3 Das Delphi-Labor

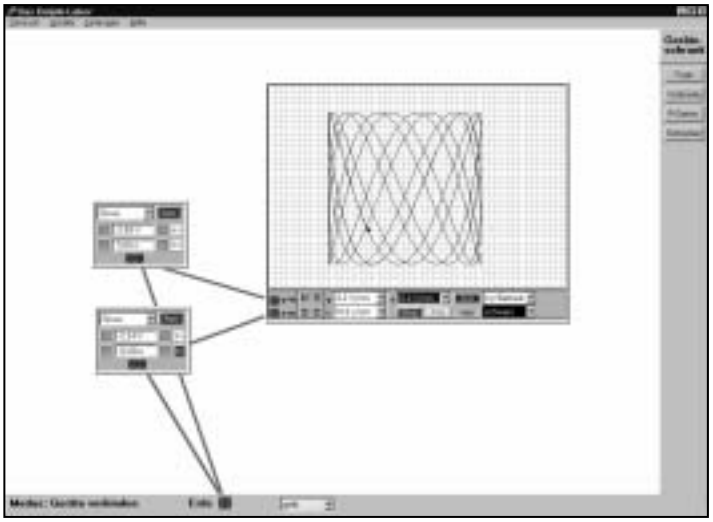
Die Nutzung wenigstens eines Teils der Möglichkeiten von Delphi soll zuletzt an einem komplexeren Beispiel gezeigt werden: einem simulierten *Elektronik-Labor* mit diversen Geräten, die mit einander am Bildschirm verbunden werden und dort arbeiten können. In diesem Beispiel werden

- diverse Objektklassen definiert, die den bei der Arbeit oft benutzten Geräteteilen entsprechen – also *Buchsen*, *Anzeigefelder*, *Knöpfe*, *Schalter*, ...
- Objekte (*Geräte*) bei Bedarf auf Mausklick erzeugt.
- polymorphe Methoden (*Arbeite*) benutzt, die für jedes Gerät anders sind.
- die erzeugten Geräte in einer dynamischen Objektklasse (einer *Geräteliste*) abgelegt.
- Bibliotheken geschrieben, die auf einer zum Zeitpunkt der Übersetzung noch unbekannten *Canvas* zeichnen, nämlich der des *Parent*-Objekts.
- unterschiedliche *Event-Handler*-Methoden zur Bearbeitung der benutzten Mausereignisse eingesetzt.
- Zeiger zur interaktiven Verknüpfung der Geräte gesetzt.

Insgesamt wird reichlich von den vorher im Buch gemachten Erfahrungen Gebrauch gemacht.

3.1 Anforderungen an das Delphi-Labor

Weil nichts aussagekräftiger als ein gutes Beispiel ist, wollen wir uns einmal den Bildschirmausdruck einer möglichen Situation im Delphi-Labor ansehen. Erzeugt wurden ein *xyt-Schreiber* und zwei *Funktionsgeneratoren*, die mit dem x- bzw. y-Eingang des Schreibers verbunden sind. Weil beide Generatoren Sinusschwingungen unterschiedlicher Frequenz erzeugen, ergibt sich eine der *Lissajous-Figuren*.



Zur Realisierung des Delphi-Labors müssen wir einige Vorbereitungen treffen:

- Wenn wir nicht jedes Gerät von Anfang an wieder neu „erfinden“ wollen, dann müssen wir eine „Kiste“ mit Bauteilen – eine Unit – zur Verfügung haben, aus der wir wieder verwendbare Komponenten entnehmen können. Beispiele finden wir schon reichlich auf dem Bildschirmausdruck: alle Geräte enthalten Knöpfe und Schalter, Auswahlmöglichkeiten, Anzeigeflächen usw. Die erforderlichen programmtechnischen Mechanismen für selbst geschriebene Units kennen wir z. B. von den *dynamischen Objektklassen* her.

- So wie es aussieht, wird das Programm nicht gerade kurz. Der Übersichtlichkeit halber werden wir deshalb auch die einzelnen Geräte in wieder verwendbaren Units unterbringen. Das hat aber zur Folge, dass die grafischen Ausgaben, z. B. die Darstellung der Geräte selbst und die der aktuellen Messwerte, nicht auf einem Formular der Unit geschehen können, sondern an anderer Stelle. Wir müssen dafür geeignete Verfahren vorsehen, ähnlich wie beim *Memory-Spiel*.

