3. Modellierung einfacher geometrischer Objekte

3.1 Ziel

In diesem Kapitel werden die Objektklassen `TStrecke`, `TKreis` und `TQuadrat` modelliert, mit denen man die entsprechenden Figuren im Hauptprogramm zeichnen lassen kann.
Zunächst wird jede der drei Klassen als Nachkomme von `TObject` deklariert.
Danach werden schrittweise die Gemeinsamkeiten der drei Klassen herausgearbeitet und auf dieser Grundlage gemeinsame Vorfahren konstruiert. Außerdem verwenden Sie erstmals im Zusammenhang mit OOP Methoden mit Parametern (insbesondere den Konstruktor `create`); und schließlich lernen Sie **abstrakte Methoden** kennen.

3.2 Aufgabe: TKreis, TQuadrat und TStrecke zunächst als Nachkomme von TObject

1. Starten Sie eine neue Lazarus-Anwendung; benennen Sie bei Projekt → Projekt speichern unter... die Unit1 in „uHaupt“ um. Speichern Sie alles in einen neuen Ordner „02GeoObject“ und kopieren Sie aus dem Tauschverzeichnis die Unit „uFigur“ in diesen Ordner. Anschließend wird sie in drei Schritten in das Projekt eingebunden:

   i) Datei → Öffnen… Öffnen Sie die Datei „uFigur.pas“.
   ii) Projekt → Datei im Editor ins Projekt aufnehmen...:
   iii) „uFigur“ in der `uses`-Liste von „uHaupt“ mit aufzählen.

Wir beginnen mit „uFigur“:

2. Gleich nach `interface` wird die `uses`-Liste angegeben:

```
uses graphics,types,math;
```

3. Deklaration der Objektklasse `TKreis`:

```
type
  TKreis = class(TObject)
  protected
    Canvas:TCanvas;
    xM,yM,r:integer ;
    procedure setr(neu:integer);
    procedure setM(neu:TPoint);
    function getM:TPoint;
    procedure konstruieren;
  public
    farbe:TColor;
    constructor create(iCanvas:TCanvas;iFarbe:TColor;ixM,iyM,ir:integer);
    procedure zeichnen;
    procedure loeschen;
    procedure verschieben(dx,dy:integer);
    property Radius:integer read r write setr;
    property Mittelpkt:TPoint read getM write setM;
  end;
```

Grundsätzlich gilt: Wenn unter `private`, `protected` oder `public` sowohl Variablen als auch Prozeduren und Funktionen deklariert werden sollen, muss die Deklaration der **Variablen zuerst** erfolgen.

Im folgenden werden die hier deklarierten Attribute und Methoden in der Reihenfolge ihres Auftretens erläutert. Der Vermerk „(→*Schon vorhanden*)“ bedeutet, dass die Prozedur bereits fertig unter `implementation` programmiert ist; versuchen Sie dann, den Quelltext zu verstehen. Der Vermerk „(→*Selbst programmieren*)“ heißt, dass Sie selber etwas tun müssen.

Unter `protected`:

`Canvas:TCanvas;`

Die Klasse `TCanvas` stellt Methoden zum Zeichnen zur Verfügung; zur Erinnerung seien noch einmal die wichtigsten aufgeführt:

`moveto(x1,y1:integer)`	Setzt den Stift auf die angegebenen Position, ohne zu zeichnen.
`lineto(x1,y1:integer)`	Linie von der aktuellen Position zum Punkt (x1|y1)
`rectangle(x1,y1,x2,y2:integer)`	Rechteck, linke obere Ecke (x1|y1); rechte untere Ecke (x2|y2).
`ellipse(x1,y1,x2,y2:integer)`	Koordinaten wie beim Rechteck
`pixels[x,y:integer]`	Lese- und Schreibzugriff auf die Pixelfarbe am Punkt (x,y).
`pen:TPen, Brush:TBrush`	Zeichenstift für Umrisse, Malerpinsel zum Ausfüllen
`pen.color, brush.color`	Lese- und Schreibzugriff auf die zugehörige Farbe

Die „Variable“ `Canvas` vom Typ `TCanvas` wird benötigt, um später im Hauptprogramm festlegen zu können, auf welche Zeichenfläche gezeichnet werden soll: Direkt ins Formular oder in eine Paintbox oder eine Image-Komponente.

