Eine Einführung in die Grundlagen der Objektorientierten Programmierung

Dr. Jürgen Czischke, Dr. Georg Dick, Horst Hildebrecht, Ludger Humbert, Werner Ueding, Klaus Wallos.

Bearbeitung: Dieter Lindenberg

Version 2011

Inhalt

Die Klasse „Bildschirm" ... 3

Die Klasse Stift ... 4

Prozeduren ... 11

Prozeduren mit Parametern ... 14

Die Klasse Maus ... 17

Bedingungen ... 18

  Logische Operatoren NOT, AND, OR ... 20

Schleifen ... 21

Die Klasse Tastatur ... 27

Die Klasse „Buntstift" ... 31

Projekt Pfeilwurf ... 42

Entwicklung eigener Klassen ... 45

  Die Klasse Ball ... 46

  Die Klasse Muehle ... 52

Die Kennt-Beziehung ... 56

Vererbung als Spezialisierung ... 66

Abstrakte Klassen als Generalisation ... 80

Klassen und Konstruktoren ... 91

Die Klasse *AnwendungNeu* ... 95

  Die Klasse EreignisanwendungNeu ... 105

Die Klasse „Bildschirm“

Ein Bildschirm ist das Modell des angeschlossenen Computerbildschirms. Auf ihm kann mit Stiften gezeichnet werden. Zu diesem Zweck ist die Zeichenebene auf dem Bildschirm mit einem Koordinatensystem versehen, dessen Ursprung sich in der oberen linken Ecke der Zeichenebene befindet und dessen Achsen horizontal nach rechts und vertikal nach unten gerichtet sind. Die Einheit ist ein Pixel.

Bildschirm
-Höhe -Breite
+ ! init + ? breite + ? hoehe + ! loescheAlles + ! gibFrei

Auftrag init
nachher Der Bildschirm ist mit seiner Zeichenebene initialisiert.

Anfrage breite : GanzeZahl
nachher Diese Anfrage liefert die Breite der Zeichenebene.

Anfrage hoehe : GanzeZahl
nachher Diese Anfrage liefert die Höhe der Zeichenebene.

Auftrag loescheAlles
nachher Die Zeichenebene ist leer.

Auftrag gibFrei
nachher Nachdem eine Taste oder der Mausknopf gedrückt wurde, steht der Bildschirm nicht mehr zur Verfügung, d.h. die Zeichenebene verschwindet. (Es erscheint die Meldung *„Programm beendet“*)

Die Klasse Stift

Der Stift ist ein Werkzeug, das sich auf dem Bildschirm bewegen kann. Er befindet sich stets auf einer genau definierten Position des Bildschirms, die durch Zeichenkoordinaten (horizontal nach rechts, vertikal nach unten) angegeben wird, und zeigt in eine Richtung, die durch Winkel beschrieben wird (0° entspricht der Richtung nach rechts, Drehsinn ist mathematisch positiv).

Der Stift kennt zwei Zustände: Ist der Stift abgesenkt (runter) und bewegt er sich über den Bildschirm, so hinterlässt er eine Spur, die von einem Zeichenmodus abhängig ist. Ist der Stift angehoben (hoch), hinterlässt er keine Spur.

Beim Zeichnen kennt der Stift drei Modi:

- Normal: der Stift zeichnet eine Linie in der Stiftfarbe;
- Wechseln: der Stift zeichnet eine Linie, wobei die Untergrundfarbe in die Stiftfarbe und die Stiftfarbe in die Untergrundfarbe geändert wird;
- Radieren: der Stift zeichnet eine Linie in der Farbe des Untergrunds.

```
Stift
-----------------------
# Position
# Richtung
# Zustand
# Modus
-----------------------
+ ! init
+ ! bewegeUm()
+ ! bewegeBis()
+ ! dreheUm()
+ ! dreheBis()
+ ! runter
+ ! hoch
+ ! schreibeZahl()
+ ! schreibeText()
+ ! normal
+ ! wechsle
+ ! radiere
+ ! zeichneRechteck()
+ ! zeichneKreis()
+ ? hPosition
+ ? vPosition
+ ? winkel
+ ! gibFrei
```

Auftrag init

nachher Der Stift ist initialisiert. Die Zeichenebene steht zur Verfügung und der Stift befindet sich angehoben oben links an Position (0,0) mit Richtung 0° im normalen Zeichenmodus.

Auftrag bewegeUm (pDistanz : Zahl)

nachher Der Stift wurde von seiner aktuellen Position in die aktuelle Richtung bewegt. "pDistanz" gibt die Länge der zurückgelegten Strecke an.

Auftrag bewegeBis (ph, pv : Zahl)

nachher Der Stift wurde unabhängig von seiner vorherigen Position auf die durch die Parameter angegebene Position bewegt. Die Winkelstellung des Stiftes hat sich nicht geändert.

Auftrag dreheUm (pWinkel : Zahl)

nachher Der Stift wurde ausgehend von seiner vorherigen Richtung um die durch "pWinkel" angegebene Winkelgröße im mathematisch positiven Sinne weitergedreht.

Auftrag dreheBis (pWinkel : Zahl)

nachher Der Stift wurde unabhängig von seiner vorherigen Richtung auf die durch "pWinkel" angegebene Winkelgröße gedreht.

Auftrag runter

nachher Der Stift ist abgesenkt.

Auftrag hoch

nachher Der Stift ist angehoben.

Auftrag schreibeText (pText : Zeichenkette)

nachher Der Stift hat den angegebenen Text auf die Zeichenebene unter Verwendung seines aktuellen Zeichenmodus unabhängig vom Zustand geschrieben. Die aktuelle Stiftposition war die linke obere Ecke des Textes. Die neue Stiftposition ist die rechte obere Ecke des Textes.

Auftrag schreibeZahl (pZahl : Zahl)

nachher Der Stift hat die angegebene Zahl auf die Zeichenebene unter Verwendung seines aktuellen Zeichenmodus unabhängig vom Zustand geschrieben. Die aktuelle Stiftposition war die linke obere Ecke der Zahl. Die neue Stiftposition ist die rechte obere Ecke der Zahl.

Auftrag normal

nachher Der Stift arbeitet im Normalmodus.

Auftrag wechsle
nachher Der Stift arbeitet im Wechselmodus.

Auftrag radiere
nachher Der Stift arbeitet im Radiermodus.

Auftrag zeichneRechteck (pBreite, pHoehe : Zahl)
nachher Der Stift hat unabhängig von seinem Zustand im aktuellen Zeichenmodus ein achsenparalleles Rechteck mit der aktuellen Position als linke obere Ecke und der angegebenen Breite und Höhe gezeichnet. Die Position und die Richtung des Stiftes sind unverändert.

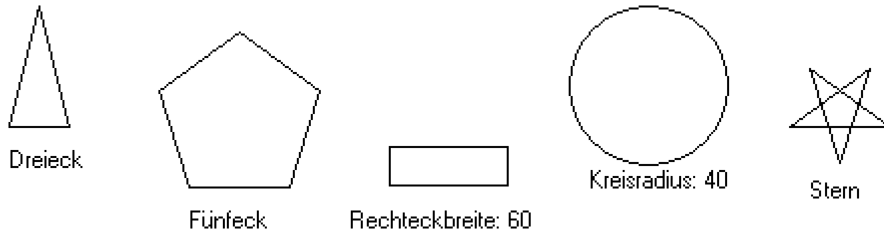
Auftrag zeichneKreis (pRadius : Zahl)
nachher Der Stift hat unabhängig von seinem Zustand im aktuellen Zeichenmodus einen Kreis mit der aktuellen Position als Mittelpunkt und dem angegebenen Radius gezeichnet. Die Position und die Richtung des Stiftes sind unverändert.

Anfrage hPosition : Zahl
nachher Diese Anfrage liefert die horizontale Koordinate der aktuellen Stiftposition.

Anfrage vPosition : Zahl
nachher Diese Anfrage liefert die vertikale Koordinate der aktuellen Stiftposition.

Anfrage winkel : Zahl
nachher Diese Anfrage liefert die momentane Bewegungsrichtung des Stifts.

Auftrag gibFrei
nachher Der Stift steht nicht mehr zur Verfügung.



Aufgabe: Zeichne obige Figuren (mit zugehörigem Text) auf den Bildschirm!

Lösung:

```
unit mHaupt;
interface
uses …….Graphics, Controls, Forms, Dialogs, mSuM;
type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  Main: TMain;
  meinSchirm: Bildschirm;
  meinStift: Stift;

implementation

{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
  meinSchirm := Bildschirm.init;
  meinStift := Stift.init;

  meinStift.bewegeBis(10,80);
  meinStift.schreibeText('Dreieck');
  meinStift.bewegeBis(10,70);
  meinStift.runter;
  meinStift.bewegeBis(40,70);
  meinStift.bewegeBis(25,10);
```

```
meinStift.bewegeBis(10,70);
meinStift.hoch;

meinStift.bewegeBis(100,110);
meinStift.schreibeText('Fünfeck');
meinStift.bewegeBis(100,100);
meinStift.runter;
meinStift.bewegeUm(50);
meinStift.dreheUm(72);
meinStift.bewegeUm(50);
meinStift.dreheUm(72);
meinStift.bewegeUm(50);
meinStift.dreheUm(72);
meinStift.bewegeUm(50);
meinStift.dreheUm(72);
meinStift.bewegeUm(50);
meinStift.dreheUm(72);
meinStift.hoch;

meinStift.bewegeBis(180,110);
meinStift.schreibeText('Rechteckbreite: ');
meinStift.schreibeZahl(60);
meinStift.bewegeBis(200,80);
meinStift.zeichneRechteck(60,20);

meinStift.bewegeBis(300,90);
meinStift.schreibeText('Kreisradius: ');
meinStift.schreibeZahl(40);
meinStift.bewegeBis(330,50);

meinStift.zeichneKreis(40);

meinStift.bewegeBis(410,95);
meinStift.schreibeText('Stern');
meinStift.bewegeBis(400,70);
meinStift.runter;
meinStift.bewegeUm(50);
meinStift.dreheUm(144);
```

```
    meinStift.bewegeUm(50);
    meinStift.dreheUm(144);
    meinStift.bewegeUm(50);
    meinStift.dreheUm(144);
    meinStift.bewegeUm(50);
    meinStift.dreheUm(144);
    meinStift.bewegeUm(50);
    meinStift.dreheUm(144);
    meinStift.hoch;

    meinStift.gibFrei;
    meinSchirm.gibFrei;
  end;

end.
```

Aufgaben

1. Zeichne ein symmetrisches Trapez, einen Drachen und eine Raute! Erstelle für die Trapezprogrammierung auch ein Struktogramm!

2. Zeichne ein Haus, ein Auto und die beiden folgenden Waggons!

- Ein Waggon wird am linken Puffer ganz links in der Mitte beginnend gezeichnet.
- Die Räder haben einen Radius von 5 Punkten (pixel).
- Der Wagen hat eine Gesamtlänge (einschließlich Puffer) von ungefähr 60 Pixel.

3. Zeichne mehrere identische Häuser nebeneinander!

4. Zeichne einen einzigen Punkt (als Kreis mit Radius 1 Pixel) an der Stelle (500, 50)!

5. Zeichne eine analoge Uhr mit Stunden- und Minutenzeiger! Es wird die Zeit $5.^{12}$ Uhr angezeigt. Am Kreisrand der Uhr befindet sich jeweils ein kleiner Strich für jede Stunde.

6. Die Breite und die Höhe deines Bildschirmes sollen von deinem Programm ermittelt werden und dein Stift soll diese Daten auf dem Bildschirm ausgeben.

7. Bringe den Stift an eine beliebige Stelle auf dem Bildschirm. Der Stift soll seine Orts- und Winkelkoordinaten auf dem Schirm ausgeben, z.B. in folgender Form: (512, 120, 25°).
Nach der Ausgabe (bei der sich der Stift ja bewegt) soll er zu eben diesem gerade von ihm angegeben Ort zurückkehren (und natürlich auch dieselbe Winkeleinstellung wie vorher haben).

8. Zeichne abwechselnd möglichst viele Quadrate und Kreise, die sich jeweils von innen berühren! Hinweis: Radien bzw. Kantenlängen (vorher mit dem Taschenrechner) mithilfe des Satzes von Pythagoras berechnen.

Prozeduren

Prozeduren sind Unterprogramme. Man fasst üblicherweise längeren Programmtext zu einer Prozedur zusammen, insbesondere dann, wenn dieser Programmtext mehr als einmal eingesetzt wird. Wir werden als ein Beispiel das Zeichnen eines gleichseitigen Dreiecks als Prozedur programmieren (implementieren). Alle Methoden einer Klasse werden durch eine (oder mehrere) Prozeduren (oder Funktionen) realisiert. Aber nicht jede kleine Hilfsprozedur ist eine Methode.

Zunächst gibt man bei der Klassendeklaration an, dass die Klasse *TMain* eine neue Methode besitzt. Ob diese Methode nun als *private* oder als *public* deklariert wird, ist zunächst unerheblich (wird aber später sehr wichtig!):

```
unit mHaupt;
interface
uses …….Graphics, Controls, Forms, Dialogs, mSuM;
type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
    private
      procedure zeichneDreieck;
  end;

var
  Main: TMain;
…………………………
```

Außerdem muss die neue Methode noch implementiert werden. Beachte, dass bei der Implementation dieser Methode noch zusätzlich der Klassenname mit angegeben werden muss. Das Delphi-Programm könnte ja mehrere Klassen beinhalten und es wüsste sonst nicht, zu welcher Klasse die Methode *Dreieck* gehört:

....

```
implementation

{$R *.dfm}

procedure TMain.zeichneDreieck;
begin
  meinStift.runter;
  meinStift.bewegeUm(40);
  meinStift.dreheUm(120);
  meinStift.bewegeUm(40);
  meinStift.dreheUm(120);
  meinStift.bewegeUm(40);
  meinStift.dreheUm(120);
  meinStift.hoch;
end;
```

Beachte: Nach dem Zeichnen des Dreiecks befindet sich der Stift in exakt derselben Position und derselben Richtung wie vorher.

Das Hauptprogramm könnte nun folgendermaßen lauten:

```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinSchirm := Bildschirm.init;
  meinStift := Stift.init;

  meinStift.bewegeBis(100,85);
  zeichneDreieck;
  meinStift.bewegeBis(300,200);
  meinStift.dreheUm(-30);
  zeichneDreieck;

  meinStift.gibFrei;
  meinSchirm.gibFrei
end;
```

Aufgaben

1. Warum ist es nicht sinnvoll, in obiger Dreiecksprozedur den Befehl *meinStift.bewegeBis(...)* zu benutzen?

Hinweis:
Bei jeder der folgenden Prozeduren sollte sich die Prozedur die Koordinaten des Anfangspunktes und den Anfangswinkel merken. Das ist mit den Methoden der Klasse *Stift* leicht möglich. Am Ende der Prozedur sollte man grundsätzlich zum Anfangspunkt und zum Anfangswinkel zurückkehren.

2. Implementiere eine Prozedur namens *zeichneHaus*! Zeichne anschließend mehrere Häuser auf dem Bildschirm!

3. Schreibe eine Prozedur, welche das Haus des Nikolaus zeichnet! Das Zeichnen wird einfacher, wenn man als Dach ein gleichseitiges Dreieck verwendet.

4. Schreibe eine Prozedur, welche ein rechtwinkliges, gleichschenkliges Dreieck ABC zeichnet! Der rechte Winkel soll bei B liegen. Es muss möglich sein, dass die Grundseite eine beliebige Richtung besitzt (also nicht unbedingt horizontal verläuft).

5. Schreibe eine Prozedur, welche ein rechtwinkliges, nicht gleichschenkliges Dreieck ABC zeichnet! Der rechte Winkel soll bei B liegen. Es muss möglich sein, dass die Grundseite eine beliebige Richtung besitzt (also nicht unbedingt horizontal verläuft).