- Die Geräte und ihre Bauteile sollen auf Mausklicks reagieren – das gehört zur Standardarbeitsweise von Delphi. Sie sollen aber unterschiedlich reagieren, je nachdem welche Funktion z. B. ein Schalter auf einem Gerät auslösen soll. Auch solche Verfahren wurden eingeführt – z. B. beim *Memory-Spiel*.

- Zusätzlich müssen viele Geräte unabhängig von Mausereignissen weiterarbeiten. Sie benötigen also eine innere Uhr, die zu vorgegebenen Zeitpunkten Ereignisse auslöst. Auf das Verfahren dafür sind wir im ersten Band bei den *bewegten Bällen* gestoßen.

- Die Arbeitsweise von verschiedenen Geräten mit „innerer Uhr“ ist unterschiedlich. Trotzdem müssen sie alle „arbeiten“, und ihre Unterschiede liegen genau in dieser *polymorphen* Methode. Wir werden also Methoden gleichen Namens, aber unterschiedlicher Wirkungsweise benutzen: virtuelle Methoden.

- Wenn wir Geräte am Bildschirm „aus dem Geräteschrank holen“, also dynamisch erzeugen, dann können wir die tatsächlich auf dem Bildschirm befindliche Gerätezahl vor dem Programmstart nicht kennen. Wir können dann die Geräte auch nicht unter einem eigenen Namen ansprechen, sondern benötigen eine Geräteliste, in die neue Geräte eingefügt werden. Ebenso sollten bei einer sinnvollen Verkabelungsmöglichkeit am Bildschirm die Leitungen übersichtlich gezeichnet werden (und nicht so rudimentär wie auf dem Bildschirmausdruck). Beides erfordert *dynamische Objektklassen*, die wir aus diesem Band kennen.

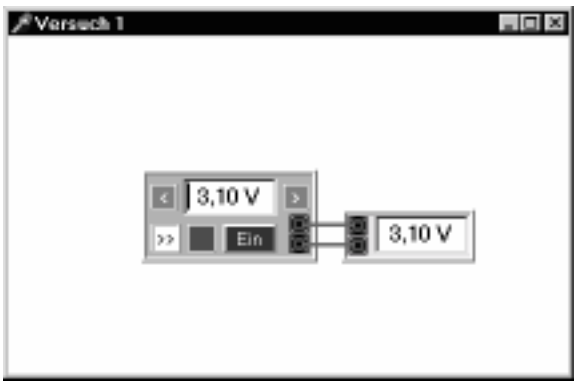
Die – jetzt nur grob umschriebenen – Aufgaben können wir im Buch nicht vollständig lösen. Wir wollen aber wenigstens einige wesentliche Teilprobleme angehen und die restlichen Lösungen skizzieren. Anzumerken ist auch, dass die Simulation von Geräten zu den einfachen Simulationsproblemen gehört – sie stellt eher programmtechnische Probleme.

Die Simulation „echter“ Schaltungen, sogar schon von am Bildschirm erzeugten Stromkreisen mit einfachen Widerständen, stellt Probleme ganz anderer Größenordnung – z. B. die Berechnung von Netzwerken - und führt auf mathematische Fragen, die wir hier nicht angehen können (und wollen).

3.2 Die Klassenhierarchie

3.2.1 Zwei einfache Geräte

Wir wollen „top-down“ vorgehen und die Eigenschaften der erforderlichen Objektklassen aus Problemen ableiten. Dazu erzeugen wir zuerst statisch zwei Geräte am Bildschirm, die wir logisch (also vom Programm her) „verdrahten“, und dann auch optisch durch zwei Leitungen, die aber (noch) keine weitere Funktion haben. Das Ergebnis sehen wir rechts.



Wir müssen dazu die beiden Geräte und die Leitungen auf einem Formular erzeugen, und das funktioniert natürlich nur, wenn wir entsprechende Bibliotheken zur Verfügung haben. Weitere Aktionen sollten nicht erforderlich sein, da die Geräte sich dann am Bildschirm wie die simulierten echten verhalten sollten, also einfach „funktionieren“. Wir wollen annehmen, dass die Grundbausteine in unserer „Werkzeugkiste“, einer Unit *uTools*, bereitstehen und die aus deren Bestandteilen zusammengesetzten Geräte in den Units *uTrafo* und *uDVoltmeter*. Mit diesen Hilfsmitteln lautet der vollständige Programmtext:

```
unit uVersuch1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs, uTools, uTrafo, uDVoltmeter;

type TVersuch1 = class(TForm)
     procedure FormCreate(Sender: TObject);
     private
     Trafo    : tTrafo;
     Voltmeter: tDVoltmeter;
     Leitung1 : tLeitung;
     Leitung2 : tLeitung;
     end;
```

