du in die Eigenschaft *Caption* der *TMenuItem* genannten Einträge im Menü. Eine waagrechte Linie erhältst du, indem du für *Caption* ein „-“ eingibst. Wenn du fertig bist, schließe das Fenster wieder.

Wenn du auf das Symbol des ***FontDialog1*** doppelklickst, öffnet sich der Schriftendialog so wie später zur Laufzeit. Jetzt können wir allerdings noch keine Schrift für *RichEdit1* auswählen, das geht erst zur Laufzeit des Programms. Also schließe ihn wieder. Gleiches gilt für *SaveDialog1* und *OpenDialog1*.

Spätestens jetzt bist du sicher neugierig, wie deine Anwendung aussieht. Starte sie und teste den Texteditor und das Menü. Der Editor arbeitet und das Menü klappt auf und zu, nur reagiert *Form1* noch nicht

auf die *onclick*-Ereignisse von *MainMenu1*.

Ereignis-Methoden schreiben

Wir haben noch keine einzige Zeile Code geschrieben, und trotzdem hat unser Programm bereits eine beachtliche Funktionalität aufgrund der Eigenschaften und Methoden der verwendeten Objekt-Klassen. Die Typ-Deklaration in *Unit1.pas* ist auch schon beachtlich angewachsen und unbemerkt auch die Formulardatei, in der die Anfangswerte der Eigenschaften abgelegt sind. Nun schreiben wir die einzelnen Ereignis-Methoden und testen jede einzeln sofort aus, bis die Syntax korrekt ist.

Methode für „Datei beenden“

Öffne durch Doppelklick das *MainMenu1*, markiere das *MenuItem* „Beenden“ suche im Objekt-Inspektor das *onclick*-Ereignis, doppelklicke auf das Editierfeld daneben: Delphi erzeugt eine Methode *TForm1.Beenden1Click*. (Den Namen von Ereignis-Methoden setzt Delphi immer zusammen aus dem Namen des Objekts, welches das Ergeignis auslöst, und dem Ereignis ohne ‚on‘.) Diese Methode soll das Programm beenden. Ein Blick in die Methoden-Liste von *TForm* (Delphi-Hilfe) lässt dich die Methode *Close* finden und verrät dir, wie sie verwendet wird:

```
Form1.Close;
```

Zwischen *begin* und *end* tippen wir diese Anweisung ein. Und testen sofort den Erfolg.

(Alternativ zur Benutzung der Hilfe kannst du auch auf Verdacht tippen: „Form1.“ und auf das Popup-Menü warten. Dies hilft vor allem dann, wenn man schon weiß, es müsste wohl *close* heißen. Das Popup-Menü zeigt zusätzlich die eventuell nötigen Parameter an.)

Methode für „Datei neu“

Zuerst musst du dir klar machen, was bei dieser Aktion geschehen soll: *RichEdit1* soll gelöscht werden, ein etwaiger Dateiname soll auf „unbenannt.txt“ gesetzt werden und *StatusBar1* soll diesen Dateinamen anzeigen. Das ist ja einigermaßen überschaubar. Also:

Du brauchst eine Variable *Dateiname* vom Typ *string*. Es ist sinnvoll, sie als Eigenschaft von *TForm1* unter *public* zu deklarieren. So ist ein globaler Zugriff auf den Dateinamen möglich. In der Hilfe zu *TRichEdit* finden wir schnell die Methode *clear*. Also verknüpfen wir das *onclick*-Ereignis des Menüeintrags „Datei/Neu“ mit einer Methode *TForm1.Neu1Click* und füllen die Methode wie nebenstehend.

```
    Schriftart1: TMenuItem;
  procedure beenden1Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
    Dateiname: string;
  end;
var
  Form1: TForm1;
...
...
procedure TForm1.Neu1Click(Sender: TObject);
begin
  RichEdit1.Clear;
  Dateiname := ‚unbenannt.txt‘;
  StatusBar1.Panels[0].Text := Dateiname;
end;
```

Methode für „Datei öffnen“

Textdateien bestehen aus einer Aneinanderreihung von Strings, dafür gibt es die Klasse *TStrings*. *TRichEdit.Lines* ist von diesem Typ und hat deswegen auch seine Methoden *LoadFromFile* und *SaveToFile*, die wir jetzt benötigen. Unsere Aufgabe: Wir müssen *OpenDialog1* ausführen. Wenn der Benutzer eine Datei ausgewählt hat, holen wir Pfad und Dateiname

```
procedure TForm1.Oeffnen1Click(Sender: TObject);
var
  nurName:string;
begin
  OpenDialog1.Title : = ‚Textdatei öffnen‘;
  OpenDialog1.Filter : =
    ‚Textdateien (*.txt) | *.TXT | alle Dateien (*.*) | *.*‘;
  if OpenDialog1.Execute = true then begin
    Dateiname : = OpenDialog1.FileName;
    RichEdit1.Lines.LoadFromFile(Dateiname);
    nurName : = extractFileName(Dateiname);
    StatusBar1.Panels[0].Text : = nurName;
  end;
end;
```

```
procedure TForm1.SpeichernUnter1Click(Sender:
 TObject);
var
  nurName:string;
  nurPfad:string;
begin
  SaveDialog1.DefaultExt : = ‚txt‘;
  SaveDialog1.Filter : =
    ‚Textdateien (*.txt) | *.TXT | alle Dateien (*.*) | *.*‘;
  if Dateiname <> ‚unbenannt.txt‘  then begin
    nurName : = extractFileName(Dateiname);
    nurPfad : = extractFilePath(Dateiname);
    SaveDialog1.FileName : = nurName;
    SaveDialog1.InitialDir : = nurPfad;
  end;
  if SaveDialog1.Execute = true then begin
    Dateiname : = SaveDialog1.FileName;
    RichEdit1.Lines.SaveToFile(Dateiname);
    nurName : = extractFileName(Dateiname);
    StatusBar1.Panels[0].Text : = nurName;
  end;
```

aus dem Dialogfenster und öffnen damit die Textdatei. Fast schon Routine: verknüpfe zuerst das *onclick*-Ereignis des Menüeintrags „Datei öffnen“ mit einer Methode: *TForm1.Oeffnen1Click*. Delphi erzeugt uns das Gerüst der Methode selber, allerdings lautet ihr Name „ffnen1Click“. Delphi erlaubt im Programmcode keine deutschen Umlaute. Wenn dich das stört, nenne die Methode um in „Oeffnen1Click“. Bevor du *OpenDialog1* ausführst mit *execute*, kannst du seine Eigenschaft *Title* ordentlich setzen, außerdem hilft es dem Benutzer, wenn nur die richtigen Dateitypen im Fenster angezeigt werden. Beschreibe deswegen die Eigenschaft *Filter* wie nebenstehend. Die Verwendung von *execute* kennst du schon von *TColorDialog* aus Kapitel 3. Nach Schließen des Dialogfensters müssen wir seine Eigenschaft *FileName* auswerten, sie enthält nämlich Pfad und Dateiname. Delphi stellt außer den Objektklassen auch viele oft gebrauchte Funktionen global zur Verfügung. Eine solche ist *extractFileName* in der Unit *SysUtils*, die aus der kompletten Pfadangabe den Dateinamen holt. So sparen wir uns die Arbeit, das selber zu programmieren. Näheres erfährst du in der Hilfe. In die Statuszeile schreiben wir dann nur den Dateinamen ohne Pfad.

Nach dieser Arbeit (immerhin haben wir schon 10 Zeilen Code geschrieben!) sollte unsere Anwendung bereits jede Textdatei lesen und anzeigen können.

Methode für „Datei Speichern unter...“

Das Speichern der Textdatei läuft fast genauso ab wie das Öffnen. Ich denke, dass du den Quellcode gut mit der Delphi-Hilfe nachvollziehen kannst.

Methode für „Bearbeiten Ausschneiden / Kopieren / Einfügen“

TRichEdit stellt uns etliche Standard-Methoden der Textbearbeitung zur Verfügung. Die Delphi-Hilfe nennt uns u.a.

ClearSelection, *ClearUndo*, *CopyToClipboard*, *CutToClipboard*, *PasteFromClipboard*. Die letzten drei Methoden wollen wir noch in unserer Anwendung nutzen. Das geht für Text „Ausschneiden“ sehr flott, wie du in nebenstehendem Kasten siehst. Das gleiche gilt für die anderen beiden Aktionen.

```
procedure TForm1.Ausschneiden1Click(Sender: TObject);
begin
  RichEdit1.CutToClipboard;
end;
```

So bleibt uns als letztes Ziel die

Methode für die Schriftauswahl

Die Anwendung von *TFontDialog* funktioniert so wie die der anderen Dialog-Fenster. Vor dem Öffnen des Dialogs übergibst du ihm die aktuelle Schriftart (*Font*) von *RichEdit1*, nach dem Schließen des Dialogs holst du die aktuelle Schriftart von *FontDialog1* in *RichEdit1* und schreibst *Font.Name* in die Eigenschaft *StatusBar1.Panels[1].Text*.