Prozeduren mit Parametern

Bevor man die oben beschriebene Prozedur *zeichneDreieck* aufrufen konnte, musste man bekanntlich den Stift erst zu dem zukünftigen Ort des Dreiecks bewegen. Diese Ortsfestlegung kann man jedoch auch gleichzeitig mit dem Prozeduraufruf angeben. Das geschieht mithilfe von Parametern:

```
unit mHaupt;
interface
uses …….Graphics, Controls, Forms, Dialogs, mSuM;
type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
    private
      procedure zeichneDreieck(x, y: zahl);
  end;

var
  Main: TMain;
…………………………
……………………
implementation

{$R *.dfm}

procedure TMain.zeichneDreieck(x, y: zahl);
begin
  meinStift.bewegeBis(x, y);
  meinStift.runter;
  meinStift.bewegeBis(x+30, y);        // willkürliche Länge
  meinStift.bewegeBis(x+15, y-60); // willkürliche Höhe
  meinStift.bewegeBis(x, y);
  meinStift.hoch;
end;
```

Im Hauptprogramm kann man dann diese Dreiecksprozedur so aufrufen:

```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinSchirm := Bildschirm.init;
  meinStift := Stift.init;
  zeichneDreieck(10, 70);
  zeichneDreieck(100, 200);
  ................................ .
```

Man könnte natürlich auch noch mehr Parameter übergeben, z.B. die Länge der Grundseite und die Höhe des Dreiecks:

```
.................................
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
    private
      procedure zeichneDreieck(x, y, l, h: zahl);
  end;

........................
implementation

{$R *.dfm}

procedure TMain.zeichneDreieck(x, y, l, h: zahl);;
begin
  meinStift.bewegeBis(x, y);
  meinStift.runter;
  meinStift.bewegeBis(x+l, y);
  meinStift.bewegeBis(x + l/2, y-h);
  meinStift.bewegeBis(x, y);
  meinStift.hoch;
end;
```

```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinSchirm := Bildschirm.init;
  meinStift := Stift.init;
  zeichneDreieck(10, 70, 30, 60);
  zeichneDreieck(100, 200, 70, 20);
  ................................ .
```

Aufgaben

1. Implementiere die Methode *drawCircle(xm, ym, r: zahl)*, welche als Parameter die Koordinaten des Kreismittelpunktes und den Radius besitzt!

2. Implementiere die Methode *drawRectangle(x, y, b, h: zahl)*, welche als Parameter die Koordinaten des linken, unteren Eckpunktes, sowie die Breite und Höhe besitzt!

3. Implementiere die Methode *drawHouse(x, y, b, h: zahl)*, welche als Parameter den linken unteren Eckpunkt, die Breite und die Gesamthöhe besitzt!

4. Implementiere die Methode *zeichneDreieck(x, y, richtung, a, c, betta: zahl)*, welche als Parameter die Koordinaten des Eckpunktes A, die Richtung der Grundseite c, die Längen der Seiten a und c und den eingeschlossenen Winkel β enthält).

Die Klasse Maus

Eine Maus realisiert die Mauseingabe des verwendeten Computers. Diese ist gekennzeichnet durch die aktuelle Position der Maus auf dem Bildschirm des Computers und die Betätigung der Maustaste zu einem bestimmten Zeitpunkt.

Auftrag init
nachher Die Maus ist initialisiert.

Anfrage istGedrueckt : Wahrheitswert
nachher Falls die (bei mehreren Tasten: linke) Maustaste im Moment gedrückt ist, ist "istGedrueckt" wahr, sonst falsch.

Anfrage doppelKlick : Wahrheitswert
nachher Falls der letzte Klick ein Doppelklick war, ist "doppelKlick" wahr, sonst falsch.

Anfrage hPosition : Zahl
nachher Diese Anfrage liefert die gegenwärtige horizontale Koordinate der Position der Maus auf dem Bildschirm, unabhängig davon, ob die Maus gedrückt wurde.

Anfrage vPosition : Zahl
nachher Diese Anfrage liefert die gegenwärtige vertikale Koordinate der Position der Maus auf dem Bildschirm, unabhängig davon, ob die Maus gedrückt wurde.

Auftrag gibFrei
nachher Die Maus steht nicht mehr zur Verfügung.

Maus
- Position - Zustand
+ ! init + ? istGedrueckt + ? doppelKlick + ? hPosition + ? vPosition + ! gibFrei

Bedingungen

```
IF meineMaus.istGedrueckt THEN meinStift.hoch;
```

Sollen, falls die Bedingung erfüllt ist, mehrere Befehle ausgeführt werden, so müssen diese mit *BEGIN* und *END* eingeschlossen werden:

```
IF meineMaus.hPosition > 400 THEN BEGIN
  meinStift.bewegeBis(meineMaus.hPosition, meineMaus.vPosition);
  meinStift.zeichneKreis(30)
END;
```

IF-Abfragen können geschachtelt werden:

```
IF meineMaus.doppelKlick THEN BEGIN
  meinStift.bewegeBis(meineMaus.hPosition, meineMaus.vPosition);
  IF meineMaus.vPosition > 200 THEN BEGIN
    meinStift.zeichneKreis(30);
    meinStift.zeichneRechteck(40, 40)
  END
END;
```

Man kann auch eine sog. zweiseitige Bedingung angeben:

```
IF meineMaus.vPosition >= 500 THEN BEGIN
  meinStift.hoch;
  meinStift.bewegeBis(meineMaus.hPosition, meineMaus.vPosition);
  meinStift.zeichneKreis(10);
END
ELSE meinStift.zeichneRechteck(20,50);
```

Wichtig: Vor *ELSE* darf kein Semikolon stehen!

```
IF meineMaus.vPosition >= 50 THEN meinStift.hoch
ELSE BEGIN
       meinStift.runter;
       meinStift.bewegeBis(70,80);
       meinStift.hoch
     END;
```

Bei geschachtelten Abfragen bezieht sich die ELSE-Anweisung immer auf die letzte freie IF-Abfrage. Vergleiche die beiden folgenden Anweisungsblöcke! Was bewirken sie jeweils bei folgenden Mauskoordinaten:
a) (100; 100); b) (100; 500) c) (500; 100) d) (500; 500)

Block 1:

```
IF meineMaus.hPosition >= 400 THEN
   IF meineMaus.vPosition > 400 THEN
      meinStift.zeichneKreis(20)
   ELSE meinStift.zeichneRechteck(30,30);
meinStift.bewegeBis(0,0);
```

Block 2:

```
IF meineMaus.hPosition >= 400 THEN
   BEGIN
     IF meineMaus.vPosition > 400 THEN
       meinStift.zeichneKreis(20)
   END
ELSE meinStift.zeichneRechteck(30,30);
meinStift.bewegeBis(0,0);
```

Logische Operatoren NOT, AND, OR

```
IF NOT meineMaus.istGedrueckt THEN
 meinStift.bewegeBis(meineMaus.hPosition, meineMaus.vPosition)
```

```
IF NOT (meineMaus.hPosition > 300 ) THEN
 meinStift.bewegeBis(meineMaus.hPosition, meineMaus.vPosition)
```

Beachte: zwischen NOT und THEN muss ein Wahrheitswert stehen. *meineMaus.hPosition* ist alleine kein Wahrheitswert. Erst die Auswertung des obigen Klammerausdrucks liefert einen Wahrheitswert.

```
IF (meineMaus.hPosition > 300 ) AND
meineMaus.istGedrueckt THEN meinStift.bewegeBis(20,30)
```

Der Stift wird im obigen Beispiel nur bewegt, wenn beide Bedingungen erfüllt sind.

```
IF (meineMaus.hPosition > 300 ) OR
meineMaus.istGedrueckt THEN meinStift.bewegeBis(20,30)
```

Der Stift wird im obigen Beispiel nur bewegt, wenn eine oder beide Bedingungen erfüllt sind.

Schleifen

```
WHILE meineMaus.istGedrueckt DO
 meinStift.bewegeBis(meineMaus.hPosition,meineMaus.vPosition);
meinStift.zeichneKreis(10);
.....................
```

Bei einer WHILE-Schleife wird zuerst die **Eingang**sbedingung geprüft. Falls diese Bedingung erfüllt sein sollte, wird der Anweisungsteil einmal ausgeführt. Danach wird wieder die Eingangsbedingung überprüft usw.
Ist die Eingangsbedingung nicht erfüllt, wird die WHILE-Schleife verlassen und es wird zum nächsten Befehl übergegangen. Es kann also durchaus sein, dass der Anweisungsteil einer WHILE-Schleife überhaupt nicht ausgeführt wird.

Sollte der Anweisungsteil bei einer WHILE-Schleife aus mehr als einem Befehl bestehen, so müssen diese Befehle mit BEGIN und END eingeschlossen werden:

```
WHILE meineMaus.istGedrueckt DO BEGIN
 meinStift.bewegeBis(meineMaus.hPosition,meineMaus.vPosition);
 meinStift.zeichneKreis(1)
END;
.....................
```

```
i := 1;
WHILE i <= 5 DO BEGIN
  meinStift.bewegeUm(30);
  meinStift.dreheUm(72);
  i := i+1
END
```

Bei der REPEAT-Schleife wird der Anweisungsteil immer wieder solange ausgeführt, bis die **Ausgang**sbedingung erfüllt ist. Der Anweisungsteil einer REPEAT-Schleife wird also mindestens einmal ausgeführt.

```
REPEAT
  meinStift.bewegeUm(5);
  meinStift.zeichneKreis(1);
UNTIL (meinStift.hPosition > 100);
```

Beachte: ***Der Anweisungsteil einer Repeat-Schleife steht nicht zwischen BEGIN und END.***

Oft wird ein Programm solange ausgeführt, bis ein Doppelklick der Maus erfolgt. Diese Technik werden wir in Zukunft oft benutzen, um ein Programm solange auszuführen bis es mit einem Doppelklick der Maus beendet wird.

```
REPEAT
  meinStift.bewegeBis(meineMaus.hPosition, meineMaus.vPosition);
  meinStift.zeichneRechteck(1,1);
UNTIL meineMaus.Doppelklick;
```

Freihandzeichnen

1. Eine Linie auf dem Bildschirm folgt den Bewegungen der Maus. Das Programm wird durch einen Doppelklick der Maus beendet. Kannst du dein Programm anschließend so verbessern, dass zu Beginn keine Linie von der oberen linken Ecke bis zur ersten Mausposition gezogen wird?

2. Mit einem Druck auf die Maustaste wird ein Punkt an der Mausposition gezeichnet. Das Programm wird durch einen Doppelklick der Maus beendet. Drücke die Maustaste und bewege die Maus anschließend schnell eine kurze Strecke über den Bildschirm. Sind an der Mausspur die Beschleunigungen dieser Bewegung zu erkennen?

3. Mit einem sog. Klick (= Drücken und wieder loslassen) auf die Maustaste wird ein Punkt an der Mausposition gezeichnet. Das Programm wird durch einen Mausdoppelklick beendet.

4. (Schreiben auf dem Bildschirm)
Solange die Maustaste gedrückt ist, folgt eine Linie auf dem Bildschirm den Bewegungen der Maus. Das Programm wird durch einen Doppelklick beendet.

5. Wenn die Maustaste gedrückt wird, wird nur ein Punkt an der Mausposition gezeichnet. Wenn die Maustaste losgelassen wird, wird eine gerade Linie von diesem Punkt bis zur neuen Mausposition gezeichnet. Das Programm wird durch einen Doppelklick beendet.

6. Implementiere die Methode *zeichneRegulaeresVieleck(x,y,k:zahl; n:INTEGER)* !
Dabei bedeuten x und y die Koordinaten eines Eckpunktes, k die Kantenlänge und n die Anzahl der Eckpunkte.

Lösungen

```
Aufgabe 1
procedure TForm1.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinStift := Stift.init;

  meinStift.bewegeBis(meineMaus.hPosition,
                               meineMaus.vPosition)
  meinStift.runter;
  REPEAT
    meinStift.bewegeBis(meineMaus.hPosition,
                               meineMaus.vPosition)
  UNTIL meineMaus.doppelKlick;

  meinStift.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei;
end;
```

```
Aufgabe 2
procedure TForm1.FormCreate(Sender: TObject);
begin
  // 3 Initialisierungen
  REPEAT
    meinStift.bewegeBis(meineMaus.hPosition,
                              meineMaus.vPosition);
    IF meineMaus.istGedrueckt THEN
                         meinStift.zeichneKreis(1);
  UNTIL meineMaus.doppelKlick;
  // 3 Freigaben
end;
```

Aufgabe 3

```
………
  REPEAT
    IF meineMaus.istGedrueckt THEN BEGIN
      meinStift.bewegeBis(meineMaus.hPosition,
                                   meineMaus.vPosition);
      meinStift.zeichneKreis(1)
    END;
    WHILE meineMaus.istGedrueckt DO;
  UNTIL meineMaus.doppelKlick;
…….
```

Aufgabe 4

Lösungsversion a)………

```
  REPEAT
    meinStift.runter;
    WHILE meineMaus.istGedrueckt DO
       meinStift.bewegeBis(meineMaus.hPosition,
                                   meineMaus.vPosition);
    meinStift.hoch;
    WHILE NOT meineMaus.istGedrueckt DO
       meinStift.bewegeBis(meineMaus.hPosition,
                                   meineMaus.vPosition);
  UNTIL meineMaus.doppelKlick;
……..
```

Lösungsversion b)

```
……………
  REPEAT
    IF meineMaus.istGedrueckt THEN meinStift.runter
    ELSE meinStift.hoch;
    meinStift.bewegeBis(meineMaus.hPosition,
                                   meineMaus.vPosition);
  UNTIL meineMaus.doppelKlick;
……..
```

Aufgabe 5

```
………………….
  REPEAT
    If meineMaus.istGedrueckt THEN BEGIN
      meinStift.bewegeBis(meineMaus.hPosition,
                                    meineMaus.vPosition);
      meinStift.zeichneKreis(1);
      WHILE meineMaus.istGedrueckt DO;
      meinStift.runter;
      meinStift.bewegeBis(meineMaus.hPosition,
                                    meineMaus.vPosition);
      meinStift.hoch;
    END
  UNTIL meineMaus.doppelKlick;
……………
```

Die Klasse Tastatur

Eine Tastatur realisiert die Tastatureingabe des verwendeten Computers. Sie speichert die eingegebenen Tastaturzeichen in der Reihenfolge ihrer Eingabe (in einem sog. Tastatur-puffer). Für einige Tastatureingaben stehen bereits Konstanten zur Verfügung, z.B.: PFEILLINKS, PFEILRECHTS, PFEILUNTEN, PFEILOBEN, BILDAUF, BILDAB etc.

Auftrag	init
nachher	Die Tastatur ist initialisiert und enthält keine Zeichen.
Anfrage	wurdeGedrueckt : Wahrheitswert
nachher	Falls eine Taste gedrückt wurde, die Tastatur also (mindestens) ein Zeichen enthält, ist "wurdeGedrueckt" wahr, sonst falsch. Hinweis: der Zustand *(wurdeGedrueckt = TRUE)* ändert sich erst wieder, wenn der Auftrag *weiter* (entsprechend oft) erteilt wird.
Anfrage	zeichen : Zeichen
vorher	Die Tastatur enthält ein Zeichen.
nachher	Diese Anfrage liefert eine Kopie des zuerst eingegebenen Zeichens im Tastaturpuffer. Das Zeichen im Tastaturpuffer selbst wird dabei nicht gelöscht.
Auftrag	weiter
vorher	Die Tastatur enthält ein Zeichen.
nachher	Das zuerst eingegebene Zeichen ist danach nicht mehr in der Tastatur (bzw. in dem Tastaturpuffer) gespeichert.
Auftrag	gibFrei
nachher	Die Tastatur steht nicht mehr zur Verfügung.

Tastatur
- Zeichen
+ ! init + ? wurdeGedrueckt + ? zeichen + ! weiter + ! gibFrei

```
Anwendungsbeispiel:

var
  Main: TMain;
  meineMaus: Maus;
  meinSchirm: Bildschirm;
  meinStift: Stift;
  meineTastatur: Tastatur;

implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
  meineMaus := Maus.init;
  meinSchirm := Bildschirm.init;
  meinStift:= Stift.init;
  meineTastatur:= Tastatur.init;

  Repeat
    IF meineTastatur.wurdeGedrueckt THEN BEGIN
      IF meineTastatur.zeichen = 'a' THEN
                       meinStift.schreibeText('Anton');
      IF meineTastatur.zeichen = 'b' THEN
                    meinStift.schreibeText('Brigitte');
      IF meineTastatur.zeichen = PFEILLINKS THEN
                  meinStift.schreibeText('Pfeillinks');
      meineTastatur.weiter
    END;
  Until meineMaus.doppelklick;

  meineMaus.gibFrei;
........................// Freigabe aller anderen Objekte
end;

end.
```

Aufgaben

1. Der Rechner wartet auf Tastatureingaben. Es werden nur die 4 Pfeiltasten akzeptiert. Wird die Pfeilrechts-Taste gedrückt, so erscheint eine 50 Pixel lange Linie nach rechts. Analog wird bei den anderen Pfeiltasten reagiert. Das ganze wird solange durchgeführt, bis ein Mausdoppelklick erfolgt.