`xM,yM,r:integer;`
Koordinaten des Kreismittelpunktes und der Kreisradius. Diese Variablen sind unter `protected` deklariert, da diese Variablen im Hauptprogramm nur gelesen, aber nicht durch direkten Schreibzugriff verändert werden dürfen. Beispielsweise müssen negative Radien stets ausgeschlossen sein.

`procedure` `setr(neu:integer);` (→*Selbst programmieren*)
Weist der Variablen r den Wert neu zu. Falls `neu` negativ ist, soll r den Wert 0 annehmen.

`procedure` `setM(neu:TPoint);` (→*Schon vorhanden*)
Weist dem Mittelpunkt die Koordinaten zu. Nebenbei lernen Sie einen neuen, bereits in Lazarus implementierten Datentyp `TPoint` kennen. Er hat nichts mit objektorientierter Programmierung zu tun, sondern fasst einfach die x- und y-Koordinate eines Punktes in einem zweidimensionalen Vektor zusammen.

`function` `getM:TPoint;` (→*Schon vorhanden*)
Das Gegenstück, um die x- und die y-Koordinate des Mittelpunktes zu lesen.
Die Prozedur `setM` und die Funktion `getM` bilden also ein Methoden-Paar zum Schreiben bzw. Lesen ein und derselben Größe. Üblicherweise beginnt der Name der „Schreib-Prozedur“ stets mit „set“ (für „setzen“) und der Name der „Lesefunktion“ stets mit „get“ (für „bekommen“); ansonsten sind die Namen identisch. Vor allem bei großen Programmen ist es dringend zu empfehlen, diese Namenskonvention einzuhalten.

`procedure` `konstruieren;` (→*Selbst programmieren*)
Diese Prozedur enthält bloß eine einzige Anweisung: Die `ellipse`-Anweisung zum Zeichnen des Kreises mit dem Mittelpunkt ($x_M|y_M$) und dem Radius r. `pen.color` und `brush.color` werden nicht hier festgelegt.

Unter `public:`

`farbe:TColor;`
Umriß und Füllung des Kreises sollen stets dieselbe Farbe haben. Da die Farbe auch vom Hauptprogramm aus geändert werden darf, ist sie unter `public` deklariert.

`constructor` `create(iCanvas:TCanvas;iFarbe:TColor;ixM,iyM,ir:integer);` (→*Schon vorh.*)
Der Konstruktor benötigt den Parameter `iCanvas` vom Typ `TCanvas`, damit man schon bei seinem Aufruf im Hauptprogramm, also beim Erzeugen eines Kreises, festlegen kann, ob auf die Zeichenfläche des Formulars, einer Paintbox oder einer Image-Komponente gezeichnet werden soll. Weiterhin hängt er von der Zeichenfarbe, den Koordinaten des Mittelpunktes und dem Kreisradius ab, damit man auch die entsprechenden Anfangswerte ebenfalls sofort beim Aufruf des Konstruktors festlegen kann.
Dass alle Parameter den Anfangsbuchstaben i haben, ist eine Konvention, um anzudeuten, dass es sich um Anfangswerte („init“) handelt. Wie Sie im Quelltext sehen, werden den projektinternen Variablen gleich hier im Konstruktor diese Parameter als Anfangswerte zugewiesen; die projektinternen Variablen haben auch die gleichen Namen, nur ohne das „i“.
Erinnerung: Die Anweisung

```
inherited create;
```

ist nötig, damit allen Instanzen der Klasse `TKreis` die Eigenschaften und Methoden des Vorfahren `TObject` vererbt werden.

`procedure` `zeichnen;` (→*Selbst programmieren*)
Weisen Sie `canvas.pen.color` und `canvas.brush.color` die Zeichenfarbe zu, rufen Sie anschließend die Prozedur `konstruieren` auf.