Abschlussarbeiten

Beim Testen der Anwendung fällt auf, dass anfangs keine Schriftart in der Statusleiste steht sondern erst nach Benutzung des Schriftendialogs. Das könnten wir mit dem Objekt-Inspektor nachholen. Andererseits benutzt man auch gerne das Ereignis *oncreate* von *TForm1*, um Initialisierungen beim Programmstart vorzunehmen. Das sieht dann so aus wie nebenstehend. Konsequenter Weise gehört hier dann auch die Festlegung von *Panels[0].Text* hier hin. Diese habe wir ganz zu Anfang mit dem Objekt-Inspektor auf den Wert „unbenannt.txt“ gesetzt. Außerdem steht beim Programm-Start „RichEdit1“ im Textfenster. Öffne mit dem Objekt-Inspektor die Eigenschaft *Lines* von *RichEdit1* und entferne den Eintrag.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  StatusBar1.Panels[1].Text:=RichEdit1.Font.Name;
end;
```

Aufgaben:

1. Füge zu deiner Anwendung noch ein Info-Fenster hinzu, das Name, Copyright und ein Logo anzeigt: Füge dazu eine vorgefertigte Unit hinzu (Datei/Neu/weitere/Formulare/ Info-Fenster). Delphi holt die entsprechende Unit als *Unit2* ins Projekt. Füge diese durch einen Eintrag bei der *uses*-Klausel von *Unit1* hinzu. Mit dem Objekt-Inspektor stellst du die Eigenschaften des Info-Fensters ein wie gewünscht. Du kannst auch ein eigenes Logo entwerfen (Tools/Bildeditor/Datei/Neu/Symboldatei) und abspeichern. Anschließend gibst du es dem Info-Fenster der Eigenschaft *Picture* von *TImage* hinzu. Jetzt brauchst du nur noch einen Eintrag „Info“ in *MainMenu1* und die Ereignis-Methode mit der Anweisung

   AboutBox.ShowModal;

   AboutBox ist die Objekt-Variable für das Info-Fenster (s.h. *Unit2*) vom Typ *TAboutBox = class(TForm)* und besitzt deswegen auch die Methode ***showmodal*** von *TForm*, mit der ein neues Formular „modal“ angezeigt wird („es erhält den Focus“), d.h die Anwendung wird unterbrochen, bis der Benutzer dieses Fenster schließt.

2. Gib deiner Anwendung beim Schließen eine Sicherheitsabfrage „Speichern?“ hinzu. Suche dazu in der Hilfe nach ***MessageDlg*** und ordne dem ***onclose***-Ereignis von *Form1* eine entsprechende Methode zu. Die Funktion *MessageDlg* öffnet ein häufig gebrauchtes Fenster für Infos, Fehlermeldungen etc. und gibt als Ergebnis die Information zurück, auf welchen Button der Benutzer geklickt hat.

3. Gib deinem Menü neben „Datei Speichern unter...“ noch einen Eintrag „Speichern“. Hat der Benutzer versehentlich auf „Speichern“ geklickt, obwohl seine Datei noch „unbenannt.txt“ heißt, soll natürlich zu der Methode von „Speichern unter...“ verzweigt werden.

Der gesamte Quellcode der *Unit1* unseres Editors:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms,Dialogs, StdCtrls, ComCtrls, Menus;
type
  TForm1 = class(TForm)
    RichEdit1: TRichEdit;
    MainMenu1: TMainMenu;
    StatusBar1: TStatusBar;
    FontDialog1: TFontDialog;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    Datei1: TMenuItem;
    Neu1: TMenuItem;
    Oeffnen1: TMenuItem;
    Speichernunter1: TMenuItem;
    N1: TMenuItem;
    beenden1: TMenuItem;
    Bearbeiten1: TMenuItem;
    Ausschneiden1: TMenuItem;
    Kopieren1: TMenuItem;
    Einfuegen1: TMenuItem;
    N2: TMenuItem;
    Schriftart1: TMenuItem;
    procedure Beenden1Click(Sender: TObject);
    procedure Neu1Click(Sender: TObject);
    procedure Oeffnen1Click(Sender: TObject);
    procedure Speichernunter1Click(Sender: TObject);
    procedure Ausschneiden1Click(Sender: TObject);
    procedure Kopieren1Click(Sender: TObject);
    procedure Einfuegen1Click(Sender: TObject);
    procedure Schriftart1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
    Dateiname: string;
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.Beenden1Click(Sender: TObject);
begin
  Form1.Close;
end;
procedure TForm1.Neu1Click(Sender: TObject);
begin
  RichEdit1.Clear;
  Dateiname:='unbenannt.txt';
  StatusBar1.Panels[0].Text := Dateiname;
end;
procedure TForm1.Oeffnen1Click(Sender: TObject);
var
  nurName:string;
begin
  OpenDialog1.Title := ,Textdatei öffnen';
  OpenDialog1.Filter :=  ,Textdateien (*.txt) | *.TXT | alle Dateien (*.*)
  | *.*';
  if OpenDialog1.Execute = true then begin
    Dateiname := OpenDialog1.FileName;
    nurName := extractFileName(Dateiname);
    RichEdit1.Lines.LoadFromFile(Dateiname);
    StatusBar1.Panels[0].Text := nurName;
  end;
end;
procedure TForm1.Speichernunter1Click(Sender: TObject);
var
  nurName:string;
  nurPfad:string;
begin
  SaveDialog1.DefaultExt := ,txt';
  SaveDialog1.Filter := ,Textdateien (*.txt) | *.TXT | alle Dateien (*.*) |
  *.*';
  if Dateiname <> ,unbenannt.txt' then begin
    nurName := extractFileName(Dateiname);
    nurPfad := extractFilePath(Dateiname);
    SaveDialog1.FileName := nurName;
    SaveDialog1.InitialDir:=nurPfad;
  end;
  if SaveDialog1.Execute = true then begin
    Dateiname := SaveDialog1.FileName;
    RichEdit1.Lines.SaveToFile(Dateiname);
    nurName := extractFileName(Dateiname);
    StatusBar1.Panels[0].Text := nurName;
  end;
end;
procedure TForm1.Ausschneiden1Click(Sender: TObject);
begin
  RichEdit1.CutToClipboard;
end;
procedure TForm1.Kopieren1Click(Sender: TObject);
begin
  RichEdit1.CopyToClipboard;
end;
procedure TForm1.Einfuegen1Click(Sender: TObject);
begin
  RichEdit1.PasteFromClipboard;
end;
procedure TForm1.Schriftart1Click(Sender: TObject);
begin
  FontDialog1.Font := RichEdit1.Font;
  if FontDialog1.Execute = true then begin
    RichEdit1.Font := FontDialog1.Font;
    StatusBar1.Panels[1].Text := FontDialog1.Font.Name;
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  StatusBar1.Panels[1].Text := RichEdit1.Font.Name;
end;
end.
```

6. Der große Überblick ...

(fehlt dir noch länger, fürchte ich)

Herzlichen Glückwunsch! Du hast Wesentliches der Delphi-Programmierung bereits verstanden: In einfacheren Anwendungen vererbt man nur eine einzige Klasse *TForm1*, und erweitert ihre Eigenschaften und Methoden. Man fügt ihr Objekte anderer Klassen und nötige Variablen, Konstanten, Arrays, ... als Eigenschaften hinzu, setzt diese Eigenschaften zur Entwicklungszeit auf bestimmte Anfangswerte und verknüpft ihre Ereignisse mit Ereignis-Methoden von *TForm1*. Diese programmiert man so, dass alle Teile auf die gewünschte Art zusammenarbeiten. Natürlich kann man zu *Form1* außer den Ereignis-Methoden auch weitere Methoden (Prozeduren und Funktionen) hinzufügen. (Dies macht man im Abschnitt *private* oder *public* der Klassendeklaration.) Wenn du das Info-Fenster in die Anwendung „Editor" eingefügt hast, weißt du auch schon etwas über das Zusammenspiel mehrerer Fenster.

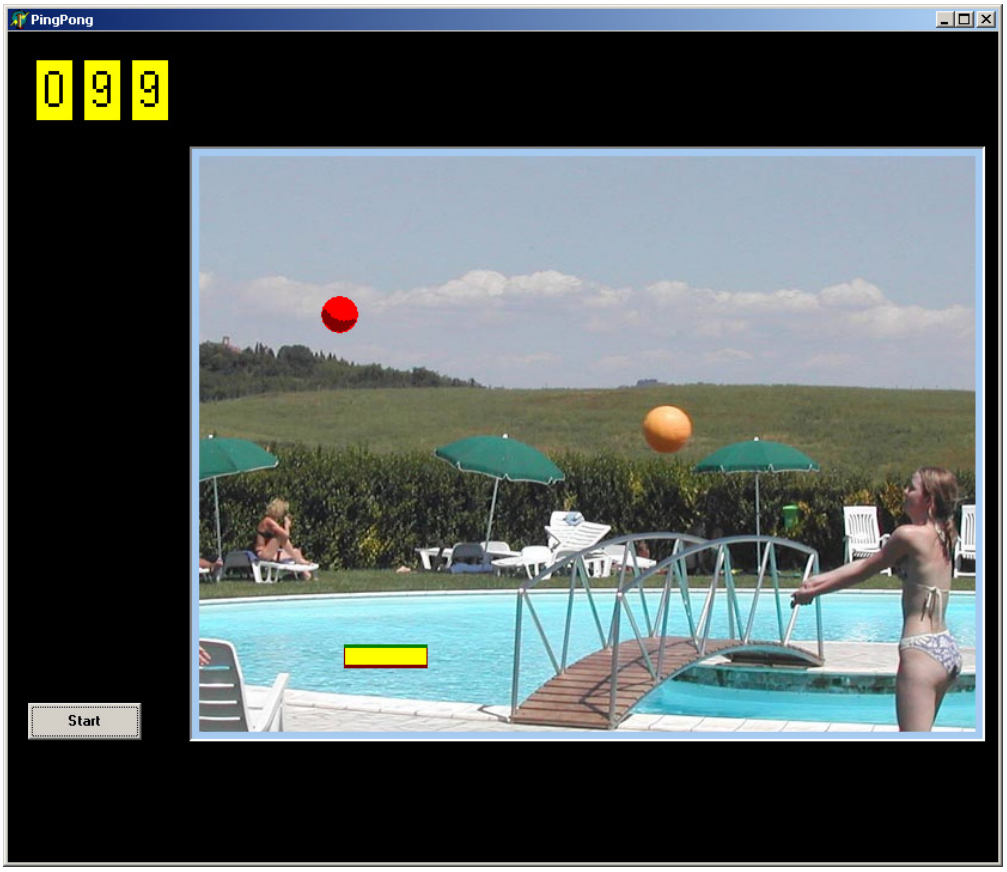
Was dir noch fehlt, ist die nötige Erfahrung und ein Überblick über die vorhandenen Klassen. Delphi-Klassen sind Teil zweier Klassenhierarchien, die als ***VCL*** (Visual Component Library) und ***CLX*** (Component Library for Cross Platform, für die Linux-Version von Delphi) bezeichnet werden. Die Komponenten sind in Delphi-Units abgelegt, z.B. ist *TButton* in der Unit *StdCtrls* deklariert, aber auch in *QStdCtrls*. Die Units, deren erster Buchstabe ein „Q" ist, gehören zur *CLX*. Die Mehrzahl der Komponenten findet man in beiden Bibliotheken. Verständlicher Weise ist der Umfang der *CLX* eingeschränkt, weil dort alle Komponenten fehlen, die Windows-spezifische Fähigkeiten haben z.B. *TADOConnection* (Verbindung zur ADO-Datenbank). Solange du nicht mit **Kylix** (Delphi für Linux) programmieren willst, bleibe bei den Units der *VCL*.

Für einen ersten Überblick studiere anhand der Delphi-Hilfe die **Komponenten in den Paletten** „Standard", „Zusätzlich", „Win32", „System" und „Dialoge", und auch das Internet bietet dir unter „Delphi", gefolgt von einem zweiten Suchbegriff, jede Menge Infos. Dann wirst du nicht umhin kommen, punktuell Beispiel-Programme zu studieren. In den Kapiteln 7 und 8 findest du zwei sehr lehrreiche Beispiele.

Außerhalb der Komponenten-Paletten gibt es wichtige (unsichtbare) Klassen: Grundlegende Datenstrukturen kapseln die Klassen *TList* und *TStrings*. Mit der Klasse *TStrings* hast du schon Bekanntschaft gemacht bei *TRichEdit.Lines*. Für Datenspeicherung wirst du über kurz oder lang nicht um die Stream-Klassen herumkommen: *TFileStream*, *TMemoryStream*, *THandleStream*. Für das Zeichnen einer Grafik zuständig ist *TCanvas*. Das ist eine Zeichenfläche, die viele Komponenten bereits besitzen, z.B. *TForm*, *TPaintBox*, *TPrinter*, *TBitmap*. Wenn du etwa ein Dokument am Bildschirm anzeigen und auch auf dem Drucker ausgeben willst, kannst du parallel ein sichtbares *TPaintBox*- und ein unsichtbares *TPrinter*-Objekt benutzen. Beide verfügen über ein *TCanvas*-Objekt. Je nachdem, wohin die Grafik gerade gehen soll, schreibst du mit ein und der selben Prozedur entweder in *TPrinter.Canvas* oder in *TPaintBox.Canvas*. In Kapitel 8 erkläre ich das ganz genau. Dann gibt es auch unsichtbare Klassen, die Bilder als Ganzes verwalten: *TIcon*, *TPicture*, *TMetaFile*, *TBitMap*. Das Pingpong-Spiel im nächsten Kapitel benutzt, um einen roten Ball anzuzeigen, die sichtbare Komponente *TImage*, die ein *TPicture*-Objekt enthält.

7. Ein Pingpong-Spiel

Die folgende, absichtlich sehr einfach gehaltene Anwendung bringt Bewegung auf den Bildschirm. Ein roter Ball, der an den Rändern des Spielfelds abprallt, soll mit einem gelben Schläger so lang wie möglich in der Luft gehalten werden. Die Zeit läuft als Highscore mit. Als Hintergrund-Grafik habe ich ein Digitalfoto eingefügt.