2. Wie Aufgabe 1. Zusätzlich werden die Ziffer 1 und die Buchstaben h, r und v akzeptiert. Bei Eingabe der Ziffer 1 dreht sich der Stift um 10 Grad nach links, die Buchstaben h und r sorgen dafür, dass der Stift hoch gehoben bzw. runter gelassen wird. Der Buchstabe v bewirkt ein Vorwärtsgehen des Stiftes um 50 Pixel. Letzteres ist sinnvoll, falls der Winkel des Stiftes vorher geändert wurde.

3. Solange die Maus nicht gedrückt wurde, passiert gar nichts. Nach dem lang erwarteten Mausklick geschieht folgendes:

> Der Stift geht zum Mausort. Solange die Tastatur nicht gedrückt wird, passiert gar nichts. Nach der erwarteten Tastatureingabe wird am Mausort ein entsprechendes Wort geschrieben.

Das ganze wird solange durchgeführt, bis ein Mausdoppelklick erfolgt.

4. Solange der Mausknopf gedrückt ist, folgt eine Linie auf dem Bildschirm den Bewegungen der Maus. Zusätzlich ist es möglich, durch Drücken einer Taste auf Radieren zu wechseln. Das Programm wird durch einen Doppelklick beendet. (Bemerkung: Wenn man zusätzlich abfragt, welche Taste gedrückt wurde, so kann man zwischen Normal- und Radiermodus hin und her wechseln).
Wahrscheinlich wird dein Programm unterschiedlich reagieren, je nachdem, ob bei gedrückter Maustaste die Tastatur betätigt wurde oder nicht. Kannst du das erklären?

5. Zu Beginn des Programms kann man viele Zeichen in den Tastaturpuffer schreiben, ohne dass diese auf dem Bildschirm sichtbar werden. Erst nach einem Mausklick erscheinen die angegebenen Zeichen alle in der Bildschirmmitte. Danach wartet das Programm solange, bis es mit einem Mausdoppelklick beendet wird.

Lösungen

Aufgabe 4

```
…………..
meineTastatur:= Tastatur.init;
  REPEAT
    meinStift.runter;
    { Wahrscheinlich ist zu Beginn des Programms der Mausknopf noch nicht
      gedrückt.}
    WHILE meineMaus.istGedrueckt DO
        meinStift.bewegeBis(meineMaus.hPosition,
                                        meineMaus.vPosition);
    IF meineTastatur.wurdeGedrueckt THEN
                                           meinStift.radiere;
    meinStift.hoch;
    WHILE NOT meineMaus.istGedrueckt DO
        meinStift.bewegeBis(meineMaus.hPosition,
                                        meineMaus.vPosition);
  UNTIL meineMaus.doppelKlick;
  meineTastatur. gibFrei;
……………………
```

Aufgabe 5

```
…………..
  meinStift.bewegeBis(meinSchirm.breite/2,
                                         meinSchirm.hoehe/2);
  While Not meineMaus.istGedrueckt DO;
  WHILE meineTastatur.wurdeGedrueckt DO BEGIN
      meinStift.schreibeText(meineTastatur.zeichen);
      meineTastatur.weiter
  END;
  Repeat Until meineMaus.doppelklick;
  meineTastatur.gibFrei;
  // Freigabe aller anderen Objekte
```

Die Klasse „Buntstift“

Oberklasse: *Stift*

Der Buntstift übernimmt die Attribute der Klasse Stift. Allerdings besitzt er darüber hinausgehende Attribute, die mithilfe von geeigneten Methoden einzeln gesetzt werden können:

Farbe:	SCHWARZ, WEISS, ROT, GRUEN, BLAU, GELB, etc.
Schriftart:	STANDARDSCHRIFT, sonstige Bezeichner vom System abhängig.
Schriftstil:	STANDARDSTIL, FETT, KURSIV, UNTERSTRICHEN, DURCHGESTRICHEN etc. Schriftstile können durch Addition kombiniert werden, z.B. FETT + UNTERSTRICHEN
Schriftgroesse:	10 bzw. andere positive ganze Zahl.
Linienbreite:	1 bzw. andere positive ganze Zahl.
Füllmuster:	DURCHSICHTIG, GEFUELLT und weitere vom System abhängige Musterkonstanten

Die Standardeinstellungen sind hier jeweils zuerst genannt (z.T. allerdings von der aktuellen Hardwareplattform abhängig).

Hinweis: in der folgenden Dokumentationl werden insbesondere auch diejenigen Methoden aufgelistet, die zwar namensgleich mit den entsprechenden Methoden der Oberklasse ***Stift*** sind, die aber teilweise anders programmiert werden mussten. Im anschließenden UML-Diagramm werden diese namensgleichen Methoden nicht noch einmal in der Unterklasse ***Buntstift*** aufgeführt.

Auftrag init
nachher Der Buntstift ist als Stift initialisiert und mit den erweiterten Standardeinstellungen versehen.

Auftrag bewegeUm (pDistanz : Zahl)
nachher Der Buntstift wurde von seiner aktuellen Position in die aktuelle Richtung unter Verwendung der aktuellen Farbe und Linienbreite bewegt. Die Distanz gibt die Länge der zurückgelegten Strecke an.

Auftrag bewegeBis (ph, pv : Zahl)
nachher Der Buntstift wurde unabhängig von seiner vorherigen Position auf die durch die Parameter angegebene Position unter Verwendung der aktuellen Farbe und Linienbreite bewegt. Die Winkelstellung des Buntstiftes hat sich nicht geändert.

Auftrag setzeFarbe (pFarbe : GanzeZahl)
nachher Die angegebene Farbe ist die aktuelle Farbe des Buntstifts.

Auftrag setzeLinienBreite (pBreite : GanzeZahl)
nachher Die angegebene Breite ist die aktuelle Breite des Buntstifts.

Auftrag setzeFuellMuster (pMuster : GanzeZahl)
nachher Das angegebene Muster ist das aktuelle Muster des Buntstifts.

Auftrag setzeSchriftArt (pArt : Zeichenkette)
nachher Die angegebene Art ist die aktuelle Schriftart des Buntstifts.

Auftrag setzeSchriftStil (pStil : GanzeZahl)
nachher Der angegebene Stil ist der aktuelle Schriftstil des Buntstifts.

Auftrag setzeSchriftGroesse (pGroesse : GanzeZahl)
nachher Die angegebene Groesse ist die aktuelle Schriftgröße des Buntstifts.

Anfrage textBreite (pText : Zeichenkette) : Zahl
nachher Diese Anfrage ermittelt die Breite des angegebenen Textes unter Berücksichtigung der aktuellen Schrifteigenschaften des Stiftes.

Anfrage zahlBreite (pZahl : Zahl) : Zahl
nachher Diese Anfrage ermittelt die Breite der angegebenen Zahl unter Berücksichtigung der aktuellen Schrifteigenschaften des Stiftes.

Auftrag schreibeText (pText : Zeichenkette)
nachher Der Buntstift hat den angegebenen Text auf die Zeichenebene unter Verwendung seines aktuellen Zeichenmodus, der aktuellen Schriftgröße, Schriftart und des aktuellen Schriftstils unabhängig vom Zustand geschrieben. Die aktuelle Stiftposition war die linke obere Ecke des Textes. Die neue Stiftposition ist die rechte obere Ecke des Textes.

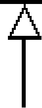
Auftrag schreibeZahl (pZahl : Zahl)
nachher Der Buntstift hat die angegebene Zahl auf die Zeichenebene unter Verwendung seines aktuellen Zeichenmodus, der aktuellen Schriftgröße, Schriftart und des aktuellen Schriftstils unabhängig vom Zustand geschrieben. Die aktuelle Stiftposition war die linke obere Ecke der Zahl. Die neue Stiftposition ist die rechte obere Ecke der Zahl.

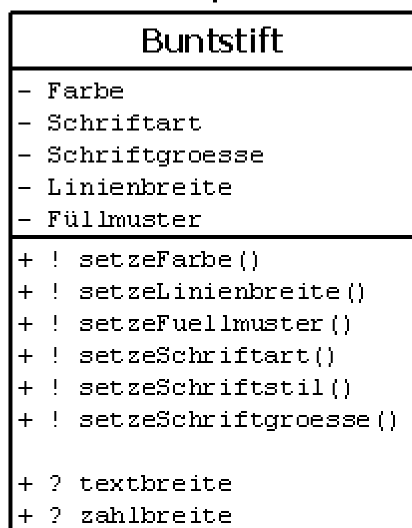
Auftrag zeichneRechteck (pBreite, pHoehe : Zahl)
nachher Der Buntstift hat in der aktuellen Farbe und Linienbreite ein achsenparalleles Rechteck mit der aktuellen Position als linker oberer Ecke und der angegebenen Breite und Höhe gezeichnet. Das Rechteck ist mit dem aktuellen Muster gefüllt. Die Position und die Richtung des Buntstiftes sind unverändert.

Auftrag zeichneKreis (pRadius : Zahl)
nachher Der Buntstift hat in der aktuellen Farbe und Linienbreite einen Kreis mit der aktuellen Position als Mittelpunkt und dem angegebenen Radius gezeichnet. Der Kreis ist mit dem aktuellen Muster gefüllt. Die Position und die Richtung des Buntstiftes sind unverändert.

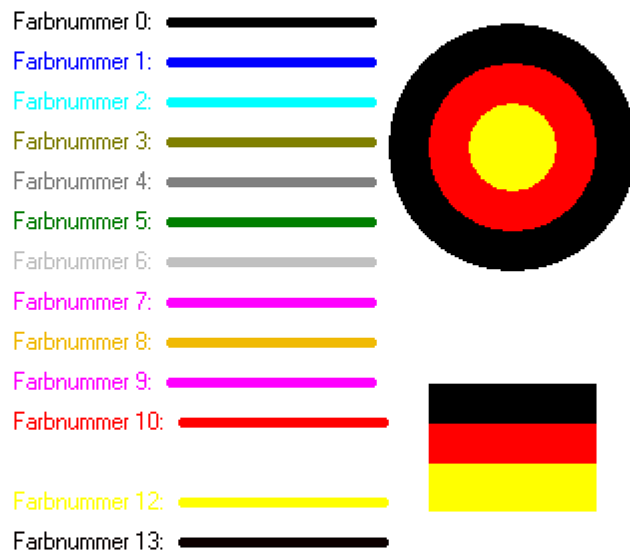
Auftrag gibFrei
nachher Der Buntstift steht nicht mehr zur Verfügung.

Stift
Position
Richtung
Zustand
Modus
+ ! init
+ ! bewegeUm()
+ ! bewegeBis()
+ ! dreheUm()
+ ! dreheBis()
+ ! runter
+ ! hoch
+ ! schreibeZahl()
+ ! schreibeText()
+ ! normal
+ ! wechsle
+ ! radiere
+ ! zeichneRechteck()
+ ! zeichneKreis()
+ ? hPosition
+ ? vPosition
+ ? winkel
+ ! gibFrei




Buntstift
- Farbe
- Schriftart
- Schriftgroesse
- Linienbreite
- Füllmuster
+ ! setzeFarbe()
+ ! setzeLinienbreite()
+ ! setzeFuellmuster()
+ ! setzeSchriftart()
+ ! setzeSchriftstil()
+ ! setzeSchriftgroesse()
+ ? textbreite
+ ? zahlbreite

Programmiere das folgende Bild!



<u>Informatik</u>

ist

unwahrscheinlich toll

Lösung

```
procedure TMain.flagge;
BEGIN
  meinBuntstift.setzeFuellMuster(GEFUELLT);

  meinBuntstift.bewegeBis(220,200);
  meinBuntstift.setzeFarbe(SCHWARZ);
  meinBuntstift.zeichneRechteck(80,20);

  meinBuntstift.bewegeBis(220,220);
  meinBuntstift.setzeFarbe(ROT);
  meinBuntstift.zeichneRechteck(80,20);

  meinBuntstift.bewegeBis(220,240);
  meinBuntstift.setzeFarbe(GELB);
  meinBuntstift.zeichneRechteck(80,20);
END;


procedure TMain.Zielscheibe;
BEGIN
  meinBuntstift.setzeFuellMuster(GEFUELLT);

  meinBuntstift.bewegeBis(260,80);
  meinBuntstift.setzeFarbe(SCHWARZ);
  meinBuntstift.zeichneKreis(60);

  meinBuntstift.bewegeBis(260,80);
  meinBuntstift.setzeFarbe(ROT);
  meinBuntstift.zeichneKreis(40);

  meinBuntstift.bewegeBis(260,80);
  meinBuntstift.setzeFarbe(GELB);
  meinBuntstift.zeichneKreis(20);
END;
```

```
procedure TMain.farbNummern;
VAR x, y: zahl;
    f: GanzeZahl;
BEGIN
  x := 10;
  y := 10;
  f := 0;
  meinBuntstift.setzeLinienBreite(5);
  REPEAT
    meinBuntstift.bewegeBis(x,y);
    meinBuntstift.setzeFarbe(f);
    meinBuntstift.schreibeText('Farbnummer ');
    meinBuntstift.schreibeZahl(f);
    meinBuntstift.schreibeText(':    ');
    meinBuntstift.bewegeBis(meinBuntstift.hPosition,
                          meinBuntstift.vPosition+7);
    meinBuntstift.runter;
    meinBuntstift.bewegeUm(100);
    meinBuntstift.hoch;

    y := y+25;
    f := f+1
  UNTIL f > 13;

END;


procedure TMain.FormCreate(Sender: TObject);
VAR b: zahl;
begin
  meinSchirm := Bildschirm.init;
  meinBuntstift := Buntstift.init;

  farbNummern;
  flagge;
  Zielscheibe;

  meinBuntstift.setzeSchriftArt('Arial');
```

```
  meinBuntstift.setzeSchriftGroesse(26);
  meinBuntstift.setzeFarbe(0);
  meinBuntstift.bewegeBis(400, 80);
  meinBuntstift.setzeSchriftStil(FETT+UNTERSTRICHEN);
  meinBuntstift.schreibeText('Informatik');
  b := meinBuntstift.textbreite('Informatik');
  meinBuntstift.bewegeBis(400, 80+35);
  meinBuntstift.runter;
  meinBuntstift.bewegeUm(b);
  meinBuntstift.hoch;

  meinBuntstift.setzeSchriftArt('Courier');
  meinBuntstift.setzeSchriftGroesse(12);
  meinBuntstift.bewegeBis(430, 150);
  meinBuntstift.setzeSchriftStil(Standardstil);
  meinBuntstift.schreibeText('ist');
  meinBuntstift.hoch;


  meinBuntstift.setzeSchriftArt('Times New Roman');
  meinBuntstift.setzeSchriftGroesse(20);
  meinBuntstift.setzeSchriftStil(KURSIV);
  meinBuntstift.bewegeBis(370, 200);
  meinBuntstift.schreibeText('unwahrscheinlich toll');


  meinSchirm.gibFrei;
  meinBuntstift.gibFrei
end;

end.
```

Aufgaben

1. Es soll ein Haus gezeichnet werden. Die Außenmauern sind breiter als z.B. die Fensterrahmen. Fenster und Tür haben natürlich unterschiedliche Farben. Schreibe ein passendes Programm!

2. Wie Aufgabe 1. Schreibe diesmal eine Prozedur mit Parametern, z.B. *procedure Haus(x,y,breite,hoehe: zahl; f:GanzeZahl)*, so dass man sehr einfach unterschiedlich große Häuser an unterschiedlichen Orten mit unterschiedlicher Grundfarbe erzeugen kann.

3. Wird der Mausknopf gedrückt, so wird nur ein Punkt an der Mausposition gezeichnet. Wenn der Mausknopf losgelassen wird, wird eine gerade Linie von diesem Punkt bis zur neuen Mausposition gezeichnet. Das Programm wird durch einen Doppelklick beendet. Zusätzlich ist es möglich, durch Drücken einer beliebigen Taste die Zeichenfarbe auf rot zu wechseln.

4. Wie Aufgabe 3, aber der Farbwechsel geschieht durch Drücken der Taste „r“ (für rot) bzw. „g“ (für gelb).

5. Wie Aufgabe 3. Es ist möglich, durch Drücken einzelner Buchstaben die Zeichenfarbe zu wechseln, einen Kreis oder ein Rechteck zu zeichnen, die Linienbreite einzustellen usw. .