`procedure` `loeschen;` (→*Selbst programmieren*)
Wie `Zeichnen`; nur eben mit der Hintergrundfarbe `clbtnface`.

`procedure` `verschieben(dx,dy:integer);` (→*Selbst programmieren*)
Erhöhen Sie `xM` um `dx`, `yM` um `dy`; rufen Sie an den richtigen Stellen die Prozeduren `zeichnen` und `loeschen` auf.

`property` `Radius:integer` **`read`** `r` **`write`** `setr;`
Diese Deklaration steht nur unter `interface` und erscheint nicht noch einmal unter `implementation`. Sie bewirkt folgendes: Ein Lesezugriff im Hauptprogramm, z.B

```
if kreis1.Radius < 40
```

liest den Wert der internen Variablen `r`, die vom Hauptrogramm aus nicht sichtbar ist. Bei einem Schreibzugriff wie etwa

```
kreis1.Radius:=50;
```

wird intern die Prozedur `setr` ausgeführt; d.h. der internen Variablen `r` wird der Wert 50 zugewiesen.

`property` `Mittelpunkt:TPoint` **`read`** `getM` **`write`** `setM;`

Für den Mittelpunkt gilt das Gleiche wie für den Radius; nur dass beim Lesezugriff im Hauptrogramm, beispielsweise mit

```
edit1.text:=inttostr(kreis1.Mittelpunkt.X);
```

nicht einfach eine interne Variable gelesen wird, sondern dass die Funktion `getM` ausgeführt und ihr Wert zurückgegeben wird.

4. Deklaration und Implementierung von `TQuadrat`:
   a) Kopieren Sie die Deklaration von `TKreis` ab
   ```
   TKreis=class(TObject)
   ```
   bis zum `end` nach `public`. Fügen Sie die Kopie direkt nach diesem `end` ein. Ersetzen Sie in der Kopie `TKreis` durch `TQuadrat`.
   b) Folgende Änderungen sind notwendig: Variable `r` durch Variable `a` (für Seitenlänge) ersetzen; die Prozedur `setr` durch `seta` und die `property Radius` durch die `Property Seitenlaenge` und im Konstruktor den Parameter `ir` durch `ia`.
   c) Kopieren Sie unter `implementation` alle Methoden von `TKreis`; fügen Sie die Kopie darunter wieder ein.
   d) Markieren Sie die Kopie; mit Suchen/Ersetzen... `TKreis` durch `TQuadrat` ersetzen (markierter Text!).
   e) Die Prozedur `setr` in `seta` umbenennen.
   f) Alle Prozeduren umprogrammieren, in denen `r` vorkommt. Insbesondere bei der Prozedur `konstruieren` aufpassen, da die Seitenlänge des Quadrats dem Kreisdurchmesser, also dem doppelten Radius, entspricht.
5. Deklaration und Implementierung von `TStrecke`:
   a) Kopieren Sie die Deklaration von `TKreis`. Fügen Sie die Kopie direkt nach dem `end` der Deklaration von `TQuadrat` ein. Ersetzen Sie in der Kopie `TKreis` durch `TStrecke`.
   b) Folgende Änderungen sind notwendig: Variablen `xM`, `yM` durch `xP`, `yP` (Koordinaten des Anfangspunktes) ersetzen; das Paar `setM`; `getM` durch `setP`; `getP`. Die Variable `r:integer` durch `xQ,yQ:integer` (für den Endpunkt) ersetzen; die Prozedur `setr` durch `setQ;` zusätzlich die Funktion `getQ`. Die `Property Mittelpunkt` durch die `Property Anfangspunkt` und die `property Radius` durch die `Property Endpunkt;` im Konstruktor den Parameter `ir` durch die Koordinaten `ixQ,iyQ`.
   c) Kopieren Sie unter `implementation` alle Methoden von `TKreis`; fügen Sie die Kopie unter `TQuadrat` wieder ein.
   d) Markieren Sie die Kopie; mit Suchen/Ersetzen... `TKreis` durch `TStrecke` ersetzen. (markierter Text!)
   e) Prozeduren und Funktionen so umbenennen, dass sie genau mit der Deklaration übereinstimmen.
   f) Alle Prozeduren und Funktionen umschreiben, bei denen es nötig ist.
   g) Damit der gute, alte Pythagoras nicht in Vergessenheit gerät: Zusätzlich eine Funktion `Laenge:real` deklarieren und implementieren, die mithilfe der Funktionen `sqrt` (Wurzel) und `sqr` (Quadrat) die Länge der Strecke berechnet.