Form1 enthält folgende Objekte:

Spielfeld:TPanel, das ist eine Tafel, die mehrere Elemente zu einer Gruppe zusammenfasst. Ihre Eigenschaften *BevelInner* und *BevelOuter* habe ich auf den Wert *bvLowered* gesetzt (damit stellt man die Kanten der Tafel ein), *Color* auf ein helles Blau und *Name* auf „Spielfeld“.

Hintergrund:TImage, eingepasst in *Spielfeld*, so dass am Rand noch ein blauer Streifen zu sehen ist. Ihre Eigenschaft *Name* enthält „Hintergrund“. In ihre Eigenschaft *Picture* (vom Typ *TPicture*) habe ich ein Urlaubsfoto („Landschaft.jpg“) geholt: Mit einem Klick auf den Button neben *Picture* öffnet man den Bildeditor und lädt das gewünschte Bild. Danach musst du natürlich die Größe von *Hintergrund* und *Spielfeld* an die Bildgröße anpassen.

Ball:TImage wird auf den Hintergrund gesetzt, sein *Name* erhält den Wert „Ball“. Den Ball kannst du z.B. mit dem Bildeditor-Tool als Icon-Datei erstellen und abspeichern („Ball.ico“) und anschließend der Eigenschaft *Picture* zuweisen. Die Größe von *Ball* setzt du dann in *Width* und *Height* auf 32 Pixel, oder einfacher: setze die Eigenschaft *AutoSize* auf *true*.

Brett:TImage: ganz analog zu *Ball*. Da jedoch das Brett größer als ein Icon ist, erstelle im Bildeditor eine Bitmap-Datei.

Für die Anzeige des Highscore kannst du 3 Objekte von der Klasse ***TLabel*** benutzen, deren einzige Aufgabe es ist, Text sehr handlich auf dem Formular auszugeben. Ihren Hintergrund habe ich in *Color* auf

gelb gesetzt und *Font.Size* auf 32.

Außer einem **Start-Button** brauchst du dann nur noch ein ***TTimer***-Objekt. Sein *Interval* setzt du auf 20 (Millisekunden), sein *enabled* auf *false* und die Ereignis-Methode des *ontimer*-Ereignisses erhält nachher die komplette Steuerung des Balls. Beim Klick auf den Start-Button wird *Timer1.enabled* auf *true* gesetzt und schon geht es los, bis der Ball den Boden berührt. Dann muss *Timer1* wieder abgeschaltet werden. Das Brett wird mit der Maus gesteuert.

Wenn du bei jeder Ballberührung einen Sound hören willst, dann füge noch ein ***TMediaPlayer***-Objekt hinzu. Seine Eigenschaft *FileName* setzt du auf den gewünschten Sound, z.B. auf „chord.wav“ im Verzeichnis „C:\windows\media\“.

Wie du im Quellcode siehst, habe ich in *Form1* drei Ereignisse mit Methoden verknüpft: *SpielfeldMouseMove,* verknüpft mit ***onMouseMove*** von *Spielfeld.* Dieses Ereignis tritt ein, wenn die Maus über dem entsprechenden Steuerelement (hier *Spielfeld*) bewegt wird. *StartBtnClick* verknüpft mit *onclick* von *StartBtn*, und *Timer1Timer*, verknüpft mit *ontimer* von *Timer1*.

Außerdem braucht *Form1* noch **Variablen**: für die momentane Geschwindigkeit von *Ball* (*Ballvx, Ballvy*) in x- und y-Richtung, für den Punktestand (*Punkte*) und eine **Prozedur**, die den Punktestand in die drei Labels für die Ziffern schreibt (*SchreibePunkte*).

```
type
  TForm1 = class(TForm)
    Spielfeld: TPanel;
    Hintergrund: TImage;
    Ball: TImage;
    Brett: TImage;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Timer1: TTimer;
    StartBtn: TButton;
    MediaPlayer1: TMediaPlayer;
    procedure SpielfeldMouseMove(Sender:
       TObject; Shift: TShiftState; X,Y: Integer);
    procedure StartBtnClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private-Deklarationen }
    Ballvx, Ballvy: double;  // Geschwindigkeit
                             //des Balls
    Punkte:integer;
    procedure SchreibePunkte(p:integer);
  public
    { Public-Deklarationen }
  end;
```

Die Methoden von *Form1* sind in bekannter Pascal-Manier geschrieben und gut verständlich. Aber ich sollte noch etwas sagen zur

Methode *SpielfeldMouseMove*:

```
procedure TForm1.SpielfeldMouseMove(Sender:
 TObject; Shift: TShiftState; X,Y: Integer);
begin
  Brett.Left := X - (Brett.Width div 2);
end;
```

Die Ereignis-Methode, die Delphi einem *onMouseMove*-Ereignis automatisch zuordnet, übergibt außer dem *Sender* (hier *Spielfeld*) noch den Status einiger Tasten (Shift, Ctrl, re/li Mausetaste ...) und die Maus-Koordinaten bezüglich des *Sender*. Deswegen können wir unser Brett bereits mit einer einzigen Code-Zeile steuern. *Spielfeld* erzeugt das *onMouseMove*-Ereignis aber nur, solange die Maus über dem *Spielfeld* bewegt wird und sich außerdem nicht über einem anderen aktiven Steuerelement (*Hintergrund*, *Ball*, *Brett*) befindet. Deswegen musst du die Eigenschaft *enabled* dieser drei Objekte auf *false* setzen. Außerdem empfiehlt es sich, einen anderen Mauszeiger zu wählen: setze die Eigenschaft *Spielfeld.Cursor* auf *crSizeWE*.

Der Quellcode der Methoden von „Pingpong“:

```
procedure TForm1. SpielfeldMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  Brett.Left:=X-(Brett.Width div 2);              // Brettmitte beim Mauszeiger
end;

procedure TForm1. StartBtnClick(Sender: TObject);
begin
  Punkte := 0;
  SchreibePunkte(Punkte);
  Ball.Top := Spielfeld.Height div 10;             // Startpunkt
  Ball.Left := Spielfeld.Width div 2;
  Ballvx := random(20)-10.5;                       // Anfangsgeschw.
  Ballvy := 5+random(5);
  Timer1.Enabled := true;                          // Start
end;

procedure TForm1. Timer1Timer(Sender: TObject);
begin
  Ball.Top := Ball.Top+round(Ballvy);              // Schritt des Ball
  Ball.Left := Ball.Left+round(Ballvx);
  inc(Punkte);                                     // Punktezahl erhöhen
  SchreibePunkte(Punkte);
                                                   // Reflexion an der Wand und Decke
  if (Ball.left>Spielfeld.Width-Ball.Width) or (Ball.Left<0)
    then Ballvx := - Ballvx;
  if Ball.top<0 then Ballvy := - Ballvy;
                                                   // Reflexion am Brett
  if (Ball.top>Brett.Top-Ball.Height) and
     (Ball.top<Brett.Top-Ball.Height+2 * Ballvy) and
     (Ball.Left>Brett.Left-(Ball.Width div 2)) and
     (Ball.Left<Brett.Left+Brett.Width-(Ball.Width div 2))
    then begin
      MediaPlayer1.Play;
      Ballvy := - Ballvy * 1.5;                    // Ball wird jedesmal schneller
      Ballvx := Ballvx * 1.5;
      Ball.Top:=Ball.Top+round(Ballvy);
      Ball.Left:=Ball.Left+round(Ballvx);
    end;
                                                   // Ball berührt Boden: Spiel zu Ende
  if Ball.Top>Spielfeld.Height - Ball.Height
    then Timer1.Enabled := false;
end;

procedure TForm1. SchreibePunkte(p:integer);
var
  s:string;
begin
  str(p,s);                                        // verwandelt Zahl in string
  Label1.Caption := copy(s,length(s),1);
  if Punkte>9 then Label2.Caption := copy(s, length(s) - 1, 1)
    else Label2.Caption := ‘0‘;
  if Punkte>99 then Label3.Caption := copy(s, length(s) - 2, 1)
    else Label3.Caption := ‘0‘;
end;
```

8. Visitenkarten drucken

Unsere nächste Anwendung „FixVisit“ druckt Visitenkarten. Nach Eingabe der Daten sieht der Benutzer eine Visitenkarte vor sich. Er kann ein Bild hinzufügen, die Schrift beeinflussen und in gewissen Grenzen die Anordnung korrigieren. Jeweils 10 Karten werden auf eine DinA4-Seite gedruckt.

Dabei wirst du Eingabefenster der Klasse *TEdit* benutzen, lernen wie man Text in ein *TCanvas*-Objekt schreibt, wie man eine Bitmap-Grafik dazufügen kann, wie man beides mit *TScrollBar*-Objekten auf der Zeichenfläche verschieben kann und endlich, wie man mit der Klasse *TPrinter* das Ganze mit fester Größe unabhängig von den Geräten auf Papier bringt. Das ist eine Menge!

Also ans Werk: Für die Eingabe der Daten brauchst du acht *TEdit*- und acht *TLabel*-Objekte. Ein Tipp: Hole zunächst nur ein *Edit* und ein *Label*, stelle ihre Eigenschaften ein (*Text* löschen, *AutoSize* auf *false, ...*). Markiere dann beide und vervielfältige sie durch Kopieren und Einfügen. Anschließend markiere alle Edits und richte sie links aus, entsprechend die Labels rechts (rechte Maustaste/Position...). Das spart Zeit. Um die Visitenkarte zu zeichnen und anzuzeigen, benutzen wir eine *TPaintBox*. Das Foto müssen wir unabhängig von der Karte irgendwo unterbringen, denn es hat ja nicht gleich die richtige Größe, außerdem brauchen wir ein Objekt, das eine Methode zum Laden des Bildes besitzt: Dafür eignet sich *TImage* mit seinem *TPicture*-Objekt.

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    . . .
    Edit8: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    . . .
    Label8: TLabel;
    PaintBox1: TPaintBox;
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    FontDialog1: TFontDialog;
    PrintDialog1: TPrintDialog;
    MainMenu1: TMainMenu;
    OpenFotoBtn: TButton;
    OpenFontBtn: TButton;
    TextScrollX: TScrollBar;
    TextScrollY: TScrollBar;
    BildScrollX: TScrollBar;
    BildScrollY: TScrollBar;
```

Des weiteren brauchen wir folgende Dialoge: ein Menü (*TMainMenu*), ein Drucken-Fenster (*TPrintDialog*), ein Schriftauswahl-Fenster (*TFontDialog*) und *TOpenPictureDialog* zum Laden des Bildes. Dann noch zwei Buttons und vier Scrollbars (*TScrollBar*).

Einzustellen gibt es unter anderem: die Größe von *Paintbox1* (700x450; dazu später mehr), vielleicht musst du *Form1* größer machen. Die ScrollBars positionieren. Zwei müssen vertikal stehen (*Kind = sbvertical*). Setze jeweils *Min* = -100 und *Max* = 100: damit verschieben wir später Text oder Bild um 100 Pixel in die entsprechende Richtung. Dann geben wir ihnen noch verständlichere Namen, damit wir sie im Quellcode besser auseinanderhalten können: *Name* = „TextScrollX“, „TextScrollY“, „BildScrollX“ und „BildScrollY“. Die Größe und Lage von *Image1* ist belanglos, weil wir *Image1* unsichtbar machen (*visible = false*) und die Eigenschaft *autosize = true* setzen. Damit passt sich die Bitmap von *Image1* an die Größe des geladenen Fotos an. *TMainMenu* versehen wir mit den Einträgen „Datei“, „Beenden“, „Drucken“, „10 Visitenkarten...“. Die beiden Buttons bekommen die *Caption* „Foto einfügen...“ und „Schriftart...“ und vielleicht auch griffige Namen: *OpenFotoBtn* und *OpenFontBtn*.