Lösungen

Aufgabe 3

```
.................
  REPEAT
    If meineMaus.istGedrueckt THEN BEGIN
      meinBuntstift.bewegeBis(meineMaus.hPosition,
                                    meineMaus.vPosition);
      meinBuntstift.zeichneKreis(1);
      WHILE meineMaus.istGedrueckt DO;
      meinBuntstift.runter;
      meinBuntstift.bewegeBis(meineMaus.hPosition,
                                    meineMaus.vPosition);
      meinBuntstift.hoch;
    END;
    If meineTastatur.wurdeGedrueckt THEN BEGIN
      IF meineTastatur.zeichen = 'r' THEN
                         meinBuntstift.setzeFarbe(ROT);
      IF meineTastatur.zeichen = 'g' THEN
                        meinBuntstift.setzeFarbe(GELB);
      meineTastatur.weiter
    END
  UNTIL meineMaus.doppelKlick;
...........
```

Aufgabe 4

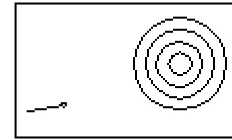
// wie in Aufgabe 3, neu ist:

```
If meineTastatur.wurdeGedrueckt THEN BEGIN
  CASE meineTastatur.zeichen of
   'r': meinBuntstift.setzeFarbe(ROT);
   'g': meinBuntstift.setzeFarbe(GELB);
   'b': meinBuntstift.setzeFarbe(BLAU);
   's': meinBuntstift.setzeFarbe(SCHWARZ);
   'K': BEGIN
          meinBuntstift.hoch;
          meinBuntstift.bewegeBis(meineMaus.hPosition,
```

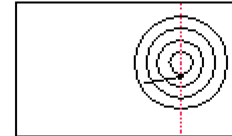
```
                                        meineMaus.vPosition);
                  meinBuntstift.zeichneKreis(10)
            END;
       'R': BEGIN
             meinBuntstift.hoch;
             meinBuntstift.bewegeBis(meineMaus.hPosition,
                                        meineMaus.vPosition);
             meinBuntstift.zeichneRechteck(20, 10)
            END;
       '3': meinBuntstift.setzeLinienBreite(3)
  END; // of CASE
  meineTastatur.weiter
END
……….
```

Projekt Pfeilwurf

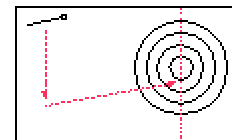
1. **Pfeil und Scheibe**
   Zeichne mit Hilfe des Stifts eine Dartscheibe und einen Pfeil gemäß nebenstehender Abbildung.



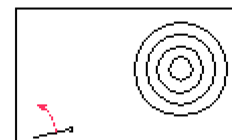
2. **Pfeil fliegt**
   Nun soll der Pfeil zur Dartscheibe fliegen. An einer gedachten Linie bleibt der Pfeil stehen.



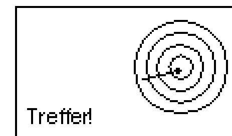
3. **Pfeil fällt**
   Der Pfeil fällt zunächst von oben senkrecht nach unten. Mit einem Mausdruck wird der Pfeil abgefeuert.



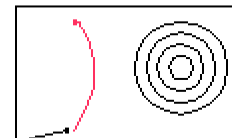
4. **Pfeil dreht**
   Nachdem der Pfeil gefallen ist, kann man noch die Wurfrichtung ändern: Während der Spieler die Maustaste gedrückt hält, soll sich der Pfeil und damit die Wurfrichtung (entgegen dem Uhrzeigersinn) ändern.



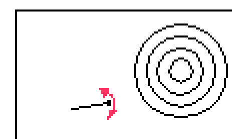
5. **Treffer?**
   Es wird analysiert, ob der Pfeil das Zentrum der Dartscheibe getroffen hat.



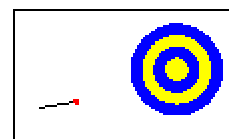
6. **Wieder hoch**
   Wenn der Spieler nicht aufpasst, kann der Pfeil aus dem unteren Bildschirmrand herauslaufen. Das soll nun verbessert werden. Wenn der Pfeil über den unteren Bildschirmrand läuft, soll er wieder nach oben versetzt werden.



7. **Richtungskorrektur**
   Mit einem Mausdruck wird der Pfeil innerhalb des freien Falls abgeschossen. Anschließend kann man noch während des Fluges mit den Tasten „r“ und „l“ die Wurfrichtung etwas nach rechts oder links ändern.



8. **Farbe**
   Das Spiel soll durch Einsatz von Farbe etwas aufgepeppt werden.



Lösungen

Aufgabe 2

```
procedure TMain.PfeilFlieg;
BEGIN
  REPEAT
    meinStift.radiere;
    zeichnePfeil;
    meinStift.bewegeUm(1);
    meinStift.normal;
    zeichnePfeil;
    sleep(10);
  UNTIL meinStift.hPosition > 690
END;


procedure TMain.zeichnePfeil;
BEGIN
  With meinStift DO BEGIN
    runter;
    bewegeUm(10);
    dreheUm(150);
    bewegeUm(3);
    bewegeUm(-3);
    dreheUm(-300);
    bewegeUm(3);
    bewegeUm(-3);
    dreheUm(150);
    bewegeUm(-10);
  END
END;
```

(Fortsetzung Aufgabe 2)

```
procedure TMain.zeichneZielscheibe;
BEGIN
  meinStift.bewegeBis(700, 100);
  meinStift.zeichneKreis(65);
  meinStift.zeichneKreis(40);
  meinStift.zeichneKreis(15);
END;


procedure TMain.FormCreate(Sender: TObject);
begin
  meinSchirm:= Bildschirm.init;
  meinStift:= Stift.init;
  zeichneZielscheibe;
  meinStift.bewegeBis(30,485);
  meinStift.dreheBis(30);
  PfeilFlieg;
  meinStift.gibFrei;
  meinSchirm.gibFrei
end;
```

Entwicklung eigener Klassen

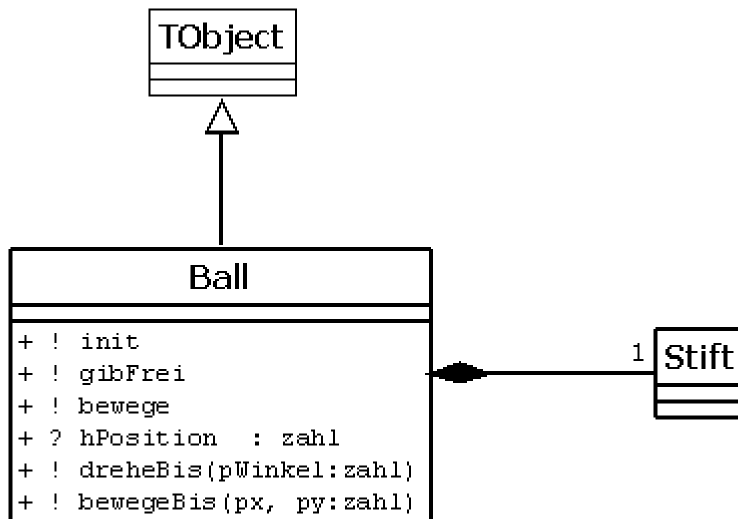
Aufgabe: Ein Ball soll sich horizontal hin und her bewegen.
Lösung:

```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinStift := Stift.init;
  meinStift.bewegeBis(meinBildschirm.breite / 2,
                         meinBildschirm.hoehe / 2);
  REPEAT
    meinStift.radiere;
    meinStift.zeichneKreis(10);
    meinStift.hoch;
    meinStift.bewegeUm(0.1);
    meinStift.runter;
    meinStift.normal;
    meinStift.zeichneKreis(10);
    IF meinStift.hPosition >= meinBildschirm.breite-20 THEN
                                meinStift.dreheBis(180);
    IF meinStift.hPosition <= 20 THEN meinStift.dreheBis(0);
  UNTIL meineMaus.doppelklick;

  meinStift.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei
end;
```

Wenn man nun mehrere, sich bewegende Bälle auf dem Bildschirm haben möchte, so wäre der entsprechende Aufwand im Hauptprogramm ziemlich groß. Man benötigte eventuell mehrere Stifte zum Zeichnen der Bälle. Deshalb ist es günstiger, man erstellt eine eigene Klasse namens Ball. Alle Objekte dieser Klasse sollen sich selber bewegen und deshalb auch selber zeichnen können. Das Hauptprogramm sollte nur das Nötigste regulieren.

Wir werden im Folgenden eine zunächst einfache Klasse namens ***Ball*** erstellen.

Die Klasse Ball



unit mBall; *// Version 1*

{ diese Unit setzt noch voraus, dass ein Bildschirm bereits existiert, auf dem Exemplare der Klasse Ball gezeichnet werden können. }

```
interface
USES mSuM;
Type Ball = class(TObject)
       private
         hatStift: Stift;
       public
         constructor init;
         destructor gibFrei;
         procedure bewege;
         function hPosition: zahl;
         procedure dreheBis(pWinkel: zahl);
         procedure bewegeBis(px, py: zahl);
     end;

implementation

 constructor Ball.init;
   BEGIN
     hatStift := Stift.init;
   END;
```

```
destructor Ball.gibFrei;
   BEGIN
     hatStift.radiere;
     hatStift.zeichneKreis(20);
     hatStift.gibFrei;
     inherited destroy
   END;

 procedure Ball.bewege;
   BEGIN
     hatStift.radiere;
     hatStift.zeichneKreis(20);
     hatStift.hoch;
     hatStift.bewegeUm(0.1);
     hatStift.runter;
     hatStift.normal;
     hatStift.zeichneKreis(20);
   END;

 function Ball.hPosition: zahl;
   BEGIN   RESULT := hatStift.hPosition;   END;

 procedure Ball.dreheBis(pWinkel: zahl);
   BEGIN
     hatStift.dreheBis(pWinkel);
   END;

 procedure Ball.bewegeBis(px, py: zahl);
   BEGIN
     hatStift.radiere;
     hatStift.zeichneKreis(20);
     hatStift.hoch;
     hatStift.bewegeBis(px, py);
     hatStift.runter; hatStift.normal;
     hatStift.zeichneKreis(20);
   END;
end.
```

Das zugehörige Hauptprogramm lautet nun:

```
unit mHaupt;
interface
uses ............Dialogs, mSuM, mBall;

type  TMain = class(TForm)
        procedure FormCreate(Sender: TObject);
      end;

var
  Main: TMain;
  meineMaus: Maus;
  meinBall: Ball;
  meinBildschirm: Bildschirm;

implementation
{$R *.dfm}
procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinBall := Ball.init;
  meinBall.bewegeBis(meinBildschirm.breite / 2,
                          meinBildschirm.hoehe /2);
  REPEAT
    meinBall.bewege;
    IF meinBall.hPosition >= meinBildschirm.breite - 20
                        THEN meinBall.dreheBis(180);
    IF meinBall.hPosition <= 20
                          THEN meinBall.dreheBis(0);
  UNTIL meineMaus.doppelklick;

  meinBall.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei
end;
end.
```

Aufgaben

Alle folgenden Programme sollen mit einem Mausdoppelklick beendet werden können!

1. Schreibe eine Dokumentation der Klasse ***Ball***!
2. Erzeuge zwei Bälle, die sich örtlich übereinander auf dem Bildschirm horizontal hin und her bewegen!
3. Erzeuge zwei Bälle, die sich örtlich übereinander auf dem Bildschirm horizontal hin und her bewegen! Ein Ball soll sich doppelt so schnell bewegen wie der andere.
4. Erzeuge zwei Bälle, die sich auf gleicher Höhe, horizontal und gleich schnell hintereinander her bewegen!
5. Die Klasse Ball soll nun um einige Methoden erweitert werden:
   - Schreibe die Funktion ***vPosition: zahl***
   - Um die Ballgeschwindigkeit einfach bestimmen zu können, ändere die Prozedur ***bewege*** um in ***bewegeUm(v: zahl)***
   - Schreibe eine Prozedur namens ***setzeGroesse(pr: zahl)***
6. Ein einziger Ball soll sich auf dem Bildschirm vertikal hin und her bewegen!
7. Ein einziger Ball soll sich schräg über den Bildschirm bewegen. An allen 4 Bildschirmrändern wird er reflektiert.

Bei mehr als zwei Bällen wird der Aufwand im Hauptprogramm immer größer. Wenn sich die Bälle auch noch gegenseitig abstoßen sollten, würde das Hauptprogramm immer umfangreicher.

Ein wichtiges Ziel in der Informatik ist es, das Hauptprogramm möglichst übersichtlich zu gestalten. Dazu ist es notwendig, möglichst viel *„Intelligenz“* in die untergeordneten Objekte zu stecken. In unserem Beispiel müssten die Bälle selbst erkennen, wann sie in die Nähe eines Bildschirmrandes oder eines anderen Balles kommen, um entsprechend reagieren zu können. Dafür müssten sie aber erst einmal die Größe des Bildschirmes und die anderen Bälle ***kennen***.

Lösungen

1. Dokumentation der Klasse Ball

Auftrag init
nachher Der Ball ist initialisiert und besitzt schon einen Stift. Sein Radius beträgt 20 Pixel. Er ist allerdings noch nicht gezeichnet.

Auftrag gibFrei
nachher Der Ball existiert nicht mehr. Auch auf dem Bildschirm ist er verschwunden.

Auftrag bewege
nachher Der Ball hat sich um 0,1 Pixel bewegt.

Anfrage hPosition
nachher Die x-Koordinate des Ballmittelpunktes wird zurückgegeben.

Auftrag dreheBis(pWinkel: zahl)
nachher Der zum Ball gehörende Stift und damit die zukünftige Bewegungsrichtung des Balles hat sich entsprechend gedreht.

Auftrag bewegeBis(px, py: zahl)
nachher Der Mittelpunkt des Balles hat die Koordinaten (px, py).

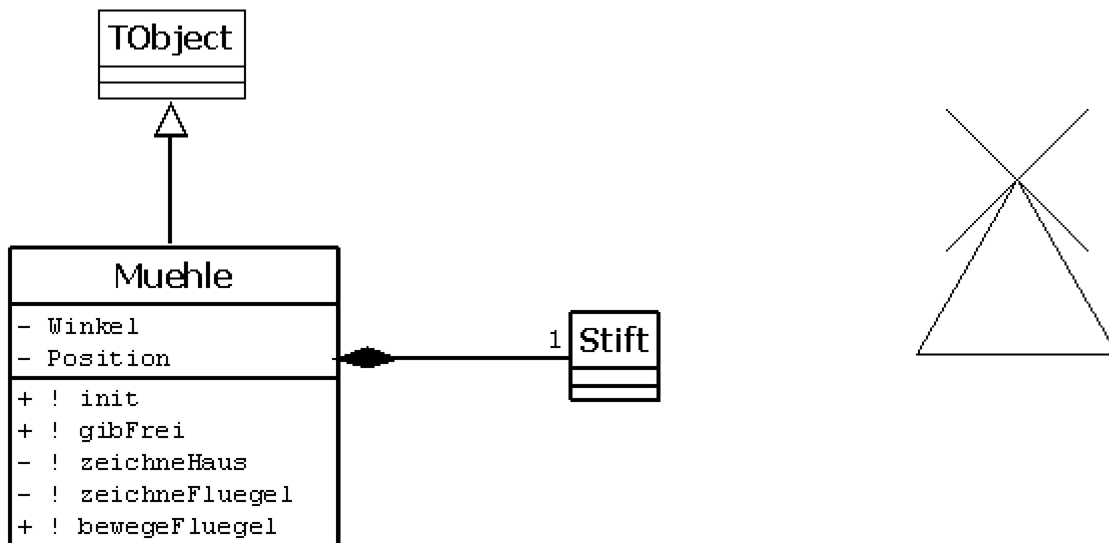
2.
```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinBall1 := Ball.init;
  meinBall1.bewegeBis(20, 20);
  meinBall2 := Ball.init;
  meinBall2.bewegeBis(20, 70);
  REPEAT
    meinBall1.bewege;
    IF meinBall1.hPosition >=meinBildschirm.breite-20
```

```
        THEN meinBall1.dreheBis(180);
    IF meinBall1.hPosition <= 20
        THEN meinBall1.dreheBis(0);
    meinBall2.bewege;
    IF meinBall2.hPosition >=meinBildschirm.breite-20
        THEN meinBall2.dreheBis(180);
    IF meinBall2.hPosition <= 20
        THEN meinBall2.dreheBis(0);
  UNTIL meineMaus.doppelklick;

  meinBall1.gibFrei;
  meinBall2.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei
end;
```