Bibliotheken angeben

Geräte vereinbaren

```
var Versuch1: TVersuch1;
implementation
{$R *.DFM}

  procedure TVersuch1.FormCreate(Sender: TObject);
  begin
  Trafo     := tTrafo.Init(self,100,100,clAqua);
  Voltmeter := tDVoltmeter.Init(self,250,130,clYellow);
  Leitung1  := tLeitung.Init(self,220,140,260,143,clFuchsia);
  Leitung2  := tLeitung.Init(self,220,155,260,158,clFuchsia);
  Voltmeter.E1.Verbindung := Trafo.Ausgang;
  end;
end.
```

Geräte und Leitungen erzeugen

Geräte logisch verbinden

Bei der Instantiierung der Geräte mit dem *Init-Konstruktor* wird jedem Gerät das *Parent*-Objekt übergeben, dem das Gerät untergeordnet ist, auf dem es sich also zeigen soll. In unserem Fall handelt es sich dabei um das aktuelle Formular, das durch den Parameter *self* in jeder seiner Methoden referenziert wird. Die restlichen Parameter geben die linke obere Ecke des Geräts (also seine Bildschirmposition) und die Farbe an. Die Endpunkte der Leitungen finden wir durch Ausprobieren.

3.2.2 UML-Diagramme

Beide Geräte verfügen über Gemeinsamkeiten: Eine *Anzeige* zur Darstellung der aktuellen Werte und *Buchsen* zur Verbindung mit anderen Geräten. Sie haben auch beide eine *Bildschirmposition*, eine *Farbe* und sicher noch andere Eigenschaften. Zusätzlich verfügt der Trafo über Knöpfe (z. B. um die Ausgangsspannung zu verändern) und einen Schalter (zum Ein- und Ausschalten). Die Gemeinsamkeiten wollen wir in einer (generischen) Objektklasse *tGeraet* zusammenfassen, von der wir dann die speziellen Geräte ableiten. *tGeraet* selbst umfasst einige Eigenschaften, die schon in den *Panel*-Komponenten vorhanden sind. *tPanel* soll also die Mutterklasse unser Geräte sein. Aus entsprechenden Überlegungen finden wir die anderen Objektklassen.

Die Eigenschaften von Klassen und ihr Zusammenhang werden oft in der *Unified Modeling Language UML* beschrieben. Leider gibt es davon verschiedene Versionen, die entsprechend der jeweiligen Anforderung meist ziemlich frei benutzt werden. Wir wollen eine einfache Variante wählen.

Die UML stellt Klassen durch Rechtecke dar, die in zwei Sektionen geteilt sind:

- zuerst werden die Attribute der Klasse angegeben
- danach folgen die Methoden.

Bei Bedarf lassen wir Attribute und Methoden weg, z. B. wenn wir nur den Zusammenhang zwischen den Klassen veranschaulichen wollen.

Eine Klasse von „Geräten“, bei der hier der Einfachheit halber alle Attribute als *public* deklariert sind, könnte wie folgt beschrieben werden:

tGerät
+ farbe: tColor + mitBild: boolean + mitUhr: boolean + mitRand: Boolean + amVerschieben: boolean + Uhr: tStoppuhr + Bild: tImage + Rand: integer + xOffset: integer + yOffset: integer
<< constructor >> + init(integer,integer,integer,integer,tColor) <<method >> + arbeite(tObject) + mausRunter(tObject,tMouseButton,tShiftState,integer,integer) + mausHoch(tObject,tMouseButton,tShiftState,integer,integer) + mausBewegung(tObject,tShiftState,integer,integer)