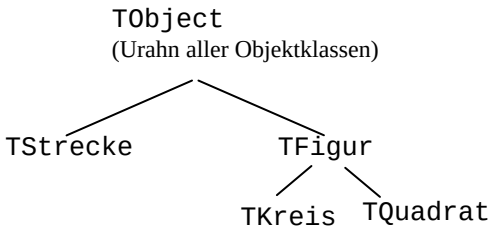
3.3 Aufgabe: uHaupt– Anwendung: Eine Ballonfahrt

1. Das Formular enthält eine Paintbox (401×401) aus der Registerkarte „Additional“, zwei Buttons „Start“ und „Reset“, sowie einen Timer (Registerkarte „System“, erstes Symbol links). Im `private`-Bereich des Formulars sind die Bauteile des Ballons deklariert:
   ```
   Korb:TQuadrat;
   Ballon:TKreis;
   seil1,seil2:Tstrecke;
   ```
2. Prozedur `FormCreate`: Erzeugen der 4 Objekte durch Aufruf des Konstruktors, z.B:
   ```
   Korb:=TQuadrat.create(paintbox1.Canvas,clmaroon,40,390,20);
   ```
   Um passende Koordinaten zu finden, empfehle ich eine Skizze auf Papier!
3. Prozedur `BtStartClick`: Zeichnen der 4 Objekte und Aktivieren des Timers.

4. Prozedur `Timer1Timer`: Zunächst Aufblasen des Ballons; sobald er eine gewisse Größe erreicht hat, Abflug. Einfachste Variante: Der Ballon fährt immer in dieselbe Richtung. Anspruchsvoller sind Richtungsänderungen in verschiedenen Flughöhen.
5. Prozedur `BtResetClick`: Deaktivieren des Timers; zurücksetzen der vier Ballon-Bauteile an den Startpunkt; der Ballon wird wieder im nicht aufgeblasenen Zustand gezeigt.
6. Prozedur `FormClose`: Freigeben der vier Ballon-Bauteile.

3.4 Der Vorfahr TFigur von TKreis und TQuadrat

Beim Modellieren der Klassen `TKreis` und `TQuadrat` in Aufgabe 3.2 haben Sie wahrscheinlich festgestellt, dass Sie den Deklarationsteil, die implementierten Prozeduren und Funktionen von `TKreis` bloß kopieren und nur an sehr wenigen Stellen abändern mussten. Vieles konnte gänzlich unverändert übernommen werden.
Alle Variablen, Funktionen und Prozeduren, die bei den beiden fast identischen Klassen `TKreis` und `TQuadrat` übereinstimmen, werden jetzt einem gemeinsamen Vorfahren `TFigur` zugeordnet, als dessen Nachkommen `TQuadrat` und `TKreis` später deklariert werden, so dass folgender „Stammbaum“ entsteht:



Bisher hatten die Deklarationsteile von `TKreis` und `TQuadrat` folgende Gestalt:

```
TKreis=class(TObject)
protected
  Canvas:TCanvas;
  xM,yM,r:integer ;
  procedure setr(neu:integer);
  procedure setM(neu:TPoint);
  function getM:TPoint;
  procedure konstruieren;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;iFarbe:TColor;
                     ixM,iyM,ir:integer);
  procedure zeichnen;
  procedure loeschen;
  procedure verschieben(dx,dy:integer);
  property Radius:integer read r write setr;
  property Mittelpunkt:TPoint read getM write setM;
end;
```

```
TQuadrat=class(TObject)
protected
  Canvas:TCanvas;
  xM,yM,a:integer ;
  procedure seta(neu:integer);
  procedure setM(neu:TPoint);
  function getM:TPoint;
  procedure konstruieren;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;iFarbe:TColor;
                     ixM,iyM,ia:integer);
  procedure zeichnen;
  procedure loeschen;
  procedure verschieben(dx,dy:integer);
  property Seitenlaenge:integer read a write seta;
  property Mittelpunkt:TPoint read getM write setM;
end;
```