3. Wie Aufgabe 2, nur der Befehl *meinBall2.bewege* wird zweimal direkt hintereinander aufgerufen.

4. Wie Aufgabe 2, nur mit dem Befehl *meinBall2.bewegeBis(70, 20)*

Die Klasse Muehle




```
unit mMuehle;
interface
USES mSuM;
TYPE Muehle = class(TObject)
       private
         zWinkel, zhPosition, zvPosition: zahl;
         hatStift: Stift;
         procedure zeichneHaus;
         procedure zeichneFluegel;
       public
         constructor init;
         procedure bewegeFluegel;
         destructor gibFrei;
     end;

implementation

constructor Muehle.init;
  BEGIN
    zhPosition := 200;
    zvPosition := 150;
    zWinkel := 45;
    hatStift := Stift.init;
    zeichneFluegel;
    zeichneHaus;
  END;
```

```
destructor Muehle.gibFrei;
  BEGIN
    hatStift.radiere;
    self.zeichneHaus;
    self.zeichneFluegel;
    hatStift.gibFrei;
    inherited destroy
  END;

procedure Muehle.zeichneHaus;
  BEGIN
    hatStift.hoch;
    hatStift.bewegeBis(self.zhPosition, self.zvPosition);
    hatStift.runter;
    hatStift.dreheBis(-60);
    hatStift.bewegeUm(100);
    hatStift.dreheBis(180);
    hatStift.bewegeUm(100);
    hatStift.dreheBis(60);
    hatStift.bewegeUm(100);
  END;

procedure Muehle.zeichneFluegel;
  BEGIN
    hatStift.hoch;
    hatStift.bewegeBis(self.zhPosition, self.zvPosition);
    hatStift.dreheBis(self.zWinkel);
    hatStift.bewegeUm(50);
    hatStift.runter;
    hatStift.bewegeUm(-100);
    hatStift.hoch;
    hatStift.bewegeBis(self.zhPosition, self.zvPosition);
    hatStift.dreheUm(90);
    hatStift.bewegeUm(50);
    hatStift.runter;
    hatStift.bewegeUm(-100);
END;
```

```
procedure Muehle.bewegeFluegel;
  BEGIN
    hatStift.radiere;
    zeichneFluegel;
    zWinkel := zWinkel + 0.01;
    hatStift.normal;
    zeichneFluegel;
    zeichneHaus
  END;

end.
```

Das zugehörige Hauptprogramm lautet:

```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm:= Bildschirm.init;
  meineMuehle := Muehle.init;
  meineMaus := Maus.init;

  REPEAT
    meineMuehle.bewegeFluegel
  UNTIL meineMaus.doppelklick;

  meineMaus.gibFrei;
  meineMuehle.gibFrei;
  meinBildschirm.gibFrei
end;
```

Aufgaben

Erstelle eine neue Klasse *Supermuehle*, welche wesentlich mehr kann:

a) Der Konstruktor lautet nun *init(px, py, pg: ganzeZahl)*. Damit lässt sich der Ort und die Größe der Mühle festlegen. Außerdem kann man nun mehrere Mühlen auf dem Bildschirm darstellen.

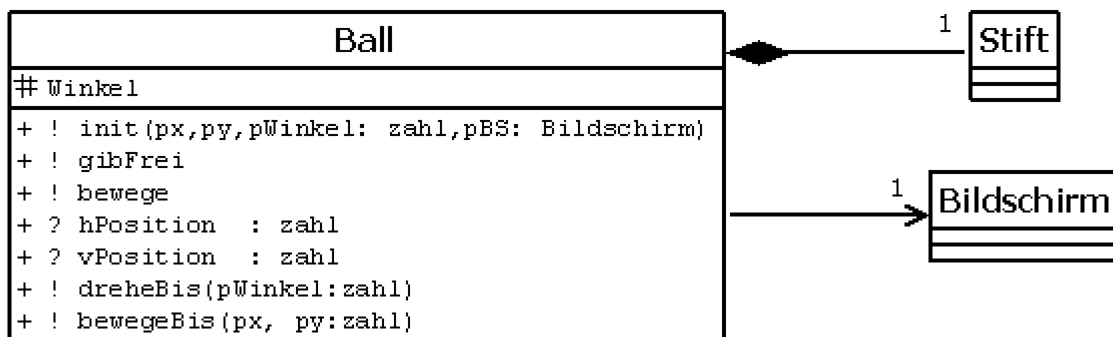
b) Es gibt eine Methode *setzeFarbe(pf: ganzeZahl)*. Damit kann man die Farbe der Mühle festlegen.

c) Es gibt eine Methode *setzeGeschwindigkeit(pv: zahl)*. Damit kann man festlegen, wie schnell sich die Mühlenflügel drehen sollen.

d) Es gibt eine Methode *setzeDrehrichtung(pd: zahl)*. Verschiedene Mühlen können sich nun mit unterschiedlicher Drehrichtung drehen.

e) Erstelle eine Dokumentation deiner Klasse ***Supermuehle***!

Die Kennt-Beziehung



Alle Bälle bewegen sich auf **einem** Bildschirm. Wenn, wie gefordert, ein Ball selbst erkennen soll, wann er sich z.B. dem Bildschirmrand nähert, so muss der Ball den Bildschirm (insbesondere dessen Ausmaße) kennen.

Weil alle Bälle denselben Bildschirm kennen, kann dieser nicht von einem Ball erzeugt werden, sondern jedem Ball muss mitgeteilt werden, dass es diesen einen Bildschirm gibt, auf dem er sich bewegen soll. Der Ball ist also weder für die Initialisierung noch für die Freigabe des Bildschirms zuständig. **Das ist ein wesentlicher Unterschied zur Hat-Beziehung.**

Die Initialisierung und Freigabe des Bildschirms wird in unserem Beispiel vom Hauptprogramm übernommen.

```
unit mBall;    //Version mit Kennt-Beziehung
interface
USES mSuM;
Type Ball = class(TObject)
       protected
         hatStift: Stift;
         kBildschirm: Bildschirm;
         zWinkel: zahl;
       public
         constructor init(px, py, pWinkel: zahl;
                          pBildschirm: Bildschirm);
         destructor gibFrei;
         procedure bewege;
         function hPosition: zahl;
```

```
        function vPosition: zahl;
        procedure dreheBis(pWinkel: zahl);
        procedure bewegeBis(px, py: zahl);
    end;

implementation
constructor Ball.init(px, py, pWinkel: zahl;
                            pBildschirm: Bildschirm);
  BEGIN
    kBildschirm := pBildschirm;
    hatStift := Stift.init;
    bewegeBis(px, py);
    zWinkel := pWinkel;
    hatStift.dreheBis(zWinkel);
   END;

destrucor Ball.gibFrei;
  BEGIN
    hatStift.radiere;
    hatStift.zeichneKreis(20);
    hatStift.gibFrei;
    inherited destroy
  END;

procedure Ball.bewege;
  BEGIN
    hatStift.radiere;
    hatStift.zeichneKreis(20);
    hatStift.bewegeUm(0.1);
    hatStift.normal;
    hatStift.zeichneKreis(20);
    IF self.hPosition >= kBildschirm.breite - 20
                          THEN dreheBis(180-zWinkel);
    IF self.hPosition <= 20 THEN dreheBis(180-zWinkel);
    IF self.vPosition >= kBildschirm.hoehe - 20
                             THEN dreheBis(-zWinkel);
    IF self.vPosition <= 20 THEN dreheBis(-zWinkel);
   END;
```

```
function Ball.hPosition: zahl;
  BEGIN
    RESULT := hatStift.hPosition;
  END;

function Ball.vPosition: zahl;
  BEGIN
    RESULT := hatStift.vPosition;
  END;

procedure Ball.dreheBis(pWinkel: zahl);
  BEGIN
    hatStift.dreheBis(pWinkel);
    zWinkel := pWinkel
  END;

procedure Ball.bewegeBis(px, py: zahl);
  BEGIN
    hatStift.radiere;
    hatStift.zeichneKreis(20);
    hatStift.bewegeBis(px, py);
    hatStift.normal;
    hatStift.zeichneKreis(20);
  END;

end.
```

Das zugehörige Hauptprogramm lautet nun:

....

```
procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinBall1 := Ball.init(300,200,30,MeinBildschirm);
  meinBall2 := Ball.init(200,200,-40,MeinBildschirm);

  REPEAT
    meinBall1.bewege;
    meinBall2.bewege;
  UNTIL meineMaus.doppelklick;

  meinBall1.gibFrei;
  meinBall2.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei
end;
```

....

Aufgaben

1. Gib der Klasse *Ball* eine neue Eigenschaft *Geschwindigkeit v*. Innerhalb der Methode *bewege* könnte man dann etwa die Anweisung *bewegeUm(v)* geben. Die Werte von v sollten ungefähr 0.1 betragen.
Es sollten auch zwei neue Methoden *setV(pv: zahl)* und *getV: zahl* implementiert werden.

Zwei Bälle bewegen sich anfangs in verschiedenen Richtungen mit unterschiedlichen Geschwindigkeiten schräg über den Bildschirm. An den Bildschirmrändern werden sie entsprechend reflektiert.
Zusätzlich kann man mit der Tastatur die Bälle kontrollieren. Bei Eingabe von „+“ oder „-“ erhöht bzw. erniedrigt sich die Geschwindigkeit des ersten Balles. Entsprechende Tasten gibt es für den zweiten Ball.

2. Zwei Bälle sollen sich über den Bildschirm bewegen und sich auch gegen-seitig abstoßen können.
Gib der Klasse *Ball* ein neues Attribut *kBall*, sodass zwei Bälle sich gegenseitig kennenlernen können.
Wenn der erste Ball initialisiert wird, existiert der zweite Ball noch nicht.
Die Klasse *Ball* benötigt also eine weitere Methode *lerneKennen(pB: Ball);*
Wenn sich zwei Bälle zentral stoßen, so tauschen sie ihre Winkel und Geschwindigkeiten aus. Die Klasse *Ball* benötigt also weitere Methoden *setWinkel(pWinkel: zahl)* und *getWinkel: zahl*

3. Es soll das System *Sonne-Erde-Mond* simuliert werden.
a) In der Bildschirmmitte steht eine große Sonne. Um diese herum kreist die Erde.
Mathematischer Hinweis: Ein im Abstand r um den Ursprung kreisender Punkt hat die Koordinaten $(r \cdot \cos(\alpha); r \cdot \sin(\alpha))$. Dabei müssen die Winkelangaben allerdings im Bogenmaß eingegeben werden
b) Um die Erde herum kreist ein noch kleinerer Mond, allerdings entsprechend schneller.
c) Zusätzlich bewegt sich die Sonne langsam horizontal über den Bildschirm.

Lösung der Aufgabe 2

```
unit mBall;
interface
USES mSuM;
Type Ball = class(TObject)
       private
         hatStift: Stift;
         kenntBildschirm: Bildschirm;
         zWinkel, zV: zahl;       // zV = Geschwindigkeit
         kBall: Ball;
       public
         constructor init(px, py, pWinkel: zahl;
                          pBildschirm: Bildschirm);
         destructor gibFrei;
         procedure bewege;
         function hPosition: zahl;
         function vPosition: zahl;
         procedure dreheBis(pWinkel: zahl);
         procedure bewegeBis(px, py: zahl);
         procedure setV(pv: zahl);
         function getV: zahl;
         procedure setWinkel(pWinkel: zahl);
         function getWinkel: zahl;
         procedure lerneKennen(pB: Ball);
     end;

implementation

constructor Ball.init(px, py, pWinkel: zahl;
                          pBildschirm: Bildschirm);
   BEGIN
     kenntBildschirm := pBildschirm;
     hatStift := Stift.init;
     hatStift.hoch;
     bewegeBis(px, py);
     zWinkel := pWinkel;
     hatStift.dreheBis(zWinkel);
```

```
      setV(0.1)
    END;

destructor Ball.gibFrei;
    BEGIN
      hatStift.radiere;
      hatStift.zeichneKreis(20);
      hatStift.gibFrei;
      inherited destroy
    END;

procedure Ball.bewege;
 VAR hilf: zahl;
    BEGIN
      hatStift.radiere;
      hatStift.zeichneKreis(20);
      hatStift.hoch;
      hatStift.bewegeUm(zV);
      hatStift.runter;
      hatStift.normal;
      hatStift.zeichneKreis(20);
      IF self.hPosition >= kenntBildschirm.breite - 20
                           THEN dreheBis(180-zWinkel);
      IF self.hPosition <= 20 THEN dreheBis(180-zWinkel);
      IF self.vPosition >= kenntBildschirm.hoehe - 20
                              THEN dreheBis(-zWinkel);
      IF self.vPosition <= 20 THEN dreheBis(-zWinkel);
      IF SQRT(SQR(self.hPosition - kBall.hPosition) +
          SQR(self.vPosition - kBall.vPosition)) <= 40
         THEN BEGIN
           hilf := self.zWinkel;
           self.setWinkel(kBall.getWinkel);
           kBall.setWinkel(hilf);
           hilf := self.zV;
           self.setV(kBall.getV);
           kBall.setV(hilf)
         END
    END;
```

```
function Ball.hPosition: zahl;
   BEGIN
     RESULT := hatStift.hPosition;
   END;

function Ball.vPosition: zahl;
   BEGIN
     RESULT := hatStift.vPosition;
   END;

procedure Ball.dreheBis(pWinkel: zahl);
   BEGIN
     hatStift.dreheBis(pWinkel);
     zWinkel := pWinkel;
   END;

procedure Ball.bewegeBis(px, py: zahl);
   BEGIN
     hatStift.radiere;
     hatStift.zeichneKreis(20);
     hatStift.hoch;
     hatStift.bewegeBis(px, py);
     hatStift.runter;
     hatStift.normal;
     hatStift.zeichneKreis(20);
   END;

procedure Ball.setV(pv: zahl);
   BEGIN
     zV := pv
   END;

function Ball.getV: zahl;
   BEGIN
     Result := zV
   END;
```

```
procedure Ball.setWinkel(pWinkel: zahl);
   BEGIN
     zWinkel := pWinkel;
     hatStift.dreheBis(pWinkel);
   END;

function Ball.getWinkel: zahl;
   BEGIN
     Result := zWinkel
   END;

procedure Ball.lerneKennen(pB: Ball);
   BEGIN
     kBall := pB
   END;

end.
```

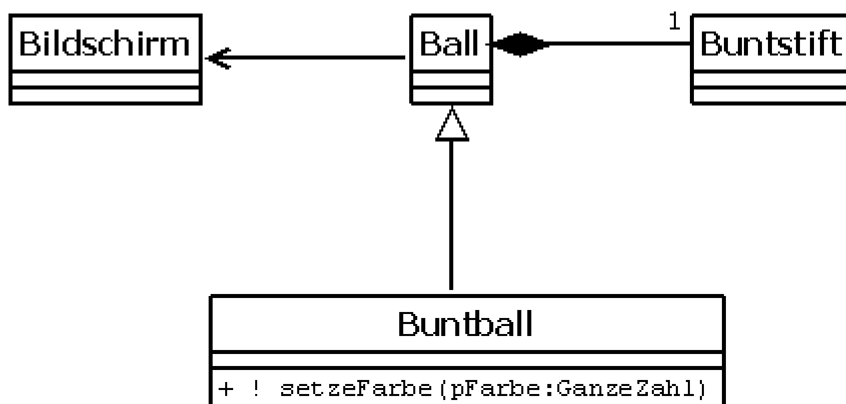
```
unit mHaupt;
interface
uses….. Dialogs, mSuM, mBall;

type  TMain = class(TForm)
        procedure FormCreate(Sender: TObject);
      end;

var  Main: TMain;
     meineMaus: Maus;
     meinBall1, meinBall2: Ball;
     meinBildschirm: Bildschirm;

implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinBall1 := Ball.init(300,200,30,MeinBildschirm);
  meinBall2 := Ball.init(200,200,50,MeinBildschirm);
  meinBall1.setV(0.05);
  meinBall2.setV(0.1);
  meinBall1.lerneKennen(meinBall2);
  meinBall2.lerneKennen(meinBall1);
  REPEAT
    meinBall1.bewege;
    meinBall2.bewege;
  UNTIL meineMaus.doppelklick;

  meinBall1.gibFrei;
  meinBall2.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei
end;

end.
```

Vererbung als Spezialisierung

Für das Folgende wird es wesentlich einfacher, wenn man die Klasse ***Ball*** leicht verändert. Die zuvor ***privaten*** Attribute erhalten nun die Eigenschaft ***protected***, und der benutzte Stift soll nun ein Buntstift sein. Ansonsten ändert sich an der Klasse Ball nichts.

```
Type Ball = class(TObject)
       protected
         hatStift: Buntstift;
         kenntBildschirm: Bildschirm;
         zWinkel: zahl;
       public
         .................. . .
     end;
```

Wir wollen nun Bälle konstruieren, die sich farbig darstellen können. Dafür erweitern wir die Klasse ***Ball*** und erhalten so eine Spezialisierung der ursprünglichen Klasse namens ***Buntball***. Diese Klasse ***Buntball*** ist eine Unterklasse der Klasse ***Ball***.