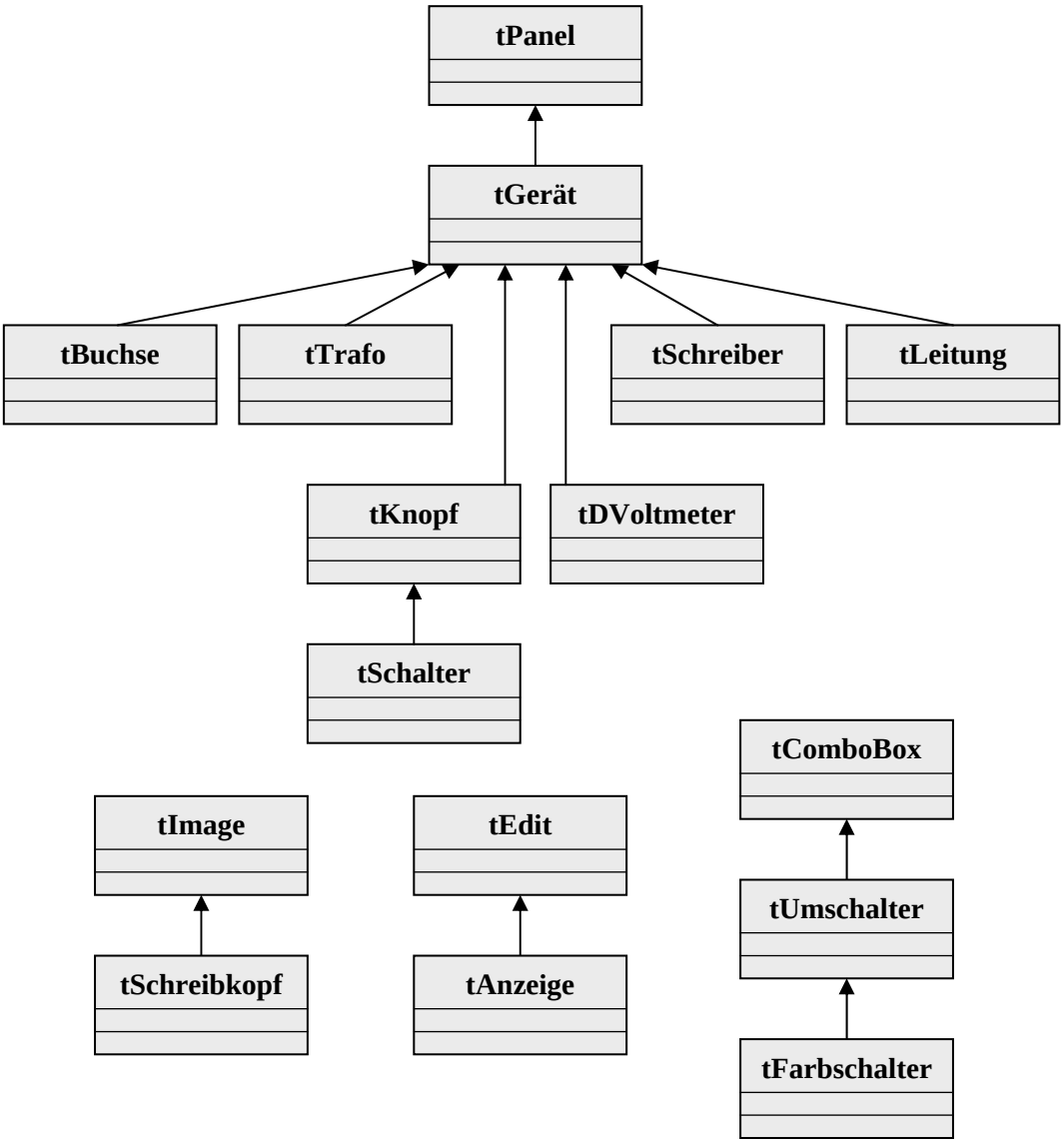
Die „Vorzeichen“ vor den Attributen und Methoden beschreiben die *Sichtbarkeit* dieser Elemente:

- öffentlich zugängliche Elemente werden als *public* deklariert und erhalten ein vorangestelltes „+“
- interne Elemente werden als *private* deklariert und erhalten ein „-“
- vor Veränderungen geschützte Elemente werden als *protected* deklariert und erhalten ein „#“

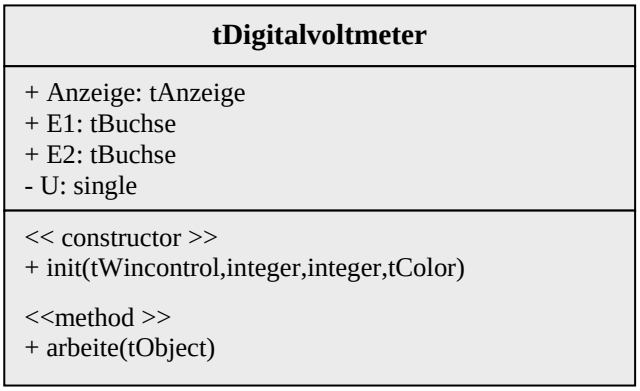
Die Abschnitte mit Konstruktoren bzw. normalen Methoden werden wie angegeben mit den Schlüsselworten << *constructor* >> bzw. << *method* >> bezeichnet. Die Typen den Parameterlisten der Methoden werden in der gewählten Reihenfolge angegeben.