```
private
  { Private-Deklarationen }
public
  { Public-Deklarationen }
  breite,hoehe:integer;
  defMitte,Textmitte:integer;
  BildposX,BildposY:integer;
  procedure zeichneKarte(welchesCanvas:
                                  TCanvas);
end;
```

Was brauchen wir zusätzlich an **Variablen**? Für die Berechnungen der Grafik die *breite* und *hoehe* der Karte (eigentlich identisch mit *width* und *height* von Paintbox1, aber etwas handlicher). Der Text soll zentriert werden, deswegen *defMitte* , und weil wir ihn auch verschieben wollen, zusätzlich *Textmitte*. Dann die Koordinaten der linken, oberen Bildecke *BildposX* und *BildposY*.

Nun die Frage, welche **Ereignis-Methoden** wir brauchen:

►Die Variablen brauchen beim Programmstart Anfangswerte. Markiere *Form1* (nicht eines seiner Objekte) und ordne dem Ereignis *oncreate* eine Methode ***TForm1.FormCreate*** zu wie nebenan. Alle Maße versuchen wir von *breite* und *hoehe* abzuleiten. Vielleicht willst du ja später dem Fenster eine andere Größe geben. Dann brauchst du außer *breite* und *hoehe* nichts zu ändern. Für die Textmitte könnte ein Drittel der Fensterbreite OK sein, das Bild könnte bei 2/3 der Breite beginnen bei einem Fünftel der Höhe (y-Werte werden von oben nach unten positiv gezählt).

```
procedure TForm1.FormCreate(Sender:
 TObject);
begin
  breite := 700;  hoehe := 450;
  Paintbox1.Width := breite;
  Paintbox1.Height := hoehe;
  defMitte := breite div 3;
  BildposX := (breite div 3) * 2;
  BildposY := hoehe div 5;
end;
```

►Dann müssen die Menü-Einträge funktionieren: Sehr schnell geht das *onclick*-Ereignis von „Datei/Beenden“: Die Methode ***TForm1.Beenden1Click*** erhält die Zeile

```
Form1.Close;
```

►Dann kommt das *onclick*-Ereignis von „Drucken/10Visitenkarten...“ dran: ***TForm1.Visitenkarten1Click***. Da müssen wir erst einiges lernen. Jedenfalls wissen wir aber, dass wir vor dem Druck *PrintDialog1* ausführen müssen. Also schreiben wir schon mal wie nebenan. Teste doch gleich die Funktion der Menüleiste ...

```
procedure TForm1.Visitenkarten1Click(Send
 er: TObject);
begin
  if PrintDialog1.Execute=true then begin
    // Druckerausgabe
    // ...
   end;
end;
```

Schnell geht auch das *onclick*-Ereignis des Schriften-Buttons:

```
FontDialog1.Execute;
Paintbox1.Repaint;
```

Nach Änderung der Schrift muss die Visitenkarte neu gezeichnet werden. Deswegen rufen wir ihre ***repaint*-Methode** auf, momentan natürlich noch ohne sichtbare Wirkung.

►Du merkst vielleicht: ich drücke mich noch ein bißchen um die Edit-Objekte herum, deren Inhalt ja in die Visitenkarte geschrieben werden muss. Wann soll der Text geschrieben werden? Eigentlich sofort, wenn irgendeine Eingabe gemacht wird. Braucht dann jedes Edit eine eigene Ereignis-Methode für das ***onchange*-Ereignis**? (*onchange* tritt ein, wenn sich der Inhalt des Edits ändert.) Das wäre sehr unübersichtlich. Es ist doch sinnvoll, die ganze Karte von einer Methode zeichnen zu lassen selbst dann, wenn noch nicht alle Eingaben gemacht sind. Dann fehlt halt noch was in der Karte. Wo gehört dann diese Methode hin?

Jetzt muss ich kurz ausholen: Alle sichtbaren Objekte (Nachkommen von *TControl*) besitzen eine Methode, mit der sie sich auf dem Bildschirm selber darstellen können. Diese heißt ***repaint***. Objekte, bei denen zu erwarten ist, dass wir Anwender etwas daraufzeichnen wollen (z.B. *TPaintBox*), besitzen dazu eine Methode ***paint***, die bei jedem Neuzeichnen von *repaint* aufgerufen wird und ein ***onpaint***-Ereignis erzeugt, welches wir mit einer selbstgeschriebenen Methode verknüpfen können. Wenn du mit dem Objekt-Inspektor auf die Suche gehst, stellst du fest, dass er z.B. für ein *TButton* kein *onpaint*-Ereignis anzeigt, für ein *TPaintBox* und ein *TForm* dagegen schon. Der Grund ist einfach: Der Programmierer von *TButton* hat zwar eine Methode *TButton.repaint* geschrieben, aber keine Methode *paint* und kein Ereignis *onpaint*, weil er es uns nicht leicht machen wollte, auf einen Button Grafitis zu malen. (Wenn wir das wirklich wollten, müssten wir von TButton eine neue Klasse ableiten und in ihr die *repaint*-Methode überschreiben. Aber du wirst es ahnen: solche Klassen gibt es längst, z.B. *TSpeedButton*)

Der Zweck von *TPaintBox* ist, dass wir etwas darauf zeichnen können. Also verknüpfen wir mit *onpaint* eine Methode *TForm1.PaintBox1Paint*. Sie wird dann bei jedem Neuzeichnen des Objekts von *PaintBox1.repaint* aufgerufen und zeichnet die PaintBox so, wie wir es wollen. Schwierig? - schon, aber sehr einfach anzuwenden!

Jetzt sind wir mit den *Edits* aus dem Schneider. Soll doch die Arbeit die *TPaintBox1.repaint* selber erledigen. Die Edits müssen nur dafür sorgen, dass *PaintBox1* etwas von der Änderung erfährt. Markiere alle acht Edits und verknüpfe mit dem *onchange*-Ereignis eine einzige Methode (die Delphi nach dem ersten Edit benennt): ***TForm1.Edit1Change***. Dort rufen wir *PaintBox1.repaint* auf und veranlassen so, dass PaintBox1 neu gezeichnet wird.

```
procedure TForm1.Edit1Change(Sender:
 TObject);
begin
 PaintBox1.Repaint;
end;
```

Was sollen denn die *ScrollBars* leisten, wenn sie verschoben werden? Eigentlich doch das gleiche wie die *Edits*: sie sollen der *PaintBox1* mitteilen, dass sie sich neu zeichnen soll. Also verknüpfen wir die *on-*

change-Ereignisse der vier *ScrollBars* ebenfalls mit *Edit1Change*. Recht effektiv, oder? Eine Zeile Code und 12 Objekte führen ihn aus.

```
public
  { Public-Deklarationen }
  breite,hoehe:integer;
  defMitte,Textmitte:integer;
  BildposX,BildposY:integer;
  procedure zeichneKarte(welchesCanvas:
                                  TCanvas);
end;
```

```
procedure TForm1.PaintBox1Paint(Sende
  r: TObject);
begin
  zeichneKarte(PaintBox1.Canvas);
end;
```

►Das ist jetzt wunderbar. Nur fehlt uns immer noch das Kernstück unseres Programms: die Methode, die die Visitenkarte zeichnet. Du willst endlich etwas sehen. Verknüpfe *onpaint* von *Paintbox1* mit ***TForm1.PaintBox1Paint***. Du erinnerst dich, was ich weiter oben über das Drucken gesagt habe: Für die Drucker-Ausgabe zeichnen wir die Visitenkarte später in das *Canvas* eines *TPrinter*-Objekts, für die Darstellung auf dem Bildschirm zeichnen wir sie jetzt in das *Canvas* von *PaintBox1*. Zweimal das gleiche: dafür lohnt es sich, eine eigene Prozedur anzulegen: diese deklarieren wir als Prozedur von *Form1*. Der erste Block ganz oben vor dem Wort *private* ist Delphi verbehalten, dort stehen die mit dem Objekt-Inspektor verknüpften Methoden. Unsere zusätzliche Prozedur deklarieren wir hinter unseren Variablen unter *public*: ***procedure zeichneKarte(welchesCanvas: TCanvas)***. In *TForm1.PaintBox1Paint* rufen wir die Prozedur *zeichneKarte* auf und übergeben ihr *PaintBox1.Canvas* zum Beschreiben. In der Druck-Methode übergeben wir später stattdessen *Printer.Canvas*.

```
procedure TForm1.zeichneKarte (welches-
  Canvas: TCanvas);
var
  s:string;
  liRand, top, abstand: integer;
begin
  top := (hoehe div 5) + TextScrollY.Position;
  abstand := FontDialog1.Font.Size;
  TextMitte := defMitte+TextScrollX.Position;
  with welchesCanvas do begin
    // Hintergrund weiss
    brush.Color := clwhite;
    pen.Color := clred;
    rectangle(0, 0, breite, hoehe);
    // Name
    s := Edit2.Text + ` ` + Edit1.Text;
    Font := FontDialog1.Font;
    Font.Height := FontDialog1.Font.Height;
    liRand := Textmitte - (textwidth(s) div 2);
    textout(liRand,top,s);
end;
```

►***<u>TForm1.zeichneKarte(welchesCanvas:TCanvas);</u>***

Jetzt musst du die Eigenschaften und Methoden der Klasse *TCanvas* in der Hilfe studieren: *Pen* ist der Zeichenstift, der Eigenschaften wie *Color*, *Width* und *Style* kennt. *Brush* liefert Farbe und Muster von Flächen, *Font* definiert die aktuelle Schrift, mit der die Methode *Textout* Text in die Zeichenfläche ausgibt. Häufig gebraucht werden auch die Methoden *draw*, *moveto*, *lineto*, *rectangle, ellipse*. *Textwidth* ermittelt die Breite (in Pixel) eines auszugebenden String, während *Font.Height* die Pixelhöhe der Schrift darstellt.

Im linken Kasten siehst du den ersten Teil der Prozedur *zeichneKarte*. Zuerst färben wir die Fläche weiß und geben ihr einen roten Rand. Das besorgt *rectangle* unter Verwendung von *brush* und *pen*. Anschließend schreiben wir Vorname und Name zentriert um *TextMitte*, das macht *textout* unter Verwendung von *Font*. Du siehst, wie die vertikale Position (*top*) zusammengesetzt wird aus 1/5 der *hoehe* und der Position der Scrollbar *TextScrollY*. Den Zeilenabstand (*abstand*) lasse ich von der Schriftgröße abhängen und addiere später Vielfache von ihm bei weiteren Textzeilen zu *top*. *Textmitte* wird von der Position von *TextScrollX* beeinflusst. Den linken Rand für die Textausgabe ermitteln wir, indem wir zuerst die Pixelbreite des Textes

mit *textwidth* bestimmen. Die Schrift holen wir aus *FontDialog1*. Ein Testlauf ist für dich bestimmt recht spannend. Wahrscheinlich aber wird dein Name winzig dargestellt. Stelle im Objekt-Inspektor schon von vornherein *FontDialog1.Font.Size* auf mindestens den Wert 32. An dieser Stelle habe ich mir eine sehr zufriedene Pause gegönnt.

Ich denke, die restlichen Textzeilen kannst du jetzt allein zeichnen. Die Schrifthöhe kannst du ab der zweiten Zeile z.B. mit

```
Font.Height := (FontDialog1.Font. Height * 6) div 10;
```

auf 60% verkleinern.

Danach haben wir noch drei Aufgaben zu lösen:

- Ein Bild in *Image1.Picture* laden.
- Dieses skaliert in *PaintBox1.Canvas* einfügen.
- Die fertige Visitenkarte ausdrucken.