Das Modul ***mBuntball*** lautet nun:

```
interface
USES mSuM, mBall;
Type Buntball = class(Ball)
       public
         procedure setzeFarbe(pFarbe: GanzeZahl);
     end;

implementation

procedure Buntball.setzeFarbe(pFarbe: GanzeZahl);
  BEGIN
    hatStift.setzeFarbe(pFarbe);
  END;

end.
```

An dem zugehörigen Hauptprogramm ändert sich nur wenig:

```
unit mHaupt;
interface
uses  ...... Dialogs, mSuM, mBall, mBuntball;

.....................
var  Main: TMain;
  meineMaus: Maus;
  meinBall1, meinBall2: Buntball;
  meinBildschirm: Bildschirm;

implementation
{$R *.dfm}
procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm := Bildschirm.init;
  meineMaus := Maus.init;
  meinBall1 := Buntball.init(300,200,0,MeinBildschirm);
  meinBall1.setzeFarbe(ROT);
  meinBall2 := Buntball.init(200,200,90,MeinBildschirm);
  meinBall2.setzeFarbe(BLAU);
```

```
  REPEAT
    meinBall1.bewege;
    meinBall2.bewege;
  UNTIL meineMaus.doppelklick;

  meinBall1.gibFrei;
  meinBall2.gibFrei;
  meineMaus.gibFrei;
  meinBildschirm.gibFrei
end;

end.
```

Im Folgenden sollen Bälle konstruiert werden, die bei ihrer Bewegung eine Spur hinterlassen. Dazu wird für die Unterklasse die Methode ***bewege*** neu geschrieben, welche aber die alte Methode ***bewege*** der Oberklasse ***Ball*** ausführt und nur anschließend noch den Kreismittelpunkt zeichnet:

```
procedure Buntball.bewege;
   BEGIN
     inherited bewege;
     hatStift.zeichneKreis(1);
   END;
```

Übungen zur Klassenbildung

1. a) Konstruiere eine Klasse namens *Uhr*. Diese besitzt im einfachsten Fall nur einen Zeiger, welcher sich im Sekundentakt weiter bewegt.
b) Konstruiere eine Unterklasse namens *Stoppuhr*. Diese läuft nur bei gedrückter Maustaste. Das heißt, eine Stoppuhr muss die Maus kennen.
c) Konstruiere eine Unterklasse *Digitaluhr*. Diese zeigt zusätzlich noch die Zeit in digitaler Form an. Eine Digitaluhr benötigt einen Buntstift, weil nur dieser Zahlen und Texte schreiben kann.
d) Konstruiere eine Unterklasse *Minutenuhr*, welche einen zweiten Zeiger besitzt, der sich genau dann einen Schritt weiterbewegt, wenn der Sekundenzeiger eine Umdrehung hinter sich gebracht hat.
e) Die Minutenuhr soll die Zeit zusätzlich auch in digitaler Form anzeigen.

2. Konstruiere eine Klasse namens ***Lampe***.
a) Eine Lampe – in Form eines schwarzen Kreises – kann auf dem Bildschirm gezeichnet werden. Der Radius wird bei der Erzeugung bestimmt. Sie lässt sich einschalten, dies wird durch das Ausfüllen des Kreises mit gelber Farbe gekennzeichnet, und sie lässt sich wieder ausschalten, dann wird nur der Kreisrand gezeichnet.
b) Es soll eine Unterklasse ***Farblampe*** konstruiert werden.
c) Es soll eine Unterklasse ***Bewegungsmelder*** konstruiert werden. Diese Lampe leuchtet nur dann auf, wenn ihr die Maus zu nahe kommt.

3. Konstruiere eine Klasse namens ***Ampel***.
Eine Ampel besteht aus einem rechteckigenGehäuse und drei farbigen Lampen. Sie besitzt eine Größe, welche schon bei der Erzeugung bestimmt wird, und eine Eigenschaft namens *phase*, welche die vier Werte rot, rot-gelb, grün und gelb annehmen kann. Außerdem gibt es eine Methode namens *weiter*, welche in die jeweils nächste Ampelphase überführt.
a) Schreibe ein Programm, welches eine Ampel darstellt, die im Sekundentakt die Phase wechselt.
b) Stelle mit dem Computer zeichnerisch eine Kreuzung mit vier Ampeln dar! Jeweils zwei gegenüberliegende Ampeln haben die gleiche Phase. Die Phasen sollen natürlich realistisch sein, d.h. wenn die Hauptstraße grün hat, muss die Ampel der Querstraße rot anzeigen.

Lösungen der Aufgaben 1a, b, c

```
unit mHaupt;
interface
uses   ................, mSuM, mUhr, mStoppuhr;

type   TMain = class(TForm)
       end;

var   Main: TMain;
      meineUhr1: Uhr;
      meineStoppuhr: Stoppuhr;
      meineDigitaluhr: Digitaluhr;
      BS: Bildschirm;
      meineMaus: Maus;

implementation
{$R *.dfm}
BEGIN
  BS := Bildschirm.init;
  meineMaus := Maus.init;
  meineUhr1 := Uhr.init(100,100,0);
  meineStoppUhr := Stoppuhr.init(200,100,10);
  meineStoppuhr.lerneMausKennen(meineMaus);
  meineDigitaluhr := Digitaluhr.init(300,100,20)
  REPEAT
    meineUhr1.bewege;
    meineStoppuhr.bewege;
    meineDigitaluhr.bewege;
    sleep(1000);
  UNTIL meineMaus.doppelklick;
  meineDigitaluhr.destroy;
  meineStoppuhr.destroy;
  meineUhr1.destroy;
  meineMaus.gibFrei;
  BS.gibFrei
END.
```

```
unit mUhr;
interface
USES Windows, mSuM;

Type Uhr = class(TObject)
       protected
        hatStift: Stift;
        zSekunde, zxM, zyM: zahl;
        procedure zeichne; virtual;

       public
        constructor init(px,py,ps:zahl); virtual;
        procedure bewege;
        destructor destroy; override;
     end;

implementation

constructor Uhr.init(px,py,ps:zahl);
 BEGIN
   hatStift := Stift.init;
   zxM := px;
   zyM := py;
   zSekunde := ps;
   hatStift.hoch;
   hatStift.bewegeBis(zxM, zyM);
   hatStift.zeichneKreis(30);
   hatStift.runter
 END;


procedure Uhr.zeichne;
 BEGIN
   hatStift.bewegeBis(zxM, zyM);
   hatStift.dreheBis(90-6*zSekunde);
   hatStift.bewegeUm(25);
 END;
```

```
procedure Uhr.bewege;
 BEGIN
   hatStift.radiere;
   zeichne;
   hatStift.normal;
   zeichne;
   zSekunde := zSekunde+1;
   IF zSekunde = 60 THEN zSekunde := 0;
 END;

destructor Uhr.destroy;
 BEGIN
   hatStift.gibFrei;
   inherited destroy
 END;

end.
```

```
unit mStoppuhr;

interface
USES mSuM, mUhr;

Type Stoppuhr = class(Uhr)
       protected
        kMaus: Maus;
        procedure zeichne; override;
       public
        procedure lerneMausKennen(pm: Maus);
     end;

implementation

procedure Stoppuhr.lerneMausKennen(pm: Maus);
 BEGIN
   kMaus := pm
 END;

procedure Stoppuhr.zeichne;
 BEGIN
   IF kMaus.istGedrueckt THEN inherited zeichne
 END;

end.
```

```
unit mDigitaluhr;

interface
USES mUhr, mSuM;
Type Digitaluhr = class(Uhr)
       protected
         hatBuntstift: Buntstift;
         procedure zeichne; override;
       public
         constructor init(px,py,ps:zahl); override;
         destructor destroy; override;
     end;

Implementation
constructor Digitaluhr.init(px,py,ps:zahl);
 BEGIN
   inherited init(px, py, ps);
   hatBuntstift := Buntstift.init;
 END;

procedure Digitaluhr.zeichne;
 BEGIN
   inherited zeichne;
   hatBuntstift.bewegeBis(zxM - 5, zyM + 40);
   IF zSekunde < 10 THEN hatBuntstift.schreibeZahl(0);
   hatBuntstift.schreibeZahl(zSekunde)
 END;

destructor Digitaluhr.destroy;
 BEGIN
   hatBuntstift.gibFrei;
   inherited destroy
 END;

end.
```

Lösung der Aufgabe 2

```
unit mHaupt;
interface

uses  ……, mSuM, mLampe, mBewegungsmelder, mFarblampe;

type  TMain = class(TForm)
      end;

var  Main: TMain;
     meinSchirm: Bildschirm;
     meineMaus: Maus;
     meineLampe: Lampe;
     meineRotlampe: Farblampe;
     meinBM: Bewegungsmelder;

implementation
{$R *.dfm}

BEGIN
  meinSchirm := Bildschirm.init;
  meineMaus := Maus.init;
  meineLampe := Lampe.init(100,100,20);
  meineRotlampe := Farblampe.init(200,100,20);
  meineRotlampe.setzeFarbe(ROT);
  meineLampe.machAn;
  meineRotlampe.machAn;
  sleep(1000);
  meineLampe.machAus;
  meineRotlampe.machAus;
  meinBM := Bewegungsmelder.init(300,100,20);
  meinBM.lerneMauskennen(meineMaus);
  REPEAT
    meinBM.passAuf
  UNTIL meineMaus.doppelklick;

  meinBM.destroy;
  meineRotlampe.destroy;
```

```
  meineLampe.destroy;
  meineMaus.gibFrei;
  meinSchirm.gibFrei

end.



unit mLampe;
interface
USES mSuM;

Type Lampe = class(TObject)
       protected
         hatBuntstift: Buntstift;
         zxM, zyM, zRadius: zahl;
         procedure zeichne(pf: ganzeZahl);
       public
         constructor init(px, py, pR: zahl);
         procedure machAn; virtual;
         procedure machAus;
         destructor destroy; override;
     end;

implementation

constructor Lampe.init(px, py, pR: zahl);
 BEGIN
   zxM := px;
   zyM := py;
   zRadius := pR;
   hatBuntstift := Buntstift.init;
 END;

procedure Lampe.zeichne(pf: ganzeZahl);
 BEGIN
   hatBuntstift.setzeFarbe(pf);
   hatBuntstift.bewegeBis(zxM, zyM);
   hatBuntstift.setzeFuellMuster(GEFUELLT);
```

```
    hatBuntstift.zeichneKreis(zRadius);
    hatBuntstift.setzeFuellMuster(DURCHSICHTIG);
    hatBuntstift.setzeFarbe(SCHWARZ);
    hatBuntstift.zeichneKreis(zRadius);
  END;

procedure Lampe.machAn;
  BEGIN
    zeichne(GELB)
  END;

procedure Lampe.machAus;
  BEGIN
    zeichne(WEISS)
  END;

destructor Lampe.destroy;
  BEGIN
    hatBuntstift.gibFrei;
    inherited destroy
  END;

end.
```

```
unit mFarblampe;

interface
USES mSuM, mLampe;

Type Farblampe = class(Lampe)
       protected
         farbe: ganzeZahl;
       public
         procedure setzeFarbe(pf: ganzeZahl);
         procedure machAn; override;
     end;
```

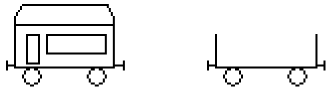
```
implementation

procedure Farblampe.setzeFarbe(pf: ganzeZahl);
 BEGIN
   farbe := pf
 END;

procedure Farblampe.machAn;
 BEGIN
   zeichne(farbe)
 END;

end.
```

```
unit mBewegungsmelder;

interface
USES mSuM, mLampe;

Type Bewegungsmelder = class(Lampe)
       protected
         kMaus: Maus;
       public
         procedure lerneMauskennen(pm: Maus);
         procedure passAuf;
     end;

implementation

procedure Bewegungsmelder.lerneMauskennen(pM: Maus);
 BEGIN
   kMaus := pM
 END;

procedure Bewegungsmelder.passAuf;
```

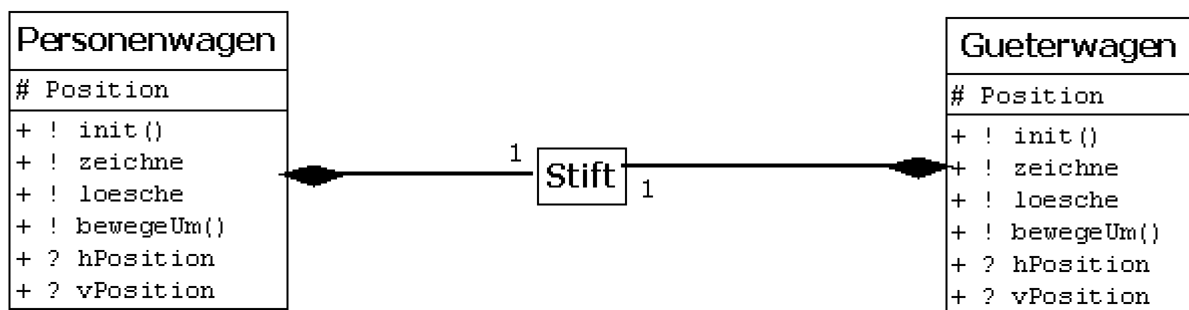
```
 BEGIN
   IF SQRT(SQR(kMaus.hPosition - zxM)
                  + SQR(kMaus.vPosition - zyM)) < 100
   THEN machAn
   ELSE machAus
 END;

end.
```

Abstrakte Klassen als Generalisation

Wir untersuchen nun die zwei Klassen *Gueterwagen* und *Personenwagen* (letztere unterscheidet sich von der ersteren nur durch eine andere Zeichenmethode).



- Ein Wagen wird am linken Puffer ganz links in der Mitte beginnend gezeichnet.
- Die Räder haben einen Radius von 5 Punkten (Pixel = picture element)
- Der Wagen hat eine Gesamtlänge (einschließlich Puffer) von ungefähr 60 Pixel.

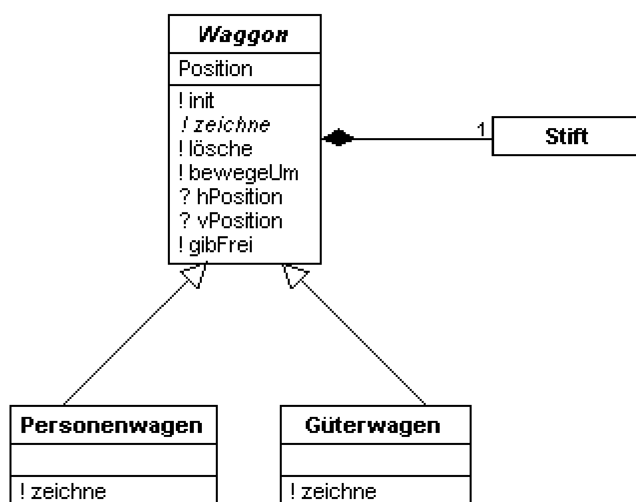


Dokumentation der Klasse "Güterwagen"

Ein Güterwagen ist ein grafisches Objekt, das auf dem Bildschirm dargestellt und bewegt werden kann. Er befindet sich stets auf einer genau definierten Position des Bildschirms, angegeben durch die Koordinaten des Mittelpunktes des linken Puffers.

Auftrag	init (pHPosition, pVPosition : Zahl)
nachher	Der Güterwagen ist initialisiert, d.h. er ist an der angegebenen Position auf dem Bildschirm dargestellt.
Auftrag	zeichne
nachher	Der Güterwagen ist an seiner aktuellen Position auf dem Bildschirm dargestellt.
Auftrag	lösche
nachher	Der Güterwagen ist nicht mehr auf dem Bildschirm dargestellt.
Auftrag	bewegeUm (pDistanz : Zahl)
nachher	Der Güterwagen wurde um die angegebene Distanz nach links bewegt.
Anfrage	hPosition : Zahl
nachher	Diese Anfrage liefert die aktuelle horizontale Position des Güterwagens.
Anfrage	vPosition : Zahl
nachher	Diese Anfrage liefert die aktuelle vertikale Position des Güterwagens.
Auftrag	gibFrei
nachher	Der Güterwagen ist vom Bildschirm gelöscht und steht nicht mehr zur Verfügung.