Der Zusammenhang zwischen Klassen, also ihre Hierarchie, wird durch Pfeile angegeben. Von abgeleiteten *Tochterklassen* führt ein Pfeil zur *Mutterklasse*, die ihre Eigenschaften vererbt. Schreiben wir UML-Diagramme mit allen Eigenschaften und Methoden

vollständig auf, dann bekommen wir schnell Platzprobleme. Es ist deshalb oft sinnvoll, nur die Klassennamen anzugeben und die restlichen Elemente durch leere Kästchen anzudeuten. Für unsere Gerätehierarchie wollen wir so verfahren – schon deshalb, weil wir die erforderlichen Attribute und Methoden derzeit noch nicht kennen.



Beginnen wollen wir mit dem Digitalvoltmeter – dem einfachsten Gerät. Es enthält als Bauteile eine *Anzeige* und zwei *Buchsen*. Als Methoden stehen ihm (neben den von *tGeraet* geerbten) ein Konstruktor *Init* zur Verfügung, der das Gerät mit den Vorgabewerten erzeugt, und eine virtuelle Methode *arbeite*, die die Arbeitsweise des Gerätes beschreibt. Als interne Größe wird ein Spannungswert *U* benutzt.



Da alle Geräte in eigene Units gepackt werden sollen, geschieht das auch hier. Die Werkzeugkiste *uTools* muss wie üblich in der *uses*-Klausel angegeben werden.

```
unit uDVoltmeter;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs, ExtCtrls, Menus, StdCtrls, uTools;

type tDVoltmeter = class(tGeraet)
       Anzeige: tAnzeige;
       E1, E2 : tBuchse;
       constructor Init(aOwner: tWinControl;
                        x,y: integer; f: tColor); virtual;
       procedure Arbeite(sender: tObject); override;
       private
       u: single;
       end;
```

Die Arbeitsweise eines Voltmeters ist denkbar einfach: Es zeigt die Spannungsdifferenz der Eingänge an. Ist ein Eingang nicht verdrahtet, dann liegt er auf dem „Erdpotential“ 0.

```
procedure tDVoltmeter.Arbeite;
begin
Anzeige.ZeigeZahl(E1.u-E2.u,7,2,'V')
end;
```

Differenz der Eingangswerte anzeigen

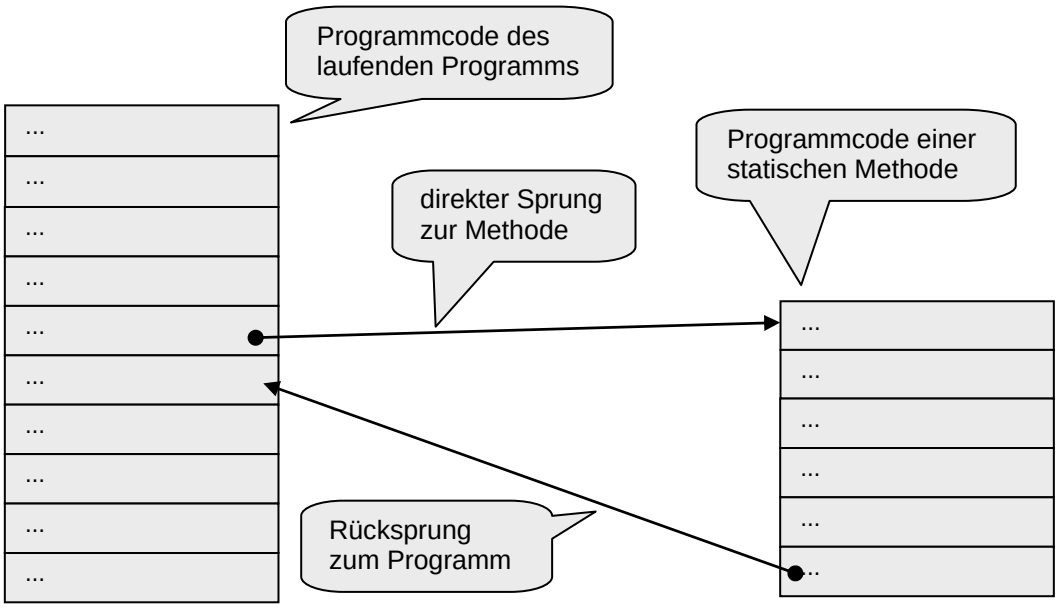
Der Konstruktor ruft den geerbten Konstruktor auf (die Bedeutung der Parameter werden wir uns später ansehen) und erzeugt danach die Komponenten des Gerätes.

```
constructor tDVoltmeter.Init(aOwner: tWinControl; x,y: integer;
                                                   f: tColor);
begin
inherited Init(aOwner,x,y,100,40,f,Arbeite,true,false,true);
u := 0;
Anzeige := tAnzeige.Init(self,'',25,5,70,20,10);
Anzeige.ZeigeZahl(u,7,2,'V');
E1 := tBuchse.Init(Self,btEingang,5,5,7,clRed);
E2 := tBuchse.Init(self,btEingang,5,20,7,clgreen);
uhr.starte(100);
end;
```