```
procedure TForm1.OpenFotoBtnClick(Sender: TObject);
begin
  OpenPictureDialog1.Title:='Bilddatei öffnen';
  OpenPictureDialog1.Filter:='Bitmaps (*.bmp)|*.BMP';
  if OpenPictureDialog1.Execute = true then begin
    Image1.Picture.LoadFromFile(OpenPictureDialog1.
                                              FileName);
    PaintBox1.Repaint;
  end;
end;
```

►Die erste Aufgabe ist schnell erledigt: Verknüpfe mit dem *onclick*-Ereignis von *OpenFotoBtn* die Ereignis-Methode ***TForm1.OpenFotoBtnClick***. Der ***TOpenPictureDialog*** ist ein Öffnen-Dialogfenster, das nur die Grafik-Dateien anzeigt, welche die Klasse *TPicture* öffnen kann: Bitmaps (BMP), Symbole (ICO), Windows-Metadateien (WMF) und erweiterte Windows-Metadateien (EMF). Ich habe die Eigenschaft *Filter* sogar auf Bitmaps eingeschränkt. Wenn dein Bild im jpeg-Format vorliegt, musst du es vorher mit einem Grafikprogramm in eine Bitmap umwandeln (oder du studierst die Klasse TJpegImage). Hat der Benutzer im Dialogfenster eine Datei ausgewählt, kann *Image1.Picture.LoadFromFile* die Datei laden. Wer schreibt das Bild in das *Canvas* von *PaintBox1*? Natürlich *PaintBox1* mit seiner *repaint*-Methode (die *PaintBox1Paint* aufruft und diese wiederum unsere Prozedur *zeichneKarte*), also fügen wir im Anschluss ans Laden die Anweisung *PaintBox1.Repaint* dazu.

►Aber richtig: *zeichneKarte* kann das ja noch gar nicht. Also muss sie noch ergänzt werden. Canvas hat eine Methode ***copyrect***, die einen Teil eines Bildes aus einer anderen Zeichenfläche holt.

```
r1 := Image1.ClientRect;
```

ermittelt ein Rechteck in Größe des Bilds. Wenn man als Ziel ein festes Rechteck r vorgibt, wird das Bild in dieses eingepasst, d.h. es wird vermutlich verzerrt. Damit es im gleichen Seitenverhältnis skaliert wird, bestimme den Faktor: In der Visitenkarte soll das Bild 200 Pixel breit sein, also

```
var
  r, r1: TRect;   // Typ für Rechteck-Variablen
  Bildfaktor : double;
begin     //  von zeichneKarte
. . .
// Foto von Image1 in Paintbox1  kopieren
    r1 := Image1.ClientRect;
    Bildfaktor := Image1.Width / 200;
    r.Left := BildposX + BildScrollX.Position;
    r.Top := BildposY + BildScrollY.Position;
    r.Right := r.Left+200;
    r.Bottom := r.Top + round(Image1.Height/
                                        Bildfaktor);
    copyrect(r, Image1.Canvas, r1);
  end;
```

```
Bildfaktor := Image1.Width / 200;
```

Damit ergeben sich auch die Ecken des Rechtecks *r* in *PaintBox1.Canvas*.

Wenn alles klappt, solltest du mal kurz über nebenstehende Weisheit nachdenken.

Die drei Todfeinde eines Programmierers sind:

1. frische Luft
2. Tageslicht
3. das unerträgliche Zwitschern der Vögel

►Zu guter Letzt soll die Visitenkarte auch ausgedruckt werden. Wie schon oben angedeutet gibt es in der **Unit *Printers*** die Klasse *TPrinter*, von der beim ersten Ansprechen der **Funktion *printer*** ein globales *TPrinter*-Objekt zurückgegeben wird. Damit du *printer* benutzen kannst, musst du zur *uses*-Klausel die Unit *Printers* hinzufügen. Ein einfacher Ausdruck gelingt dann wie nebenan.

```
with Printer do
begin
  BeginDoc;
  setMapMode(printer.canvas.handle,MM_lometric);
  Canvas.TextOut(200, -200, `Das ist ein Probeaus-
                                              druck`);
  EndDoc;
end;
```

Jeder Druckauftrag (für jede einzelne Druckseite!) beginnt mit der Methode *Printer.BeginDoc* und wird beendet mit *Printer.EndDoc*. Dazwischen muss *Printer.Canvas* beschrieben werden. Wenn du einen maßgenauen Ausdruck willst, benutze die Windows-Funktion *setMapMode*. *mm_lometric* bedeutet, dass ein Pixel die Breite 0,1mm hat (bei mm_himetric: 0,01mm). Deswegen wird unsere Visitenkarte mit 700x450 Pixel auf dem Papier genau 7cm breit und 4,5cm hoch. Allerdings wird in diesem Modus die y-Richtung nach unten negativ gezählt, anders als in *PaintBox.Canvas*, das musst du beim Beschreiben von *Printer.Canvas* beachten!!

```
procedure TForm1.Visitenkarten1Click(Sender: TObject);
var
  i:integer;
begin
   PrintDialog1.Copies :=  1;            // Vorgaben für den Printdialog
   Printer.Orientation := poPortrait;  // Hochformat
  if PrintDialog1.Execute = true then begin
    printer.BeginDoc;
    setMapMode(printer.canvas.handle,MM_lometric); // lometric: 1 Pixel = 0,1mm
    for i := 0 to 4 do begin
      setWindowOrgEx(Printer.Canvas.Handle, -200, 300 + i*500, nil); // Ursprung auf dem
                                  // Papier 200Pixel nach rechts und 300 Pixel nach unten verlegen
      zeichneKarte(Printer.Canvas);
      setWindowOrgEx(Printer.Canvas.Handle, -1000, 300+i*500, nil);
      zeichneKarte(Printer.Canvas);
    end;
    printer.EndDoc;
  end;
end;
```

Jetzt können wir endlich unsere Methode ***TForm1.Visitenkarten1Click*** zu Ende schreiben. Im Wesentlichen benutzen wir das eben Gesagte und unsere Prozedur *zeichneKarte*. Wie du im Quellcode siehst, verwende ich noch die Windows-Funktion *setWindowOrgEx*. Mit ihr verlege ich den Ursprung des Koordinatensystems von der linken, oberen Papierecke dorthin, wo die Visitenkarte hinkommen soll. Insgesamt zeichne ich mit *zeichneKarte* 10 mal eine Visitenkarte, und weil ich als Parameter *Printer.Canvas* übergebe, wirken sich alle Zeichenanweisungen auf *Printer.Canvas* aus, nicht auf *PaintBox1.Canvas*.

Aber bitte, verzweifle noch nicht, wenn dein Drucker in diesem Moment noch ein leeres Blatt ausgibt: Die y-Richtung ist in diesem Abbildungsmodus nach unten negativ. *ZeichneKarte* zeichnet bis jetzt alles oberhalb des Papiers ins Leere. Du musst also noch dafür sorgen, dass je nach übergebenem Canvas alle y-Werte positives bzw. negatives Vorzeichen bekommen. Das kannst du mit einer Variablen *yVorz* besorgen, die je nach Canvas den Wert -1 oder +1 hat.

Auf den nächsten beiden Seiten findest du den kompletten Quellcode von *FixVisit*.

```
procedure TForm1.zeichneKarte(welchesCanvas:
 TCanvas);
var
 yVorz: integer
begin
 if welchesCanvas = Printer.Canvas then yVorz := -1
                                   else yVorz := 1;
 with welchesCanvas do begin
  . . .
  rectangel(0, 0, breite, hoehe * yVorz);
  . . .
  top := 300 * yVorz;
  left := 200;
  textout(left, top, `blablabla`);
 . . .
 end;
end;
```

Du hast jetzt bereits unheimlich viel über Delphi und seine Klassen gelernt und besitzt nun gute Voraussetzungen, alleine weiterzukommen. Mir hat es Spass gemacht, dich dabei begleiten zu dürfen. Vielen Dank! Über eine Rückmeldung an *bertram.hafner@t-online.de* freue ich mich jederzeit.

Und falls du dich später nochmal aufraffen willst, könnten wir zusammen im letzten Kapitel selber eine neue Klasse erstellen und sie als Delphi-Komponente in die IDE integrieren. Dazu muss ich aber erst noch ein bißchen Vorarbeit leisten.

Literatur-Hinweise:

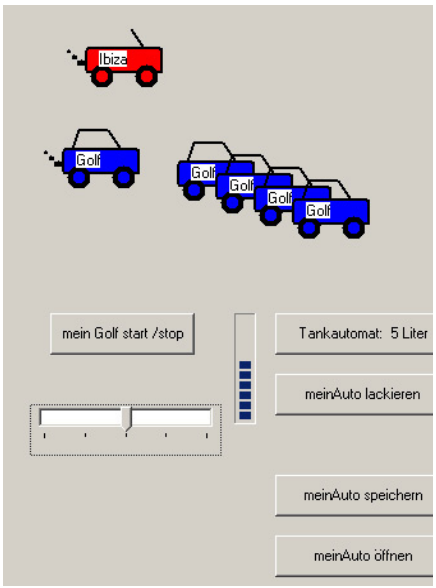
Grundlagen: Elmar Warken, Delphi 3, Addison Wesley, Bonn 1997 (>1000 Seiten !)

Einstieg: Doberenz / Kowalski: Programmieren lernen in Borland Delphi 6, Hanser, 2001

allgemein: Delphi-Source.de

```
unit Unit1;

interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, Menus, ExtDlgs, Buttons, Prin-
  ters;
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    . . .
    Edit8: TEdit;
    Label1: TLabel;
    . . .
    Label8: TLabel;
    PaintBox1: TPaintBox;
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    FontDialog1: TFontDialog;
    PrintDialog1: TPrintDialog;
    MainMenu1: TMainMenu;
    OpenFotoBtn: TButton;
    OpenFontBtn: TButton;
    TextScrollX: TScrollBar;
    TextScrollY: TScrollBar;
    BildScrollX: TScrollBar;
    BildScrollY: TScrollBar;
    Datei1: TMenuItem;
    Beenden1: TMenuItem;
    Drucken1: TMenuItem;
    Visitenkarten1: TMenuItem;
    procedure OpenFotoBtnClick(Sender: TObject);
    procedure PaintBox1Paint(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Visitenkarten1Click(Sender: TObject);
    procedure Beenden1Click(Sender: TObject);
    procedure OpenFontBtnClick(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
    breite,hoehe:integer;
    defMitte,Textmitte:integer;
    BildposX,BildposY:integer;
    procedure zeichneKarte(welchesCanvas:TCanvas);
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  breite := 700;
  hoehe := 450;
  Paintbox1.Width := breite;
  Paintbox1.Height := hoehe;
  defMitte := breite div 3;
  BildposX := (breite div 3) * 2;
  BildposY := hoehe div 5;
end;

procedure TForm1.Beenden1Click(Sender: TObject);
begin
  Form1.Close;
end;