Offensichtlich lässt sich hier aus den beiden Wagenklassen eine allgemeinere Oberklasse generalisieren:



Dokumentation der Klasse "Waggon"

Die Klasse *Waggon* ist eine sog. abstrakte Klasse. Eine abstrakte Klasse enthält mindestens eine Methode, welche in dieser Klasse nicht implementiert werden kann. Die Klasse *Waggon* dient als Oberklasse von Personenwagen und Güterwagen, die auf dem Bildschirm dargestellt und bewegt werden können. Ein Waggon befindet sich stets auf einer genau definierten Position des Bildschirms, angegeben durch die Koordinaten des Mittelpunktes des linken Puffers.

In den Unterklassen *muss* der Auftrag `zeichne` überschrieben werden. Jeder Waggon hat einen Stift, der unter dem Namen `hatStift` angesprochen werden kann.

Auftrag	init (pHPosition, pVPosition : Zahl)
nachher	Der Waggon ist initialisiert, d.h. er ist an der angegebenen Position auf dem Bildschirm dargestellt.
Auftrag	*zeichne*
nachher	(Dieser Dienst ist abstrakt und muss in den Unterklassen so überschrieben werden, dass nachher der Waggon an seiner aktuellen Position auf dem Bildschirm dargestellt ist.)
Auftrag	lösche
nachher	Der Waggon ist nicht mehr auf dem Bildschirm dargestellt.
Auftrag	bewegeUm (pDistanz : Zahl)
nachher	Der Waggon wurde um die angegebene Distanz nach links bewegt.
Anfrage	hPosition : Zahl
nachher	Diese Anfrage liefert die aktuelle horizontale Position des Waggons.
Anfrage	vPosition : Zahl
nachher	Diese Anfrage liefert die aktuelle vertikale Position des Waggons.
Auftrag	gibFrei
nachher	Der Waggon ist vom Bildschirm gelöscht und steht nicht mehr zur Verfügung.

Eine besondere Rolle spielt der Dienst ***zeichne*** in der Klasse ***Waggon***. Weil innerhalb der Methode ***lösche*** u.a. auch die Methode ***zeichne*** aufgerufen wird, muss also die Klasse ***Waggon*** nicht nur die Methode ***lösche*** sondern auch die Methode ***zeichne*** besitzen.

Auch wenn die Methode ***zeichne*** in der Klasse *Waggon* nicht realisiert werden kann, so muss sie doch als Vorlage für die Unterklassen vorhanden sein. Diese "nicht realisierbare" Methode heißt *abstrakt*. Sie muss in den Unterklassen *überschrieben* werden. Abstrakte Dienste werden im UML-Klassendiagramm *kursiv* gekennzeichnet.

Man sagt auch, dass die Methode *zeichne* aufgeschoben wird, bis ihre Realisierung in einer Unterklasse erfolgen kann.

Von einer abstrakten Klasse kann es keine Objekte geben (weil nicht alle Methoden definiert sind).

Damit lassen sich offensichtlich die beiden Unterklassen wesentlich vereinfacht darstellen:

Dokumentation der Klasse Gueterwagen

Oberklasse: *Waggon*

Ein Güterwagen ist ein Waggon, der auf dem Bildschirm dargestellt und bewegt werden kann. Er befindet sich stets auf einer genau definierten Position des Bildschirms, angegeben durch die Koordinaten des Mittelpunktes des linken Puffers.

Auftrag **zeichne**
nachher Der Güterwagen ist an seiner aktuellen Position auf dem Bildschirm dargestellt.

Es folgt nun der gesamte Programmcode:

```
unit mWaggon;
interface
USES mSuM;
Type Waggon = class(TObject)
       protected
         x, y: zahl;
         hatStift: Stift;
       public
         constructor init(px, py: zahl);
         procedure zeichne; virtual; abstract;
         procedure loesche;
         procedure bewegeUm(pDistanz: zahl);
         function hPosition: zahl;
         function vPosition: zahl;
         destructor gibFrei;
     end;

implementation

constructor Waggon.init(px, py: zahl);
  BEGIN
    hatStift:= Stift.init;
    x := px;
    y := py;
    zeichne;
  END;

destructor Waggon.gibFrei;
  BEGIN
    loesche;
    hatStift.gibFrei;
    inherited destroy;
  END;

procedure Waggon.loesche;
  BEGIN
    hatStift.radiere;
    zeichne;
```

```
    hatStift.normal;
  END;

function Waggon.hPosition: zahl;
  BEGIN
    RESULT := x
  END;

function Waggon.vPosition: zahl;
  BEGIN
    RESULT := y
  END;

procedure Waggon.bewegeUm(pDistanz: zahl);
  BEGIN
    loesche;
    x := x+pDistanz;
    zeichne
  END;
end.
```

```
unit mGueterwagen;
interface
USES mSuM, mWaggon;
Type Gueterwagen = class(Waggon)
        public
           procedure zeichne; override;
     end;

implementation

procedure Gueterwagen.zeichne;
  BEGIN
    hatStift.hoch;                // linker Puffer
    hatStift.bewegeBis(x,y);
    hatStift.bewegeBis(x,y-2);
```

```
    hatStift.runter;
    hatStift.bewegeBis(x,y+2);
    hatStift.bewegeBis(x,y);
    hatStift.bewegeBis(x+4,y);

    hatStift.bewegeBis(x+4,y-15);     //offener Kasten
    hatStift.bewegeBis(x+4,y+1);
    hatStift.bewegeBis(x+54,y+1);
    hatStift.bewegeBis(x+54,y-15);

    hatStift.hoch;                    // rechtes Rad
    hatStift.bewegeBis(x+45,y+6);
    hatStift.runter;
    hatStift.zeichneKreis(5);

    hatStift.hoch;                    // linkes Rad
    hatStift.bewegeBis(x+13,y+6);
    hatStift.runter;
    hatStift.zeichneKreis(5);

    hatStift.hoch;                // rechter Puffer
    hatStift.bewegeBis(x+58,y);
    hatStift.bewegeBis(x+58,y-2);
    hatStift.runter;
    hatStift.bewegeBis(x+58,y+2);
    hatStift.bewegeBis(x+58,y);
    hatStift.bewegeBis(x+54,y);
  END;

end.
```

```
unit mPersonenwagen;

interface
USES mSuM, mWaggon;
Type Personenwagen = class(Waggon)
        public
           procedure zeichne; override;
     end;

implementation

procedure Personenwagen.zeichne;
  BEGIN
    hatStift.hoch;                      // linker Puffer
    hatStift.bewegeBis(x,y);
    hatStift.bewegeBis(x,y-2);
    hatStift.runter;
    hatStift.bewegeBis(x,y+2);
    hatStift.bewegeBis(x,y);
    hatStift.bewegeBis(x+4,y);


    hatStift.bewegeBis(x+4,y-20);      //großer Kasten
    hatStift.zeichneRechteck(50,22);

    hatStift.bewegeBis(x+4,y-24);
    hatStift.bewegeBis(x+8,y-30);
    hatStift.bewegeBis(x+50,y-30);
    hatStift.bewegeBis(x+53,y-24);
    hatStift.bewegeBis(x+53,y-20);

    hatStift.hoch;                     // Tür
    hatStift.bewegeBis(x+10,y-15);
    hatStift.runter;
    hatStift.zeichneRechteck(7,15);

    hatStift.hoch;                     // Fenster
    hatStift.bewegeBis(x+20,y-15);
```

```
    hatStift.runter;
    hatStift.zeichneRechteck(30,10);

    hatStift.hoch;                      // rechtes Rad
    hatStift.bewegeBis(x+45,y+6);
    hatStift.runter;
    hatStift.zeichneKreis(5);

    hatStift.hoch;                      // linkes Rad
    hatStift.bewegeBis(x+13,y+6);
    hatStift.runter;
    hatStift.zeichneKreis(5);

    hatStift.hoch;                  //rechter Puffer
    hatStift.bewegeBis(x+58,y);
    hatStift.bewegeBis(x+58,y-2);
    hatStift.runter;
    hatStift.bewegeBis(x+58,y+2);
    hatStift.bewegeBis(x+58,y);
    hatStift.bewegeBis(x+54,y);
  END;

end.
```

Aufgaben

1. Erstelle auf dem Bildschirm einen Zug, der aus drei Waggons besteht! Der Zug soll sich von links nach rechts bewegen.

2. Der Zug aus der Aufgabe 1 soll an den Bildschirmrändern reflektiert werden.

3. Der Zug reagiert auf Tasteneingaben. Bei „+“ wird er schneller, bei „-„ langsamer.

4. Der Zug sollte eine zusätzliche Lokomotive als Zugmaschine erhalten. Erstelle also eine neue Unterklasse von *Waggon* namens *Lokomotive*!

5. Eine einzige Lokomotive soll im Kreis fahren können. Dafür müssten in der Methode *zeichne* fast alle *bewegeBis-Befehle* durch *bewegeUm-Befehle* ersetzt werden. Außerdem benötigt die Lokomotive eine neue Eigenschaft *Richtung*, welche die Himmelsrichtung (in Grad) angibt, in welche die Lokomotive fährt.

6. Ein ganzer Zug soll im Kreis fahren können.

7. Erstelle eine abstrakte Oberklasse namens *Figur*, welche zwei Unterklassen namens *Quadrat* und *GleichseitigesDreieck* besitzt! Gemeinsame Eigenschaften sind der Ort des linken unteren Eckpunktes und die Länge einer Seite. Außerdem sollen unter jeder gezeichneten Figur der Umfang und die Fläche angegeben werden. Das heißt, die beiden Funktionen *Umfang* und *Flaeche* sind abstrakte Methoden der Oberklasse, welche in den Unterklassen konkretisiert werden müssen.

Lösung Aufgabe 1

```
unit mHaupt;
interface
uses  …Dialogs, mSuM, mGueterwagen, mPersonenwagen;

type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  Main: TMain;
  meinPersonenwagen: Personenwagen;
  meinGueterwagen: Gueterwagen;
  meinBildschirm: Bildschirm;
  meineMaus: Maus;

implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
  meinBildschirm:= Bildschirm.init;
  meineMaus:= Maus.init;
  meinPersonenwagen:= Personenwagen.init(100,200);
  meinGueterwagen:= Gueterwagen.init(160,200);

  REPEAT
    meinGueterwagen.bewegeUm(1);
    meinPersonenwagen.bewegeUm(1);
    sleep(5)
  UNTIL meineMaus.doppelklick;

  meinPersonenwagen.gibFrei;
  …………..
end;
end.
```

Klassen und Konstruktoren

Das Kennzeichen eines Konstruktors ist, dass er an einer Klasse (nicht an einer Instanz, also einem Objekt) aufgerufen wird, wodurch eine Instanz erzeugt wird (Reservierung von Speicher usw.). Der Aufruf normaler Methoden, die mit *procedure* oder *function* beginnen, ist erst möglich, wenn eine Instanz existiert. Um dies zu unterscheiden, beginnt die Deklaration eines Konstruktors mit dem Schlüsselwort *constructor*:

Der Konstruktor bezeichnet eine sog. Klassenmethode. Klassenmethoden sind Methoden, die nicht schon die Existenz eines Objektes dieser Klasse voraussetzen. Beim Konstruktor ist dies unmittelbar einsichtig, denn durch dessen Aufruf wird ja erst ein entsprechendes Klassenobjekt erzeugt.
Es können jedoch auch noch weitere Klassenmethoden oder sogar Klassen-konstanten existieren.

Konstruktoren müssen (sinnvollerweise !) vom Typ *public* sein.

Eine Klassenmethode wird durch Voranstellen des Klassennamens aufgerufen.
Beispiel: x := TBall.create

Der Name eines Konstruktors könnte prinzipiell beliebig sein, üblicherweise wird jedoch der Name *create* gewählt, manchmal auch *init*.

Obwohl die Deklaration keinen Rückgabewert enthält, gibt ein Konstruktor immer einen Verweis auf das Objekt, das er erstellt, zurück.

Eine Klasse kann auch mehrere Konstruktoren haben. Im Normalfall hat sie jedoch nur einen mit der Bezeichnung Create.

Wird explizit kein Konstruktor implementiert, so wird automatisch der Konstruktor der übergeordneten Klasse aufgerufen.

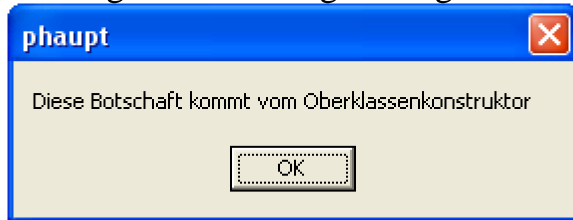
Betrachte dazu folgendes Programm:

```
unit mKlassen;
interface
uses Dialogs;
type
  Oberklasse = class(TObject)
     public
       wort: String;
       constructor create;
  end;

  Unterklasse = class(Oberklasse)
     public
       inhalt: Integer;
  end;

implementation

constructor Oberklasse.create;
BEGIN
  showmessage('Diese Botschaft kommt vom Oberklassenkonstruktor')
END;

END.
```

```
unit mHaupt;
interface

uses  … Dialogs, mKlassen;
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  Main: TMain;
  UKObjekt: Unterklasse;
  OKObjekt: Oberklasse;

implementation
{$R *.dfm}
```

```
procedure TMain.FormCreate(Sender: TObject);
begin
  UKObjekt := Unterklasse.create;
  UKObjekt.inhalt := 17;
end;

end.
```

Das Ergebnis des obigen Programms ist folgendes Fenster:



Enthält hingegen die Unterklasse einen eigenen Konstruktor, so wird **nur** dieser aufgerufen, der Konstruktor der Oberklasse jedoch nicht mehr.

Oft muss bei der Erzeugung eines Unterklassenobjektes dasselbe initialisiert werden wie in der Oberklasse, und nur zusätzlich noch einige Attribute der Unterklasse mehr. Dieses Problem wird folgendermaßen gelöst:

```
constructor Unterklasse.create;
BEGIN
  inherited create;
  unterklasseAttribut := …
END;
```

Statt *inherited create* genügt auch die bloße Anweisung *inherited*, solange der Konstruktorname in Unter- und Oberklasse identisch ist. Falls die Unterklasse einen eigenen Konstruktor besitzt, sollte dort immer zuerst *inherited* aufgerufen werden.

Wie funktioniert ein Konstruktor?
Der Konstruktor reserviert zunächst Speicher für das neue Objekt auf dem sog. Heap (das ist ein bestimmter Speicherbereich). Anschließend werden alle Attribute der Klasse initialisiert. Ordinalfelder werden mit dem Wert Null, alle Zeiger mit nil und alle Strings mit einem Leerstring initialisiert. Aus diesem Grund brauchen vom Programmierer nur noch die Attribute initialisiert zu werden, denen ein bestimmter Anfangswert zugewiesen werden soll. Danach werden alle weiteren Aktionen in der Implementierung des Konstruktors ausgeführt. Am Ende gibt der Konstruktor einen Verweis auf das neu erstellte und initialisierte Objekt zurück.

Ein Zuweisungsversuch ObjektKlasseA := ObjektKlasseB ist bekanntlich im allgemeinen nicht möglich.
Für den Sonderfall, dass A eine Oberklasse von B ist, ist dies jedoch möglich.
Beispiel: OKObjekt := UKObjekt;
In diesem Fall werden nur die syntaktisch möglichen Werteübertragungen vorgenommen, und OKObjekt greift als Instanz der Oberklasse sowieso nur auf die in der Oberklasse bereitgestellten Attribute und Dienste zu. OKObjekt kann also nicht auf die zusätzlichen Attribute oder Methoden der Unterklasse zugreifen, weil OKObjekt immer noch auf Grund seiner Deklaration eine Instanz der Oberklasse ist und bleibt.

Eine etwaige umgekehrte Zuweisung der Form UKObjekt := OKObjekt; wird schon beim Compilieren als falsch erkannt.