Komponenten erzeugen

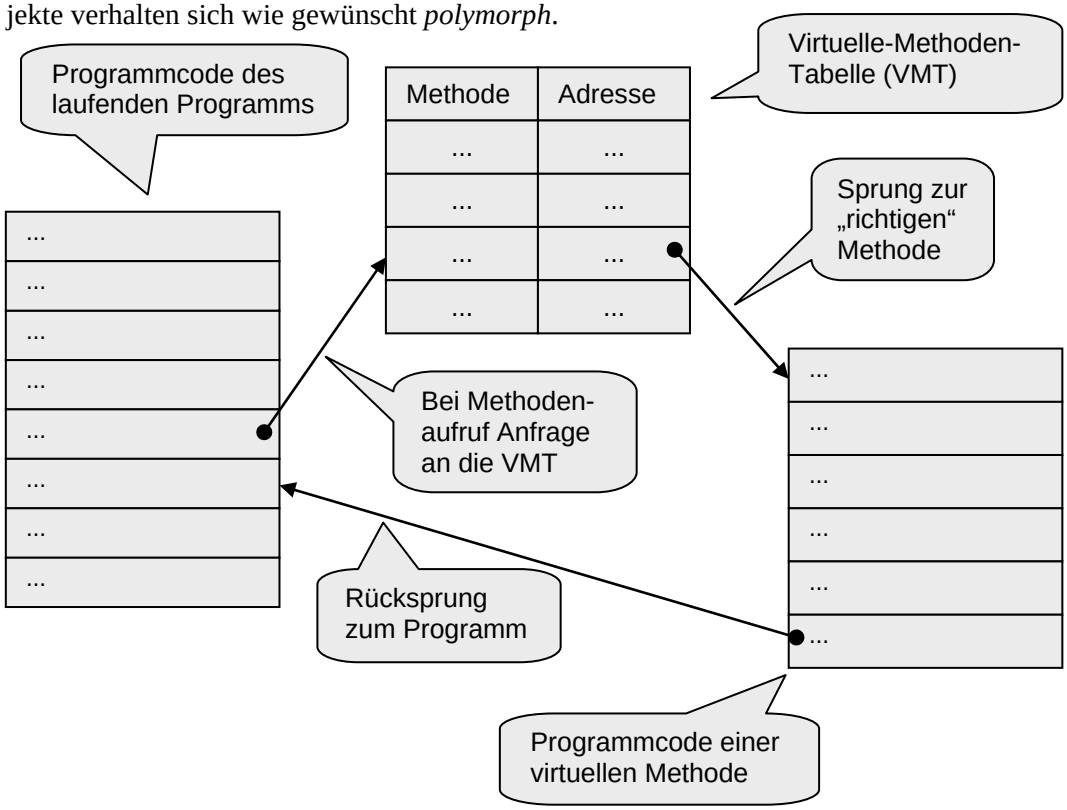
3.3 Virtuelle Methoden

Vereinbart man in einer Klasse eine Methode und übersetzt den Delphicode, dann „springt“ das laufende Programm beim Aufruf dieser Methode zur entsprechenden Adresse im Programmcode und führt die dort aufgeführten Befehle aus. Man nennt solche Methoden *statisch*, weil nach der Übersetzung die Einsprungadresse festliegt und nicht mehr verändert werden kann. Der Vorteil des Verfahrens liegt in seiner Geschwindigkeit: die Methoden werden *direkt* angesprungen.