procedure TForm1.Visitenkarten1Click(Sender: TObject);
var
  i:integer;
begin
  PrintDialog1.Copies:=1;     // Vorgaben für den Printdialog
  Printer.Orientation:=poPortrait;  // Hochformat
                                    // Printdialog ausführen
  if PrintDialog1.Execute=true then begin
    printer.BeginDoc;           // Druckerausgabe
    setMapMode(printer.canvas.handle,MM_lometric);
                                    // lometric: 1 Pixel = 0,1mm
                                    // himetric: 1 Pixel = 0,01mm
    for i:=0 to 4 do begin
      setWindowOrgEx(Printer.Canvas.Handle,-200, 300+i*500,
                                                        nil);
            // Ursprung auf dem Papier 200Pixel nach rechts
            // und 300 Pixel nach unten verlegen
      zeichneKarte(Printer.Canvas);
      setWindowOrgEx(Printer.Canvas.Handle,-1000, 300+i*500,
                                                        nil);
      zeichneKarte(Printer.Canvas);
    end;
    printer.EndDoc;
  end;
end;

procedure TForm1.OpenFontBtnClick(Sender: TObject);
begin
  FontDialog1. Execute;
  Paintbox1. Repaint;
end;
```

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 PaintBox1. Repaint;
end;

procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
  zeichneKarte(PaintBox1.Canvas);
end;

procedure TForm1.zeichneKarte(welchesCanvas:TCanvas);
    // Beachte: diese procedure zeichnet auf PaintBox1.Canvas
    // ebenso wie auf Printer.Canvas (Unit Printers), aber:
    // Printer.Canvas rechnet y-Werte nach unten negativ !!
    // dies wird durch die Variable yVorz berücksichtigt.
var
  s:string;
  liRand, liRand2: integer;
  top, abstand, yVorz: integer;
  r, r1: TRect;
  Bildfaktor: double;
begin
  if welchesCanvas = Printer.Canvas then yVorz := -1
                                    else  yVorz := 1;
  top := ((hoehe div 5) + TextScrollY.Position) * yVorz;
  abstand := FontDialog1.Font.Size * yVorz;
  TextMitte:= defMitte + TextScrollX.Position;
  with welchesCanvas do begin
     // Hintergrund weiss
    brush.Color := clwhite;
    pen.Color := clred;
    rectangle(0, 0, breite, hoehe * yVorz);
     // Name
    s:=Edit2.Text + ´  ` + Edit1.Text;
    Font := FontDialog1.Font;
    Font.Height := FontDialog1.Font.Height;
                //heigth misst in Pixel nicht in Punkten wie size
    liRand := Textmitte - (textwidth(s) div 2);
    textout(liRand, top, s);
     // Straße
    s:=Edit3.Text;
    Font.Height := (FontDialog1.Font.Height*6) div 10;
              // 60% der Höhe des Namen
    liRand2 := Textmitte - (textwidth(s) div 2);
    textout(liRand2, top + 2*abstand, s);

// PLZ Ort
  s:=Edit4.Text + ` ` + Edit5.Text;
  liRand2 := Textmitte - (textwidth(s) div 2);
  textout(liRand2, top + 3*abstand, s);
  // Tel, Handy, eMail
  Font.Height := (FontDialog1.Font.Height*5) div 10;
            // 50% der Höhe des Namen
  if Edit6.Text <> '' then begin
    s := ´Tel:    ` + Edit6.Text;
    textout(liRand, top + 7*abstand, s);
  end;
  top := top + 7*abstand;
  if Edit7.Text <> '' then begin
    s := 'mobil: ` + Edit7.Text;
    top := top + abstand;
    textout(liRand, top, s);
  end;
  if Edit8.Text <> '' then begin
    s := 'eMail: ` + Edit8.Text;
    top := top + abstand;
    textout(liRand, top, s);
  end;
    // Foto von Image1 in Paintbox kopieren
  r1 := Image1.ClientRect;         // so groß wie die Bitmap
  Bildfaktor := Image1.Width/200;
                                  //Bildbreite soll 200 Pixel betragen
  r.Left := BildposX + BildScrollX.Position;     //Zielrechteck
  r.Top := (BildposY + BildScrollY.Position) * yVorz;
  r.Right := r.Left + 200;
  r.Bottom := r.Top + round(Image1.Height/Bildfaktor)*yVorz;
  copyrect(r, Image1.Canvas, r1);
                    //kopiert Bitmap von Image1.Canvas
 end;               //in PaintBox1.Canvas bzw. in Printer.Canvas
end;

procedure TForm1.OpenFotoBtnClick(Sender: TObject);
begin
  OpenPictureDialog1.Title := 'Bilddatei öffnen';
  OpenPictureDialog1.Filter := 'Bitmaps (*.bmp)|*.BMP';
  if OpenPictureDialog1.Execute = true then begin
   Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
   PaintBox1.Repaint;
  end;
end;

end.
```



9. Entwicklung einer Klasse *TAuto*

In diesem Kapitel möchte ich auf den Ausgangspunkt meines Delphi-Einstiegs zurückkommen und Schritt für Schritt erklären, wie man eine neue Delphi-Klasse entwickelt. Ich bleibe bei der dort gewählten Beispiel-Klasse Auto, weil ich meinen kindlichen Spieltrieb bewahrt habe. Allerdings meine ich nicht die Autos aus unserer Wirklichkeit sondern eher Spielzeugautos, die auf dem Bildschirm fahren. Unsere Objekte von einer solchen Klasse *TAuto* sollen eine Modellbezeichnung bekommen (Golf, Ibiza, ...), in mehreren Typen vorkommen (Limousine, Kombi, Cabrio) und unterschiedliche Farbe besitzen. Sie sollen auf dem Formular sichtbar sein, der Motor soll gestartet und gestoppt werden können, zum Fahren sollen verschiedene Gänge eingelegt werden und über die Benzinmenge des Tanks wollen wir Auskunft bekommen, damit wir rechtzeitig tanken können. Damit es nicht zu umfangreich wird, begrenzen wir jetzt mal unsere Wunschliste außer: Ich will mein liebes Auto auch abspeichern können, damit ich morgen wieder mit dem gleichen weiterfahren kann.

Das UML-Diagramm für meine Klasse *TAuto* sieht dann nach einiger Überlegung mit Bleistift und Papier so aus wie hier. Die Frage, ob man eine ganz neue Klasse erstellen soll oder eine vorhandene vererben, entscheidet sich sehr oft zugunsten der Vererbung: natürlich könnte ich die Autos auf das Canvas von *Form1* zeichnen. Aber wenn ich von der Klasse *TPaintBox* ausgehe, habe ich viele Vorteile: die Autos sind unabhänger, weil jedes Auto dann sein eigenes Canvas hat. (Stell dir vor, du willst sie auf einem Hintergrund fahren lassen ...!) Dazu viele Eigenschaften (*left*, *top*, *visible*, *width*, *height* ...), Ereignisse (*onclick*, *onpaint* ...) und Methoden (*create*, *paint* ...). Da ist doch manches dabei, was man brauchen kann.

TAuto = class (TPaintbox)
<<eigenschaften>>
+Modell: string (nur lesen)
+Autotyp: TAutotyp (nur lesen)
+Farbe: TColor
+Gang: TGang
+Benzinmenge: integer (nur lesen)
-Motor: TTimer
+MotorLaeuft: Boolean (nur lesen)
<<ereignisse>>
+onTankChange
<<constructor>>
+create(AOwner:TWinControl; at:TAutotyp; Fa:TColor; Modell: string)
<<methoden>>
+paint; override
+MotorStart
+MotorStop
- Schritt(Sender: TObject)
+Tanken(Menge: double)
+SaveToFile(Pfad_Name: TFileName)
+LoadFromFile(Pfad_Name: FileName)

Die nächste Frage: Soll die Klasse *TAuto* in die IDE als Komponente aufgenommen werden (dann kann man die Autos aus einer Komponentenpalette mit der Maus ins Formular setzen und die Eigenschaften im Objekt-Inspektor einstellen)? Oder genügt eine Unit *Autos*, die man bei der *uses*-Klausel in eine Anwendung einbindet? Ersteres rentiert sich nur, wenn man eine Klasse entwickelt, die man in künftigen Anwendungen immer wieder benötigt. Außer dem Objekt-Inspektor hat man ja noch das Pulldown-Menü, das ebenfalls zu jedem Objekt jede Auskunft erteilt. Wenn du dich für die Komponente entscheidest, wähle „Komponente/neue Komponente“ und folge den Anweisungen unter Studium der Hilfe. Meist genügt aber die zweite Möglichkeit. Die Unit auch noch hin-

terher als Komponente von Delphi registriert werden. Ich gehe hier den zweiten Weg: Beginne zuerst eine neue Anwendung und speichere sie in ein eigenes Verzeichnis (Unit1 und Projekt-Datei). Wähle dann „Datei/Neu/Unit“. Delphi öffnet eine „Unit2“. Gib ihr in der ersten Zeile den Namen „Unit Autos“ und speichere sie unter „Autos.pas“ ebenfalls in das vorige Verzeichnis. Jetzt kannst du diese neue Unit *Autos* unter der *uses*-Klausel in *Unit1* dazufügen. *Unit1* mit dem Formular *Form1* brauchst du, um deine Kreation sichtbar zu machen und auszutesten, denn die Unit *Autos* stellt nur die abstrakte Klasse *TAuto* dar (du weißt: der Bauplan), die konkreten Objekte („Instanzen“) davon fahren in *Form1* (nicht in Formel1 :-).

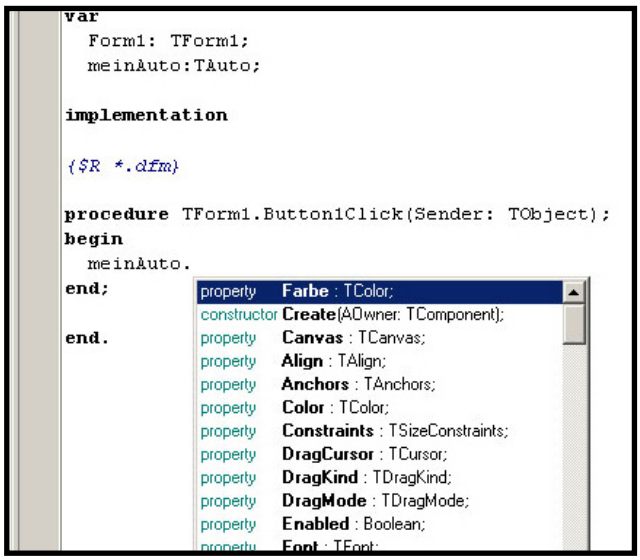
```
unit Autos;
interface
uses
  Classes,Graphics,ExtCtrls;
type
TAuto = class(TPaintbox)
  private
    FFarbe:TColor;
    procedure setFarbe(f:TColor);
  public
    property Farbe:TColor read FFarbe write setFarbe;
end;
implementation
procedure TAuto.setFarbe(f:TColor);
begin
  FFarbe := f;
  repaint;
end;
end.
```

Womit wollen wir jetzt beginnen? Wir schauen ein bißchen bei der Deklaration von *TForm1* in *Unit1* nach und deklarieren *TAuto* entsprechend. Ich erkläre dir jetzt, wie *TAuto* eine Eigenschaft *Farbe* bekommt (mit der später das Auto gezeichnet wird).