Ersetzt man bei `TKreis` die Variable `r` und bei `TQuadrat` die Variable `a` durch eine Variable `d`, die bei `TKreis` den Radius und bei `TQuadrat` die Seitenlänge bedeutet, so findet man folgende, bereits unter `TFigur` zusammengefasste Gemeinsamkeiten:

```
TFigur=class(TObject)
protected
  Canvas:TCanvas;
  xM,yM,d:integer; //d als Radius bei TKreis, als Seitenlänge bei TQuadrat
  procedure setd(neu:integer);
  procedure setM(neu:TPoint);
  function getM:TPoint;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;iFarbe:TColor;ixM,iyM,id:integer);
  property Mittelpunkt:TPoint read getM write setM;
end;
```

Die hier deklarierten Eigenschaften und Methoden sind auch bei den Nachkommen `TQuadrat` und `TKreis` bekannt. Die Prozedur `konstruieren` hingegen kann nicht in den Vorfahren `TFigur` übernommen werden, da sie bei `TKreis` die `Ellipse`-Anweisung und bei `TQuadrat` die `Rectangle`-Anweisung enthält, um die entsprechende Figur zu zeichnen.

Folglich können zunächst auch nicht die Prozeduren `zeichnen` und `loeschen` nach `TFigur` übernommen werden, da sie die Prozedur `konstruieren` aufrufen, die in `TFigur` unbekannt ist. Da die Prozedur `verschieben` wiederum auf `loeschen` und `zeichnen` zurückgreift, kann zunächst auch sie nicht beim Vorfahren `TFigur` implementiert werden.
Später, im Kapitel 3.6 über abstrakte Methoden, werden Sie eine Möglichkeit kennen lernen, wie man die Prozeduren `zeichnen`, `loeschen` und `verschieben` trotzdem schon bei einem Vorfahren unterbringen kann, obwohl sie (direkt oder indirekt) auf verschiedene Implementierungen der Prozedur `konstruieren` zurückgreifen.

Ob man nun die `property Radius` bei `TKreis` belässt,

```
TKreis=class(TFigur)
...
property Radius:integer read d write setd;
```

und die `property Seitenlaenge` bei `TQuadrat`

```
TQuadrat=class(TFigur)
...
property Seitenlaenge:integer read d write setd;
```

oder stattdessen bei `TFigur` eine neue `property Groesse` einführt,

```
TFigur=class(TObject)
...
property Groesse:integer read d write setd;
```

ist Geschmacksfrage. Ich persönlich gebe der ersten Variante den Vorzug, da so der Quelltext eines Hauptprogramms, das Instanzen von `TKreis` und `TQuadrat` verwendet, besser verständlich wird.

Bei `TKreis` und `TQuadrat` bleiben jetzt nur noch die für Kreise und Quadrate spezifischen Eigenschaften und Methoden im Deklarationsteil stehen. Beachten Sie, dass `TKreis` und `TQuadrat` jetzt nicht mehr als Nachkommen von `TObject`, sondern als Nachkommen von `TFigur` deklariert sind:

```
TKreis=class(TFigur)
protected
  procedure konstruieren;
public
  procedure zeichnen;
  procedure loeschen;
  procedure verschieben(dx,dy:integer);
  property Radius:integer read d write setd;
end;
```

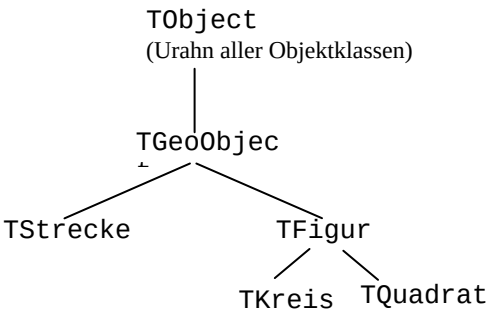
```
TQuadrat=class(TFigur)
protected
  procedure konstruieren;
public
  procedure zeichnen;
  procedure loeschen;
  procedure verschieben(dx,dy:integer);
  property Seitenlaenge:integer read d write setd;
end;
```