Bei unseren Geräten ist dieses Verfahren für die Methode *Arbeite* unbrauchbar, weil ja – je nach Geräteklasse – unterschiedliche Methoden benutzt werden müssen. Dem Delphi-Compiler muss deshalb mitgeteilt werden, dass *erst zur Laufzeit* bestimmt werden kann, welche der unterschiedlichen *Arbeite*-Methoden die richtige ist. Die Lösung des Problems liegt in der Benutzung von *virtuellen Methoden* und einer *Virtuelle-Methoden-Tabelle* (VMT). In der VMT sind alle virtuellen Methoden einer Klasse aufgeführt, und zusätzlich die Adressen, unter denen die für diese Klasse gültigen Methoden zu finden sind. Da schon *tObject* über virtuelle Methoden verfügt, besitzt *jede* Klasse von Delphi eine VMT. VMTs werden automatisch erzeugt.

Statt beim Aufruf einer Methode direkt zum Programmcode zu springen, wird bei virtuellen Methoden *vor dem Sprung* in der VMT *der Klasse des aufrufenden Objekts* nachgesehen, an welcher Stelle sich die für die Klasse gültige virtuelle Methode befindet. Diese wird dann benutzt. Da Tochterklassen eigene virtuelle Methoden unter dem gleichen Namen (und mit der gleichen Parameterliste) wie die Mutterklasse einführen können, verfügen die Methoden in ihren VMTs jeweils über unterschiedliche Adressen, so dass durch den Aufruf der gleichen Methode unterschiedliche Aktionen ausgelöst werden: die Objekte verhalten sich wie gewünscht *polymorph*.



Wenn Tochterklassen eine Methode als virtuell (*virtual*) deklarieren, dann wird die Methode der Mutterklasse nur *verborgen* und kann – ähnlich wie die Konstruktoren – unter Voranstellung des Namens der Mutterklasse immer noch aufgerufen werden. Soll der geerbte Programmcode wirklich ersetzt (*überladen*) werden, dann benutzt man statt *virtual* das Schlüsselwort *override*.

Oft werden in den übergeordneten Klassen virtuelle Methoden als gemeinsame Eigenschaften der Tochterobjekte vereinbart, ohne an dieser Stelle des Entwurfs schon implementiert zu werden. Statt dort „leere" Methoden zu schreiben (bestehend nur aus der Befehlsfolge *begin end*) kann auch das Schlüsselwort *abstract* benutzt werden, das die Methode als *abstrakt* kennzeichnet. Abstrakte Methoden werden erst in Tochterklassen wirklich implementiert.

In der folgenden Unit *uTools* wird die *Arbeite*-Methode der Klasse *tGeraet* als virtuell und abstrakt vereinbart. Die Tochterklassen der „echten" Geräte implementieren die Methode dann geeignet. Die Uhren der Geräte rufen die Methode ggf. periodisch auf, um das Funktionieren der Geräte zu gewährleisten.

3.4 Die Werkzeugkiste *uTools*

Mit den jetzt bekannten Anforderungen an die Bestandteile von Geräten können wir die gesuchte Werkzeugkiste *uTools* erzeugen, in der alle benötigten Bauteile und Methoden enthalten sind. Wir beginnen mit der Klasse *tGeraet*.

3.4.1 Die Klasse der Geräte

Geräte enthalten eine *Bildschirmposition*, eine *Breite* und eine *Höhe* – alles Eigenschaften der Mutterklasse *tPanel*. Auch die *Farbe* der Geräte könnten wir in deren Eigenschaften unterbringen. Weil wir aber ab und zu die Farbe ändern wollen (z. B. beim Verschieben der Geräte), speichern wir die Gerätefarbe auch als besondere Eigenschaft des Geräts. Weitere Eigenschaften dienen der „Verschönerung". Geräte können mit *Rand* und ggf. mit einem *Bild* dargestellt werden, z. B. wenn wir auf dem rechteckigen Panel runde Buchsen zeichnen. Viel wesentlicher ist aber die eingebaute *Uhr*, die das Funktionieren des Geräts steuert. Weniger wichtig sind einige Hilfseigenschaften, die z. B. das Verschieben der Geräte erleichtern. Das UML-Diagramm der Klasse wurde oben angegeben.