Die **Eigenschaft (=*property*) *Farbe*** ist mehr als eine Variable *Farbe*. Beim Setzen einer neuen Farbe ändert sich nicht nur der Wert einer Variablen von *clred* auf *clgreen*, sondern das Auto muss auch neu gezeichnet werden. Deswegen stehen hinter der Eigenschaft *Farbe* **eine Variable *FFarbe*** (es ist üblich einfach, ein ’F’ davor zu setzen) **und zwei Methoden**: zum Lesen und Setzen der Farbe. Zum Lesen genügt meist nur das Auslesen der Variablen, deswegen steht hinter der Klausel ***read*** nur *FFarbe* (dort dürfte auch eine Methode stehen). Hinter ***write*** steht der Name der Methode, die für das Setzen der Farbe verantwortlich ist. Diese muss aber vorher schon deklariert sein als Prozedur. Ebenso natürlich die Variable *FFarbe*. *setFarbe* übergibt dann den gewünschten Wert im Parameter *f* der Variablen *FFarbe* und sorgt mit *repaint* (von *TPaintBox* geerbt) dafür, dass das Auto neu gezeichnet wird. Beim Start des Programms kannst du natürlich noch nichts sehen, denn du hast ja noch gar kein Objekt von *TAuto* erzeugt. Außerdem erhältst du vielleicht die Fehlermeldung „unbekannter Bezeichner TPaintBox/TColor“. Die Hilfe sagt dir, dass *TColor* in der Unit *Graphics* definiert ist und *TPaintBox* in *ExtCtrls*, also füge sie hinzu. Auf alles, was du unter ***public*** deklarierst, ist ein Zugriff von außerhalb der Klasse *TAuto* möglich (z.B. meinAuto.Farbe := clred;). Deklarationen unter ***private*** sind nach außen nicht sichtbar. Möchtest du, dass Eigenschaften von einer Komponente im Objekt-Inspektor sichtbar sind, musst du anstatt *public* die Klausel ***published*** verwenden. Wenn du eine Seite zurückblätterst, findest du im UML-Diagramm Eigenschaften und Methoden mit einem ’+’. Diese möchte ich unter *public* deklarieren, diejenigen mit einem ’-’ unter *private*.

```
unit Unit1;
. . .
var
  Form1: TForm1;
  meinAuto: TAuto;
. . .
```

Du kannst deine Klasse *TAuto* bereits einem Test unterziehen: Deklariere in der *Unit1* im Variablen-Block, wo auch *Form1* deklariert ist, eine Variable *mein-*




```
unit Unit1;
. . .
procedure TForm1.FormCreate(Sender:
 TObject);
begin
  meinAuto:=TAuto.create(Self);
end;
. . .
```

Auto. Füge in *Form1* einen Button ein, verknüpfe mit seinem *onclick*-Ereignis eine Methode und schreibe in diese

```
meinAuto.Farbe := clred;
```

Du wirst dich freuen: in dem Moment, wo du 'meinAuto.' getippt hast, zeigt dir Delphi im Pulldown-Menü die Eigenschaften und Methoden von *TAuto* an! Das sind viele. Alle von *TPaintBox* geerbt. Außer: **Farbe !!!**

Starte deine Anwendung, drücke auf den Button - und wundere dich nicht über eine Fehlermeldung „Zugriffsverletzung ...". Was ist passiert?

Wir haben doch eine funktionierende Klasse *TAuto* und eine Variable *meinAuto*. Das ist richtig. Aber was enthält diese Variable? Du hast erst ihren Typ deklariert. *meinAuto* ist ein Objekt-Zeiger, der noch auf kein Objekt zeigt, er hat noch den Wert 'nil'. Du musst erst ein Objekt vom Typ *TAuto* im Speicher erzeugen, dann kannst du den Zeiger darauf stellen. Dies erledigt eine Methode, die man **Konstruktor** nennt.

Alle Objekt-Klassen brauchen einen Konstruktor.

Häufig heißt diese Methode ***create***, so auch bei *TForm* und *TPaintBox,* und sie ist als Funktion definiert, die einen solchen Zeiger zurückgibt. Wir werden später für *TAuto* einen neuen Konstruktor schreiben. Vorläufig benutzen wir aber den geerbten, am besten beim Start der Anwendung. Benutze das *oncreate*-Ereignis von *Form1* und verknüpfe nebenstehende Methode. *TPaintBox.create* erwartet einen Parameter, der einen Zeiger auf eine Komponente darstellt, welcher das erzeugte Objekt gehören soll . Ein *TPaintBox*-Objekt ist nämlich kein eigenständiges Fenster und braucht deswegen ein Fenster-Objekt, dem es gehört („Owner"). Wem soll unser *meinAuto* gehören? Natürlich *Form1*. Dazu verwendet man den **Bezeichner *Self***, der immer einen Zeiger darstellt auf das Objekt, innerhalb dem der Bezeichner *Self* benutzt wird, hier also *Form1*.

So, jetzt hat alles seine Ordnung, und du kannst die Eigenschaft *Farbe* von *meinAuto* benutzen oder auch von anderen Objekten des Typs *TAuto*. Die größte Hürde ist geschafft !! Du hast eine neue Klasse aus einem Vorfahren vererbt, ihr eine Eigenschaft hinzugefügt und ein Objekt dieser Klasse in deiner Anwendung erzeugt.

Die beiden Eigenschaften *Modell* und *AutoTyp* sollen nur zu lesen sein, also brauchst du keine *write*-Direktive mit Methode. Für *AutoTyp* solltest du Vorsorge treffen, dass man nur vorgegebene Typen setzen kann (z.B. Limousine, Kombi, Cabrio), weil die Methode *paint* damit zurecht kommen muss. Deswegen empfehle ich dir, außerhalb von *TAuto* dafür einen Aufzählungstyp *TAutoTyp* zu definieren in folgender Art:

```
type
  TAutotyp = (atLimousine, atKombi, atCabrio);
```

Dann brauchst du die Variable *FAutoTyp* nicht als *string* zu definieren, sondern kannst *TAutoTyp* verwenden. Dann kann man die Variable nur mit einem der vorgegebenen Werte versehen und nicht etwa mit „Geländewagen“. Entsprechendes habe ich etwas später mit der Eigenschaft *Gang* gemacht. Diese macht aber erst Sinn, wenn *TAuto* ein Objekt *Motor* hat. Dass man zur Bewegungssteuerung ein *TTimer*-Objekt einsetzen kann, kennst du ja schon von dem PingPong-Spiel. Ein solches Objekt habe ich *Motor* genannt, es kann ja auch wie dieser schneller und langsamer „laufen“.

Ich präsentiere dir jetzt die gesamte „Unit Autos“, damit du einen Überblick gewinnen kannst, und bespreche danach die Themen Konstruktor, Überschreiben der Methode *paint* und Erstellen des Ereignisses *onTankChange*. Wie man einen *TAutoRecord* deklariert und zum Laden und Speichern benutzt, das kannst du dem Listing selber entnehmen. (Es werden nur die Daten des Autos gespeichert, die für die Erzeugung eines Auto-Objekts nötig sind.)

```
unit Autos;
interface
uses Graphics,Classes,ExtCtrls,Controls,SysUtils,Dialogs;

type
  TAutotyp = (atLimousine, atKombi, atCabrio);
  TGang = (gaRueck,gaLeer,gaVor1,gaVor2,gaVor3);
  TAutoRecord = record
    Modell:string[20];
    Autotyp:TAutotyp;
    Farbe:TColor;
  end;
  TTankChangeEvent = procedure(Sender:TObject;
                               Menge:integer) of object;
const
  Gang:array[-1..3] of TGang =(gaRueck,gaLeer,gaVor1,
                               gaVor2,gaVor3);
type
  TAuto = class(TPaintbox)
  private
    FModell:string;
    FAutotyp:TAutotyp;
    FFarbe:TColor;
    FGang:TGang;
    FBenzinmenge:double;
    Motor: TTimer;
    FMotorLaeuft:Boolean;
    FOnTankChange:TTankChangeEvent;
    procedure setFarbe(f:TColor);
    procedure setGang(g:TGang);
    procedure schritt(Sender:TObject);
public
    property Modell:string read FModell;
    property Autotyp:TAutotyp read FAutotyp;
    property Farbe:TColor read FFarbe write setFarbe;
    property Gang:TGang read FGang write setGang;
    property Benzinmenge:double read FBenzinmenge;
    property MotorLaeuft:Boolean read FMotorLaeuft;
    property OnTankChange:TTankChangeEvent
             read FOnTankChange write FOnTankChange;
    constructor create(AOwner:TWinControl;Fa:TColor;
                       at:TAutotyp;Modell:string);
    procedure paint; override; //TPaintBox.paint überschreiben
    procedure MotorStart;
    procedure MotorStop;
    procedure Tanken(Menge:double);
    procedure SaveToFile(fn:TFileName);
    procedure LoadFromFile(fn:TFileName);
end;

implementation

constructor TAuto. create(AOwner:TWinControl;Fa:TColor;
                          at:TAutotyp;modell:string);
begin
  inherited create(AOwner);
  parent := AOwner;        //TPaintBox braucht einen Besitzer
  height := 80; width := 100;    //geerbte Eigenschaften setzen
  left := 0; top := 0;
  setfarbe(Fa);
  FAutotyp := at;
  FModell := modell;
  FGang := gaLeer;
  FBenzinmenge := 50;              //Tank voll
  FMotorLaeuft := false;          //Standardzustand
  Motor := TTimer.Create(Self);   //Motor erschaffen als Objekt
                                  von TAuto
  Motor.Enabled := false;         //Motor aus
  Motor.OnTimer := schritt;      //dem onTimer-Event wird die
                  //Procedurevariable TAuto.schritt zugeordnet
  repaint;
end;

procedure TAuto.paint;
begin
  inherited paint;          // TPaintBox.paint aufrufen
  with canvas do begin
    Pen.Width:=2; Pen.Color:=clblack; Brush.Color:=FFarbe;
    Roundrect(20,40,80,60,3,3);  //Korpus
    Pen.Width := 4;
    ellipse(25,55,40,70); ellipse(60,55,75,70); //Räder
    Pen.Width := 2;
    moveto(65,40); lineto(55,25);    //Scheibe
    case FAutotyp of                 //Dach
      atLimousine: begin
```

```
      lineto(32,25); lineto(27,40);
    end;
    atKombi: begin
      lineto(22,25); lineto(20,40);
    end;
  end;
  Brush.Color:=clwhite;TextOut(30,42,FModell);
  if FMotorLaeuft then begin      //Abgase
    ellipse(13,50,20,54);ellipse(9,46,13,49);ellipse(5,42,8,44);
  end;
 end;
end;

procedure TAuto.schritt(Sender:TObject);
begin
 if FGang<>gaLeer then
  if FGang = gaRueck then left := left-1 else left:=left+1;
 FBenzinmenge := FBenzinmenge - 0.05;   //Sprit-Verbrauch
 if assigned(FOnTankChange)
    then FOnTankChange(Self, round(FBenzinmenge));
 if FBenzinmenge<0 then MotorStop;
end;

procedure TAuto.MotorStart;
begin
 if FBenzinmenge>0 then begin
  Motor.Enabled := true;
  FMotorLaeuft := true;
  setGang(gaLeer);
  repaint;
 end;
end;

procedure TAuto.MotorStop;
begin
 Motor.Enabled := false;
 FMotorLaeuft := false;
 setGang(gaLeer);
 repaint;
end;

procedure TAuto.Tanken(Menge:double);
begin
 if MotorLaeuft then MessageDlg('Tanken bei laufendem
       Motor nicht möglich!!', mtInformation,[mbOk], 0)
 else begin
  FBenzinmenge := FBenzinmenge+Menge;
  if FBenzinmenge>50 then FBenzinmenge := 50;
  if assigned(FOnTankChange)
      then FOnTankChange(Self,round(FBenzinmenge));
 end;
end;

procedure TAuto.SaveToFile(fn:TFileName);
var
 f: file of TAutorecord;
 diesesAutorec: TAutorecord;
begin
 diesesAutorec.Modell := FModell;
 diesesAutorec.Autotyp := FAutotyp;
 diesesAutorec.Farbe := FFarbe;
 assignfile(f,fn);
 rewrite(f);            //öffnen für Schreibzugriff
 write(f,diesesAutorec);
 closefile(f);
end;

procedure TAuto.LoadFromFile(fn:TFileName);
var
 f: file of TAutorecord;
 diesesAutorec: TAutorecord;
begin
 assignfile(f,fn);
 {$I-} reset(f); {$I+}     //öffnen für Lesezugriff
 if ioresult = 0 then begin
  read(f,diesesAutorec);
  closefile(f);
  FModell := diesesAutorec.Modell;
  FAutotyp := diesesAutorec.Autotyp;
  setFarbe(diesesAutorec.Farbe);
  repaint;
 end else MessageDlg('Fehler beim Öffnen!!',
                      mtInformation,[mbOk], 0);
end;

procedure TAuto.setFarbe(f:TColor);
begin
 if motorlaeuft then
  MessageDlg('Lackieren bei laufendem Motor nicht
                möglich!!', mtInformation,[mbOk], 0)
 else begin
  FFarbe := f;
  repaint;
 end;
end;

procedure TAuto.setGang(g:TGang);
begin
 if Motorlaeuft=false then begin
  FGang := gaLeer;
  Motor.Interval:=0;
 end else begin
  FGang:=g;
  if (g=gaRueck)or(g=gaVor1) then Motor.Interval:=45;
  if g=gaVor2 then Motor.Interval:=20;
  if g=gaVor3 then Motor.Interval:=12;
  if g=gaLeer then Motor.Interval:=0;
 end;
end;

end.
```