3.5 Aufgabe: TFigur implementieren

Beenden Sie ggf. Lazarus. Erzeugen Sie im Ordner „02GeoObject“ eine Kopie von „uFigur.pas“. Benennen Sie die Kopie in „uFigur1.pas“ um. Das ist jetzt die alte Version, wo `TKreis` und `TQuadrat` noch direkte Nachkommen von `TObject` sind.
Öffnen Sie Ihr Lazarus-Projekt und ändern Sie „uFigur“ so ab, dass `TKreis` und `TQuadrat` den Vorfahren `TFigur` bekommen. Wenn Ihr Hauptprogramm (die Ballonfahrt, Aufgabe 3.3) bisher funktioniert hat, sollte es auch jetzt funktionieren.

3.6 TFigur, TStrecke und ihr Vorfahr TGeoObject

Da auch `TFigur` und `TStrecke` noch vieles gemeinsam haben, soll nun auch für sie ein gemeinsamer Vorfahr `TGeoObject` modelliert werden:



Bisher sehen die Deklarationsteile von `TFigur` und `TStrecke` so aus:

```
TFigur=class(TObject)
protected
  Canvas:TCanvas;
  xM,yM,d:integer ;
  procedure setd(neu:integer);
  procedure setM(neu:TPoint);
  function getM:TPoint;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;iFarbe:TColor;ixM,iyM,id:integer);
  property Mittelpunkt:TPoint read getM write setM;
end;

TStrecke=class(TObject)
protected
  Canvas:TCanvas;
  xP,yP:integer ;
  xQ,yQ:integer;
  procedure setP(neu:TPoint);
  function getP:TPoint;
  procedure setQ(neu:TPoint);
  function getQ:TPoint;
  procedure konstruieren;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP,ixQ,iyQ:integer);
  procedure zeichnen;
  procedure loeschen;
  procedure verschieben(dx,dy:integer);
  property Anfangspunkt:TPoint read getP write setP;
  property Endpunkt:TPoint read getQ write setQ;
  function Laenge:real;
end;
```

Ändert man bei `TFigur` die Koordinaten `xM`, `yM` in `xP`, `yP` ab, wobei diese bei `TFigur` nach wie vor den Mittelpunkt und bei `TStrecke` den Anfangspunkt beschreiben, so findet man folgende, bereits unter `TGeoObject` zusammengefaßte Gemeinsamkeiten:

```
TGeoObject = class(TObject)
protected
  Canvas:TCanvas;
  xP,yP:integer ;
  procedure setP(neu:TPoint);
  function getP:TPoint;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP:integer);
end;
```

Beachten Sie, dass der `constructor create` nur die Parameter enthält, die beide Klassen `TStrecke` und `TFigur` gemeinsam haben.

Bei `TStrecke` und `TFigur` bleiben jetzt nur noch die für die jeweilige Klasse spezifischen Eigenschaften und Methoden stehen. `TStrecke` und `TFigur` sind jetzt als Nachkommen von `TGeoObject` deklariert:

```
TFigur=class(TGeoObject)
protected
  d:integer;
  procedure setd(neu:integer);
public
  constructor create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP,id:integer);
  property Mittelpunkt:TPoint read getP write setP;
end;

TStrecke=class(TGeoObject)
protected
  xQ,yQ:integer;
  procedure setQ(neu:TPoint);
  function getQ:TPoint;
  procedure konstruieren;
```

```
public
  constructor create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP,ixQ,iyQ:integer;
  procedure zeichnen;
  procedure loeschen;
  procedure verschieben(dx,dy:integer);
  property Anfangspunkt:TPoint read getP write setP;
  property Endpunkt:TPoint read getQ write setQ;
  function Laenge:real;
end;
```

Auch hier könnte man sich überlegen, ob man die „Properties“ `Mittelpunkt` bzw. `Anfangspunkt` in `TFigur` bzw. `TStrecke` streicht und sie als

```
property Punkt:TPoint read getP write setP;
```

beim Vorfahren `TGeoObject` deklariert, worauf ich aber im Interesse besser verständlicher Quelltexte im Hauptprogramm verzichten würde.