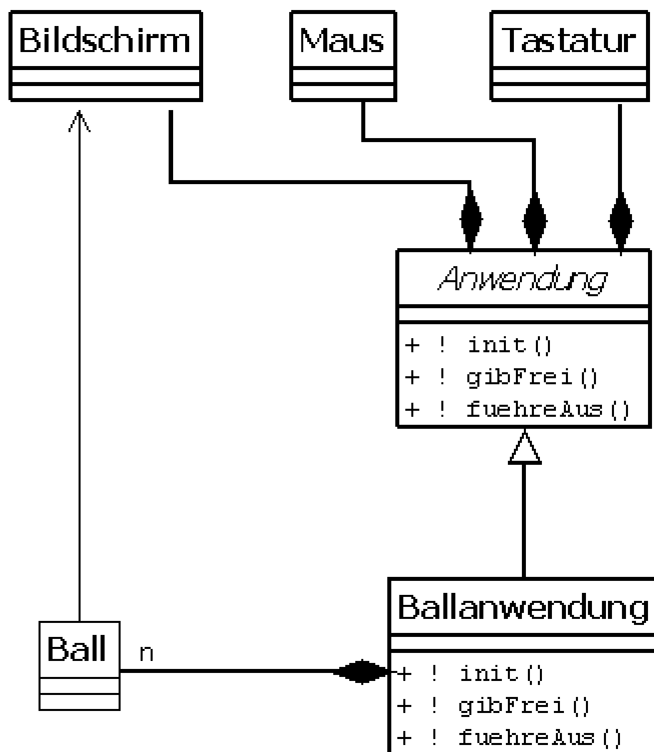
Erlaubt ist allerdings beispielsweise folgende Zuweisung:
UKObjekt := irgendeinObjekt as Unterklasse;
wobei die Variable irgendeinObjekt z.B. vom Typ TObject ist.
Sinn macht das allerdings nur, wenn der Programmierer genau weiß, dass *irgendeinObjekt* in diesem Moment als eine Instanz der Unterklasse interpretiert werden kann.

Die Klasse *AnwendungNeu*

Vergleicht man alle bisherigen Hauptprogramme miteinander, so erkennt man die gemeinsame Struktur:

> Initialisierung der benötigten Objekte.
> Jeweils unterschiedlicher Ausführungsteil.
> Freigabe der benutzten Objekte.

Es ist sinnvoll, für all diese Hauptprogramme eine gemeinsame Klasse namens ***Anwendung*** zu konstruieren. Konkrete Programme werden dann als Unterklassen dieser Klasse *Anwendung* deklariert.



Bemerkung: in dem Modul mSuM gibt es bereits eine Klasse Anwendung, welche allerdings keinen Stift besitzt.

Dokumentation der Klasse „AnwendungNeu“

AnwendungNeu ist eine abstrakte Klasse als Basis für alle weiteren Anwendungen. Sie besitzt bereits einen Bildschirm, eine Maus, einen Stift und eine Tastatur.
Bei der Realisierung von weiteren Anwendungen als Unterklasse muss insbesondere die abstrakte Methode *fuehreAus* überschrieben werden. Die Dienste *init* und *gibFrei* müssen aufgerufen werden.
Nur in einer Unterklasse darf auf die oben aufgeführten Bezugsobjekte über die Namen „hatBildschirm“, „hatTastatur“, „hatStift“ und „hatMaus“ zugegriffen werden.

Auftrag	init
nachher	Die Anwendung ist initialisiert. Die Objekte Bildschirm, Maus, hatStift und Tastatur sind erzeugt und initialisiert.

Auftrag	*fuehreAus*
	abstrakter Auftrag, der in einer Unterklasse erfüllt werden muss.

Auftrag	gibFrei
nachher	Die Anwendung steht nicht mehr zur Verfügung. Bildschirm, Maus, Stift und Tastatur wurden freigegeben.

Im Folgenden wird der Quellcode dargestellt:

```
unit mAnwendung;

interface
USES mSuM;
```

```
Type AnwendungNeu = class(TObject)
       protected
         hatStift: Stift
         hatMaus: Maus;
         hatBildschirm: Bildschirm;
         hatTastatur: Tastatur;
       public
         constructor init; virtual;
         destructor gibFrei; virtual;
         procedure fuehreAus; virtual; abstract;
     end;

implementation


constructor Anwendung.init;
  BEGIN
    hatMaus := Maus.init;
    hatBildschirm := Bildschirm.init;
    hatStift := Stift.init;
    hatTastatur := Tastatur.init;
  END;

destructor Anwendung.gibFrei;
  BEGIN
    hatTastatur.gibFrei;
    hatStift.gibFrei;
    hatBildschirm.gibFrei;
    hatMaus.gibFrei;
  END;

end.
```

```
unit mBallanwendung;

interface
USES mAnwendung, mBall;
Type Ballanwendung = class(AnwendungNeu)
       private
         meinBall1, meinBall2: Ball;
       public
         constructor init; override;
         destructor gibFrei; override;
         procedure fuehreAus; override;
     end;

implementation

constructor Ballanwendung.init;
  BEGIN
    inherited init;
    meinBall1 := Ball.init(300,200,0,hatBildschirm);
    meinBall2 := Ball.init(200,200,90,hatBildschirm);
  END;

destructor Ballanwendung.gibFrei;
  BEGIN
    meinBall1.gibFrei;
    meinBall2.gibFrei;
    inherited gibFrei;
  END;

procedure Ballanwendung.fuehreAus;
  BEGIN
    REPEAT
      meinBall1.bewege;
      meinBall2.bewege;
    UNTIL hatMaus.doppelklick;
  END;

end.
```

```
unit mHaupt;

interface
uses ….. mBallanwendung;    // Beachte: mSuM fehlt hier
type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var
  Main: TMain;
  meinProgramm: Ballanwendung;

implementation
{$R *.dfm}


procedure TMain.FormCreate(Sender: TObject);
begin
  meinProgramm := Ballanwendung.init;
  meinProgramm.fuehreAus;
  meinProgramm.gibFrei;
end;

end.
```

Aufgaben

1. Auch für die Mühle soll eine entsprechende Mühlenanwendung geschaffen werden.

2. Projekt Kompass
Auf dem Bildschirm sollen ein oder mehrere Kompasse erscheinen, deren Nadeln auf die Maus als „Magnetischen Nordpol" zeigen. Die Nadeln folgen jeder Mausbewegung.
Hinweise: Benutze die Funktion *arctan*, beachte, dass das Ergebnis in Radiant angegeben wird (und nur Werte zwischen -π/2 und +π/2 besitzt), die Drehmethoden des Stiftes aber eine Gradangabe benötigen.

3. Projekt Autorennen
Zwei Autos sollen von einem Startpunkt zu einem Zielpunkt fahren. Die Autofahrt soll auf dem Bildschirm dargestellt werden. Verwende eine Anwendungsklasse namens ***Autofahrt***!

```
   Start                                                          Ziel
------------------------------------------------------------------------------------
                        o-o
------------------------------------------------------------------------------------
                        o-o
------------------------------------------------------------------------------------
```

Die Autos werden diesmal durch Zeichenketten dargestellt. Die Autos können mitteilen, ob sie am Ziel angekommen sind.

a) die Autos fahren gleich schnell.

b) es wird immer wieder zufällig bestimmt, welches Auto sich als nächstes bewegt. Auf dem Bildschirm erscheint andauernd eine Meldung wie etwa „Auto 1 liegt vorn".

c) jedes Auto bewegt sich nur auf einen bestimmten Tastendruck hin. So können zwei Spieler ein Rennen fahren, indem sie möglichst schnell „ihre" Taste betätigen. Um zu vermeiden, dass eine Taste permanent niedergehalten wird, bewegt sich z.B. das Auto 1 nur, wenn abwechselnd die Tasten „a" und „s" gedrückt werden. Zuerst bewegt es sich nur bei „a", danach nur bei „s", dann wieder nur bei „a" usw. Das Auto 2 reagiert analog auf die Tasten „k" und „l".

d) Gib das Klassendiagramm und die Dokumentation für die Klasse ***Auto*** an!

Lösungen

Aufgabe 2

```
unit mKompass;

interface
USES mSuM;

Type Kompass = class(TObject)
       private
         hatStift: Stift;
         zxM, zyM: zahl;
         kMaus: Maus;
         procedure zeichneZeiger;
       public
         constructor init(px, py: zahl; pm:Maus);
         procedure ausrichten;
         destructor gibFrei;
     end;

implementation

constructor Kompass.init(px, py: zahl;pm:Maus);
 BEGIN
   zxM := px;
   zyM := py;
   kMaus:= pm;
   hatStift := Stift.init;
   hatStift.bewegeBis(zxM,zyM);
   hatStift.zeichneKreis(20);
   hatStift.runter
 END;


procedure Kompass.zeichneZeiger;
BEGIN
   hatStift.bewegeUm(-7);
   hatStift.bewegeUm(14);
   hatStift.dreheUm(135);
```

```
   hatStift.bewegeUm(3);
   hatStift.bewegeUm(-3);
   hatStift.dreheUm(-270);
   hatStift.bewegeUm(3);
   hatStift.bewegeUm(-3);
   hatStift.dreheUm(135);
   hatStift.bewegeUm(-7)
END;


procedure Kompass.ausrichten;
VAR winkel: zahl;
 BEGIN
   hatStift.radiere;
   zeichneZeiger;

   IF kMaus.hPosition - zxM <> 0 THEN BEGIN
      winkel := arctan((kMaus.vPosition - zyM)/
                    (kMaus.hPosition - zxM)) *180/pi;
      IF (kMaus.vPosition <> zyM) THEN
        IF (kMaus.hPosition > zxM) THEN
            winkel := -winkel
        ELSE winkel := 180-winkel
      ELSE IF kMaus.hPosition > zxM THEN winkel := 0
           ELSE winkel := 180
   END
   ELSE IF kMaus.vPosition < zyM THEN winkel := 90
        ELSE winkel := -90;

   hatStift.dreheBis(winkel);
   hatStift.normal;
   zeichneZeiger
 END;
```

```
destructor Kompass.gibFrei;
 BEGIN
   hatStift.gibFrei;
   inherited destroy
 END;


end.
```

```
Unit mKompassanwendung
Interface
USES mAnwendung, mKompass, SysUtils;
Type Kompassanwendung = class(Anwendung)
       private
         meinKompass1, meinKompass2: Kompass;
       public
         constructor init;  override;
         destructor gibFrei; override;
         procedure fuehreAus; override;
     end;


implementation

constructor Kompassanwendung.init;
  BEGIN
    inherited init;
    meinKompass1 := Kompass.init(300,200,hatMaus);
    meinKompass2 := Kompass.init(200,200,hatMaus);
  END;

destructor Kompassanwendung.gibFrei;
  BEGIN
    meinKompass1.gibFrei;
    meinKompass2.gibFrei;
    inherited gibFrei;
  END;
```

```
procedure Kompassanwendung.fuehreAus;
  BEGIN
    REPEAT
      meinKompass1.ausrichten;
      meinKompass2.ausrichten;
      sleep(1);
    UNTIL hatMaus.doppelklick;
  END;

end.
```

```
unit mHaupt;
interface
uses … mKompassanwendung;

type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;

var  Main: TMain;
     meinProgramm: Kompassanwendung;
implementation
{$R *.dfm}

procedure TMain.FormCreate(Sender: TObject);
begin
  meinProgramm := Kompassanwendung.init;
  meinProgramm.fuehreAus;
  meinProgramm.gibFrei;
end;

end.
```

Die Klasse EreignisanwendungNeu

Bemerkung: in dem Modul mSuM gibt es bereits eine Klasse Ereignis-anwendung.

Es gibt viele Programme, die warten permanent auf ein eintretendes Ereignis wie z.B. Mausdruck, Mausloslassen, Mausklick, Mausdoppelklick oder Tastatureingabe. Ist dieses Ereignis erfolgt, so wird entsprechend reagiert. Um solche Programme zu ermöglichen, wird die Klasse *EreignisanwendungNeu* eingeführt. Ein Objekt dieser Klasse ist praktisch ein Programm, welches dauernd auf ein solches Ereignis wartet und die entsprechende Reaktion darauf von einem Unterprogramm (entspricht hier einem Objekt der Unterklasse) ausführen lässt. Diese Reaktion darf in unserem Falle nicht zu lange dauern, ansonsten könnten zwischenzeitlich eintretende Ereignisse nicht registriert werden.

```
unit mEreignisanwendung;


interface
USES mSuM, mAnwendungNeu;

type EreignisanwendungNeu=class(AnwendungNeu)
  private
    zbeendet:Wahrheitswert;
    zMausGedrueckt:Wahrheitswert;
    zAlteHPosition,zAlteVPosition:Zahl;
  public
    constructor init; override;
  protected
    procedure fuehreAus; override;
    procedure beenden;
    procedure bearbeiteTaste(pZeichen:Zeichen);virtual;
    procedure bearbeiteMausdruck(ph,pv:Zahl); virtual;
    procedure bearbeiteMausLos(ph,pv:Zahl); virtual;
```

```
    procedure bearbeiteDoppelklick(ph,pv:Zahl);virtual;
    procedure bearbeiteMausBewegt(ph,pv:Zahl); virtual;
    procedure bearbeiteLeerlauf; virtual;
end;

implementation


constructor EreignisanwendungNeu.init;
begin
  inherited init;
  zbeendet:=false;
  zMausGedrueckt:=false;
  zAlteHPosition:=hatMaus.HPosition;
  zAlteVPosition:=hatMaus.VPosition;
end;


procedure EreignisanwendungNeu.fuehreAus;
begin
  repeat
    if hatMaus.IstGedrueckt and not zMausGedrueckt
    then begin
      self.BearbeiteMausDruck(hatMaus.hPosition,
                              hatMaus.vPosition);
      zMausGedrueckt:=true;
    end
    else if zMausGedrueckt and not hatMaus.IstGedrueckt
         then begin
           self.BearbeiteMausLos(hatMaus.hPosition,
                                 hatMaus.vPosition);
           zMausGedrueckt:=false;
         end;

    if hatMaus.doppelklick then
      self.BearbeiteDoppelklick(hatMaus.hPosition,
                                hatMaus.vPosition);

    if hatTastatur.wurdeGedrueckt then begin
```

```
        bearbeiteTaste(hatTastatur.Zeichen);
        hatTastatur.weiter
     end;

     if (hatMaus.hPosition<>zAlteHPosition) or
                     (hatMaus.vPosition<>zAlteVPosition)
     then begin
       zAlteHPosition:=hatMaus.HPosition;
       zAlteVPosition:=hatMaus.VPosition;
       self.bearbeiteMausBewegt(zAlteHPosition,
                                      zAlteVPosition);
     end;

     self.bearbeiteLeerlauf;

  until zbeendet;
end;


procedure EreignisanwendungNeu.beenden;
begin
  zbeendet:=true;
end;


procedure
 EreignisanwendungNeu.bearbeiteTaste(pZeichen:Zeichen);
begin
    {abstrakt}
end;


procedure
 EreignisanwendungNeu.bearbeiteMausdruck(ph,pv:Zahl);
begin
    {abstrakt}
end;
```

```
procedure
   EreignisanwendungNeu.bearbeiteMausLos(ph,pv:Zahl);
begin
      {abstrakt}
end;


procedure
EreignisanwendungNeu.bearbeiteDoppelklick(ph,pv:Zahl);
begin
      {abstrakt}
end;

procedure
EreignisanwendungNeu.bearbeiteMausBewegt(ph,pv:Zahl);
begin
      {abstrakt}
end;


procedure EreignisanwendungNeu.bearbeiteLeerlauf;
begin
      {abstrakt}
end;

end.
```

```
unit mTestEreignisAnwendung;

interface

USES mEreignisAnwendung, mSuM;

Type TestEreignisAnwendung = class(EreignisAnwendungNeu)
     public
       // procedure bearbeiteTaste(pZeichen:Zeichen); override;
```

```
    // procedure bearbeiteMausdruck(ph,pv:Zahl); override;
     procedure bearbeiteMausLos(ph,pv:Zahl);override;
     procedure bearbeiteDoppelklick(ph,pv:Zahl);override;
    // procedure bearbeiteMausBewegt(ph,pv:Zahl); override;
    // procedure bearbeiteLeerlauf; override;
   end;



implementation


procedure
  TestEreignisAnwendung.bearbeiteMausLos(ph,pv:Zahl);
BEGIN
  hatStift.bewegeBis(100,100);
  hatStift.schreibeText('Hallo');
END;


procedure
TestEreignisAnwendung.bearbeiteDoppelklick(ph,pv:Zahl);
BEGIN
  beenden
END;

end.
```

```
unit mHauptprogramm;

interface
uses …..Dialogs, mTestEreignisAnwendung;

type
  TMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  public
    myProgram: TestEreignisAnwendung;
  end;

var  Main: TMain;

implementation
{$R *.dfm}


procedure TMain.FormCreate(Sender: TObject);
begin
  myProgram := TestEreignisAnwendung.init;
  myProgram.fuehreAus;
  myProgram.gibFrei
end;

end.
```