constructor TAuto.create

Die Konstruktor genannte Methode muss ein Objekt aus der Klasse *TAuto* erzeugen. Da TAuto ein Nachfahre der Klasse TPaintBox ist, besitzt sie bereits einen Konstruktor, der in *TPaintBox* so deklariert ist:

```
constructor create(AOwner: TComponent);
```

Beim Aufruf können und müssen wir als einzigen Parameter einen Zeiger auf den Besitzer des Objekts übergeben. Wenn wir aber bei der Erzeugung eines Auto-Objekts bereits *Farbe*, *Modell* und *AutoTyp* festlegen wollen, müssen wir eine neue Konstruktor-Methode deklarieren mit zusätzlichen Parametern:

```
constructor create(AOwner:TWinControl; Fa:TColor; at:TAutotyp; Modell:string);
```

Es leuchtet ein, dass *TAuto* als Abkömmling von *TPaintBox* bei der Erzeugung eines Objekts die alte *create*-Methode aufrufen muss. Das macht man immer als erstes mit der Direktive *inherited*:

```
inherited create(AOwner);
```

TPaintBox braucht um sichtbar werden zu können ein Eltern-Fenster (Eigenschaft *parent*):

```
parent := AOwner;
```

Bei der Deklaration des Parameters scheint ein Widerspruch zu bestehen: *TComponent* gegen *TWinControl*. Da *parent* nicht in *TComponent* deklariert ist sondern erst in einem Abkömmling *TWinControl*, braucht *parent* einen Zeiger auf ein *TWinControl*-Objekt (unser *Form1*), nicht nur auf ein *TComponent*. Umgekehrt ist aber *TPaintBox.Create* damit zufrieden, weil ja TWinControl ein TComponent ist. Du brauchst nicht alles auf Anhieb zu verstehen. Irgendwann musste das ja kommen! Vielleicht solltest du gelegentlich die Ahnentafel studieren :-(

Dann initialisieren wir die neuen Eigenschaften soweit nötig, setzen die übergebenen Parameterwerte und erschaffen mit

```
Motor := TTimer.Create(Self);
```

ein *TTimer*-Objekt als Komponente von *TAuto*. Jetzt kannst du mit dessen *onTimer*-Ereignis eine Methode *schritt* von *TAuto* verknüpfen, die das Auto bewegen soll:

```
Motor.OnTimer := schritt;
```

```
constructor TAuto.create(AOwner:TWinCon
 trol;Fa:TColor; at:TAutotyp; modell:string);
begin
  inherited create(AOwner);
  parent:=AOwner;
  height:=80;width:=100;
  left:=0;top:=0;
  setFarbe(Fa);
  FAutotyp := at;
  FModell :=modell;
  FGang:=gaLeer;
  FBenzinmenge:=50;
  FMotorLaeuft:=false;
  Motor:=TTimer.Create(Self);
  Motor.Enabled:=false;
  Motor.OnTimer:=schritt;
  repaint;
end;
```

Das Verknüpfen von Ereignissen funktioniert hier nicht mehr wie in den vorigen Kapiteln mit dem Objekt-Inspektor (aber genauso einfach!), weil wir dieses *TTimer*-Objekt erst bei der Erzeugung eines *TAuto*-Objekts miterzeugen beim Start der Anwendung und nicht schon zur Entwurfszeit der Anwendung visuell darauf zugreifen können.

In der *Unit1* wird dann ein Objekt von *TAuto* nicht mehr so erzeugt:

```
meinAuto:=TAuto.create(Self);
```

sondern so:

```
meinAuto:=TAuto.create(Self, clred, atCabrio, 'Golf');
```

```
var
  Form1: TForm1;
  meinAuto: TAuto;
  Auto: array[1..5] of  TAuto;

implementation
{$R *.dfm}
procedure TForm1.FormCreate(Sender:
                                  TObject);
var
  i: integer;
begin
  for i := 1 to 5 do begin      // 5 neue Golf
    Auto[i] := TAuto.create(Form1, clblue,
                        atLimousine, 'Golf');
    Auto[i].Left := 300 + 30 * i;
    Auto[i].Top := 100 + 10 * i;
  end;
  meinAuto := Auto[1];
  . . .
```

In diesem Zusammenhang sollte ich erwähnen, dass man Objektzeiger auch in einem Array deklarieren kann, wie du links an dem Ausschnitt von *Unit1* siehst.

procedure TAuto.paint

Die Methode paint soll das Auto auf *TAuto.canvas* zeichnen. Da der Vorfahr *TPaintBox* eine gleichnamig Methode paint besitzt, deklarieren wir sie als *override*. Damit ersetzt („überschreibt“) unsere Methode die alte. Nun ist die Frage: brauchen wir die alte Methode auch noch? Die Delphi-Hilfe erläutert: „Die Methode Paint generiert das Ereignis OnPaint.“ Wenn du also möchtest, dass *TAuto* ein *onPaint*-Ereignis besitzt, dann rufe als erstes die geerbte Methode auf:

```
inherited paint;
```

dann brauchst du ggf. so ein Ereignis nicht selber zu implementieren. Die Anweisungen zum Zeichnen kannst du leicht nachvollziehen oder abändern. Die aktuelle *FFarbe* verwende ich als *Brush.Color*. Limousine und Kombi zeichne ich geringfügig anders als ein Cabrio. Und wenn der Motor läuft, qualmt der Auspuff.

```
procedure TAuto.paint;
begin
  inherited paint;
  with canvas do begin
    Pen.Width := 2; Pen.Color := clblack;
    Brush.Color := FFarbe;
    Roundrect(20,40,80,60,3,3);   //Korpus
    Pen.Width := 4;                //Räder
    ellipse(25,55,40,70);ellipse(60,55,75,70);
    Pen.Width:=2;
    moveto(65,40); lineto(55,25); //Scheibe
    case FAutotyp of               //Dach
      atLimousine: begin
        lineto(32,25); lineto(27,40);
      end;
      atKombi: begin
        lineto(22,25); lineto(20,40);
      end;
    end;
    Brush.Color := clwhite;
    TextOut(30,42,FModell);
    if FMotorLaeuft then begin     //Abgase
      ellipse(13,50,20,54);ellipse(9,46,13,49);
      ellipse(5,42,8,44);
    end;
  end;
end;
```

Das Ereignis onTankChange

Mit der Entwicklung eines Ereignisses möchte ich meinen Delphi-Einstieg beenden. Das Betriebssystem Windows sendet **Nachrichten** wie die Nachricht über eine Mausbewegung. Diese Nachricht durchläuft die Objekte einer Anwendung und kann dort als **Ereignis (*Event*)** eines oder mehrerer Objekte ausgelöst werden, wie z.B. *onMouseClick* eines Button, sodass wir mit einer Ereignis-Methode darauf reagieren können. Wir können einer neuen Klasse *TAuto* auch selbst definierte Ereignisse hinzufügen, etwa ein Ereignis *onTankChange*, das immer dann ausgelöst werden soll, wenn der Tankinhalt sich ändert. Das geschieht bei jedem Schritt des Fahrens aber auch beim Tanken. Eine Anwendung kann dann z.B. die Tankuhr danach stellen.

Ereignisse zählen zu den Eigenschaften einer Delphi-Klasse, ihr Typ ist eine Prozedur-Variable, sie enthält einen Zeiger auf eine Methode. Ein Ereignis auslösen heißt dann, dass das Objekt überprüft, ob der Zeiger mit einer Methode verknüpft wurde, und wenn ja, diese aufruft. Schwierig?

Wie der Typ dieser Prozedur-Variablen aussehen soll, muss außerhalb der Klasse *TAuto* deklariert werden:

```
Type
  TTankChangeEvent = procedure(Sender: TObject; Menge: integer) of object;
```

Jedes Ereignis muss den Auslöser als Parameter (*Sender*) mitgeben. Unser Ereignis soll dazu noch die aktuelle Benzinmenge übermitteln. Der Rest läuft ab wie bei den anderen Eigenschaften von *TAuto*: Zur Eigenschaft *onTankChange* vom Typ *TTankChangeEvent* gehört eine Variable *FOnTankChange*, die den Prozedur-Zeiger enthält. Hinter den Direktiven *read* und *write* wird nur diese Variable angegeben, keine Methode. Es wird ja nur der Zeiger gelesen oder gesetzt.

```
type
  TAuto = class(TPaintbox)
  private
    . . .
    FOnTankChange: TTankChangeEvent;
  public
    . . .
    property OnTankChange: TTankChangeEvent
       read FOnTankChange write FOnTankChange;
  . . .
```

Zwei Fragen bleiben noch: Wo und wie wird das Ereignis ausgelöst? Und wie verknüpfe ich damit eine Ereignis-Methode? *TAuto* hat eine Methode *schritt*, die immer aufgerufen wird, wenn es einen Pixel weiterfahren soll (also bei *onTimer)*. Dabei „verbraucht es Sprit“, also lösen wir hier *onTankChange* aus. *TAuto* hat eine Methode *Tanken*, dabei ändert sich der Tankinhalt, also lösen wir hier *onTankChange* aus. Beide Male gleich:

```
if assigned(FOnTankChange) then FOnTankChange(Self, round(FBenzinmenge));
```

assigned prüft, ob *FOnTankChange* mit einer Methode verknüpft ist oder auf „nil“ steht. Im ersten Fall wird die Methode ausgeführt mit Übergabe des *Sender* (=*Self* = *TAuto*-Objekt) und der Benzinmenge. Verknüpfen kannst du das Ereignis in *Unit1* innerhalb der Methode *TForm1.FormCreate* z.B. so:

```
meinAuto := TAuto.create(Form1, clblue, atCabrio, 'Golf');
meinAuto.OnTankChange := stelleTankuhr;
```

mit einer Methode *Form1.stelleTankuhr*, welche die übergebene Benzinmenge auswertet.

So. Jetzt wird es aber allerhöchste Zeit, dass ich das Ereignis *onBuchFertig* der Klasse *TWahnsinn* mit einer Methode *TBertram.TrinkeFeinesHelles(Sender: TSchwarzbräu)* verknüpfe, sonst könnte der Zeiger noch auf „nil“ stehen, wenn es eintritt.