Zum `constructor create`: Auf den ersten Blick erscheint es unsinnig, diesen mit den gemeinsamen Parametern beim Vorfahren `TGeoObject` zu deklarieren, da er bei beiden Nachkommen `TFigur` und `TStrecke` – mit *allen*, also den gemeinsamen sowie den klassenspezifischen Parametern – wieder auftaucht.
Es wird sich aber beim Implementieren (d.h. beim Programmieren) des Konstruktors zeigen, dass man sich mit diesem Verfahren einiges an Schreibarbeit erspart. Außerdem hat dieses Verfahren auch auf Anwendungsebene gewisse Vorteile, die zu erläutern den Rahmen des aktuellen Kapitels sprengen würden.

3.7 Aufgabe: TGeoObject implementieren

Beenden Sie ggf. Lazarus. Kopieren Sie wieder „uFigur.pas“ im selben Projektordner „02GeoObject“. Benennen Sie die Kopie in „uFigur2.pas“ um. Das ist jetzt die alte Version, wo es nur den Vorfahren `TFigur` von `TKreis` und `TQuadrat` gibt.
Öffnen Sie Ihr Lazarus-Projekt und ändern Sie „uFigur“ so ab, dass `TStrecke` und `TFigur` den Vorfahren `TGeoObject` bekommen.
Auch danach sollte Ihr Hauptprogramm, die Ballonfahrt, ohne dass Änderungen nötig wären, noch funktionieren.

Tipp zur Programmierung:

Die Konstruktoren von *TGeoObject* und dem Nachkommen *TFigur* werden im folgenden erläutert:

```
constructor TGeoObject.create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP:integer);
begin
  inherited create; //Aufruf des Konstruktors von TObject
  canvas:=icanvas;
  farbe:=ifarbe;
  xp:=ixp;
  yp:=iyp;
end;
```

Die Anweisung

```
inherited create;
```

ruft den Konstruktor des unmittelbaren Vorfahren, in diesem Fall also von `TObject`, auf. Auf diese Weise erbt `TGeoObject` alle Eigenschaften und Methoden von `TObject` und bekommt damit die „Fähigkeiten“, über die jedes Objekt verfügen muss (z.B. die Methode `free` zur Freigabe von Speicherplatz.)
In den folgenden vier Anweisungen werden den `TGeoObject`-Variablen die Anfangswerte zugewiesen, damit klar ist, in welche Zeichenfläche in welcher Farbe an welche Stelle das Objekt später gezeichnet wird.

```
constructor TFigur.create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP,id:integer);
begin
  inherited create(iCanvas,iFarbe,ixP,iyP);//Aufruf des Konstruktors des direkten Vorfahren
TGeoObject
  d:=max(id,0);
end;
```

Die Anweisung

```
inherited create(iCanvas,iFarbe,ixP,iyP);
```

ruft den Konstruktor des unmittelbaren Vorfahren, in diesem Fall also von `TGeoObject`, auf. Aus diesem Grund tauchen jetzt auch die vier Parameter des Konstruktors von `TGeoObject` auf, und den zugehörigen vier Variablen

werden durch diesen Prozeduraufruf ihre Anfangswerte zugewiesen. Deshalb folgt jetzt auch nur noch eine einzige Anweisung, die der für `TFigur` spezifischen Variablen d ihren Anfangswert zuweist.

Beachten Sie auch folgenden Unterschied: Die Zeile

```
constructor TGeoObject.create(iCanvas:TCanvas;iFarbe:TColor;ixP,iyP:integer);
```

ist der Kopf einer Prozedur und enthält daher auch die vom Doppelpunkt gefolgten Typen der Parameter (`TCanvas`, `TColor`, `integer`).
Hingegen handelt es sich bei der Zeile

```
inherited create(iCanvas,iFarbe,ixP,iyP);
```

um einen Prozeduraufruf, so dass nur die Parameter ohne Typ angegeben werden.