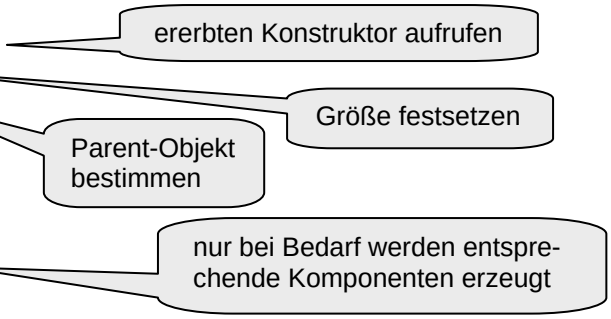
Zur Erzeugung von Geräten dient der Konstruktor *Init*. Dieser übernimmt beim Aufruf unter dem Namen *aOwner* das *Parent*-Objekt, auf dem sich das Gerät zeigen soll, und die erforderlichen Werte für die Panel-Mutterklasse. Zusätzlich erhält der Konstruktor eine Ereignisbehandlungsroutine *func*, die von der Uhr regelmäßig gestartet wird. Die Reaktionen auf Mausereignisse werden durch drei weitere Event-Handler definiert: *Maus-*

Runter, *MausBewegung* und *MausHoch* werden den Ereignissen *OnMouseDown*, *OnMouseMove* und *OnMouseUp* zugeordnet. Die Parameterlisten dieser Methoden sind für die entsprechenden Mausereignisse fest vorgegeben. Eine virtuelle Methode *arbeite* wird von jedem Gerät neu definiert, um die eigene Arbeitsweise zu beschreiben.

```
tGeraet = class(tPanel)
   public
   Farbe                    : tColor;
   MitBild, MitUhr, MitRand: boolean;
   Uhr                      : tStoppuhr;
   Bild                     : tImage;
   Rand                     : integer;
   AmVerschieben            : boolean;
   xOffset, yOffset         : integer;
   constructor Init(aOwner: tWinControl; x,y,b,h: integer;
                    f: tColor; func: tNotifyEvent;
                    WithClock,WithImage,WithBevel: boolean);
   procedure arbeite(sender: tObject); virtual; abstract;
   procedure MausRunter(Sender: TObject; Button: TMouseButton;
                        Shift: TShiftState; X, Y: Integer);
   procedure Mausbewegung(Sender: TObject;
                         Shift: TShiftState; X,Y: Integer);
   procedure MausHoch(Sender: TObject; Button: TMouseButton;
                      Shift: TShiftState; X, Y: Integer);
   end;
```

Bei der Implementierung der Methoden werden eigentlich nur die wichtigen Eigenschaften der Objektklasse gesetzt. Zusätzlich werden die Event-Handler zugewiesen und Verschönerungsarbeiten bei der Darstellung ausgeführt, die wir hier nicht im Einzelnen angeben wollen.

```
constructor tGeraet.Init(aOwner: tWinControl; x,y,b,h: integer;
                         f: tColor; func: tNotifyEvent;
                         WithClock,WithImage,WithBevel: boolean);
begin
inherited create(aOwner);
SetBounds(x,y,b,h);
parent   := aOwner;
name     := GetNewName;
Farbe    := f;
color := f;
caption  := '';
MitUhr   := WithClock;
MitBild  := WithImage;
MitRand  := WithBevel;
```



```
if MitUhr then uhr := tStoppuhr.Init(self,func);
AmVerschieben := false;
```

Arbeitsweise der Uhr festlegen

{... hier stehen Verschönerungsarbeiten ...}

```
if MitBild then
  begin
  Bild := tImage.Create(self);
  Bild.parent := self;
```

{... hier stehen weitere Verschönerungsarbeiten ...}

```
  end;
```

Reaktionen auf Mausereignisse festlegen

```
OnMouseDown := MausRunter;
OnMouseMove := Mausbewegung;
OnMouseUp   := MausHoch;
end;
```

```
procedure tGeraet.MausRunter(Sender: TObject;
       Button: TMouseButton;Shift: TShiftState; X, Y: Integer);
begin
if not (Modus in [Setzen,Verschieben]) then exit;
AmVerschieben := true;
color := clGray;
xOffset := x; yOffset := y;
end;
```

Beim Drücken der Maustaste auf das Verschieben des Geräts vorbereiten

```
procedure tGeraet.Mausbewegung(Sender: TObject;
                               Shift: TShiftState; X,Y: Integer);
begin
if AmVerschieben then
  begin
  left := left+x-xOffset;
  top := top+y-yOffset;
  end
end;
```

Bei Mausbewegungen das Gerät ggf. verschieben

```
procedure tGeraet.MausHoch(Sender: TObject;
       Button: TMouseButton;Shift: TShiftState; X, Y: Integer);
begin
if not AmVerschieben then exit;
color := farbe;
AmVerschieben := false;
end;
```

Verschieben beenden