Zum Vergleich:

```
procedure TForm1.wuerfeln(augenzahl:integer);   //Prozedurkopf  mit Typ
...
wuerfeln(6);            //Dagegen Prozeduraufruf ohne Typ
```

Implementieren Sie den Konstruktor der Klasse `TStrecke` ähnlich wie bei `TFigur`.

Die Klassen `TKreis` und `TQuadrat` brauchen keinen eigenen Konstruktor, da sie dieselben Variablen haben wie ihr Vorfahr `TFigur`. Es kommt weder in `TKreis` noch in `TQuadrat` eine neue, für diese Figur spezifische Variable hinzu, deren Anfangswert in einem eigenen Konstruktor festgelegt werden müsste.

3.8 Abstrakte Methoden

Wenn man die Quelltexte der Methoden `TStrecke.zeichnen`, `TQuadrat.zeichnen` und `TKreis.zeichnen` vergleicht, ergeben sich – abgesehen vom Prozedurkopf – keine Unterschiede:

```
procedure TStrecke.zeichnen;
begin
  canvas.Pen.Color:=farbe;
  canvas.Brush.Color:=farbe;
  konstruieren;
end;
```

```
procedure TQuadrat.zeichnen;
begin
  canvas.Pen.Color:=farbe;
  canvas.Brush.Color:=farbe;
  konstruieren;
end;
```

```
procedure TKreis.zeichnen;
begin
  canvas.Pen.Color:=farbe;
  canvas.Brush.Color:=farbe;
  konstruieren;
end;
```

Der einzige Unterschied besteht in der Implementierung der Prozedur `konstruieren`, die bei `TStrecke` mit `moveto` und `lineto` arbeitet, bei `TQuadrat` die `Rectangle`-Anweisung und bei `TKreis` die `Ellipse`-Anweisung verwendet.
Das heißt, allein die Prozedur `konstruieren` ist für das spezifische Aussehen der jeweiligen Figur verantwortlich. Was läge also näher, als nur diese Prozedur in den einzelnen Klassen `TStrecke`, `TKreis` und `TQuadrat` zu implementieren, und die Methode `zeichnen`, die in allen drei Klassen gleich aussieht, in den gemeinsamen Vorfahren `TGeoObject` zu verschieben?
Die Schwierigkeit dabei besteht darin, dass ein Quelltext der Form

```
procedure TGeoObj.zeichnen;
begin
  canvas.Pen.Color:=farbe;
  canvas.Brush.Color:=farbe;
  konstruieren;
end;
```

nicht möglich ist, wenn in `TGeoObject` nicht auch die Prozedur `konstruieren` bekannt ist.
Wie soll man sie dort aber deklarieren und implementieren, wo noch gar nicht klar ist, was für eine Figur konstruiert werden soll?
Die Antwort lautet, dass die Prozedur `konstruieren` zwar unter `TGeoObject` deklariert, aber erst bei den abgeleiteten Klassen `TStrecke`, `TQuadrat` und `TKreis` auf ihre jeweilige spezifische Weise implementiert wird. Die in `TGeoObj` zwar deklarierte, aber nicht implementierte Prozedur `konstruieren` ist damit eine typisches Beispiel für eine abstrakte Methode.

Definition:
Methoden, die in einer Klasse zwar deklariert und aufgerufen, aber erst in davon abgeleiteten Klassen (d.h. Nachkommen) implementiert werden, bezeichnet man als **abstrakte Methoden**.

3.8.1 Deklaration abstrakter Methoden

```
type
TGeoObj=class(TObject)
protected
  Canvas:TCanvas;
  xP,yP:integer ;
  procedure setP(neu:TPoint);
  function getP:TPoint;
  procedure konstruieren;virtual;abstract;
public
  farbe:TColor;
  constructor create(iCanvas:TCanvas;...);
  procedure zeichnen;
  procedure loeschen;
end;

TStrecke = class(TGeoObj);
  ...
  procedure konstruieren; override;
  ...
end;

TKreis = class(TGeoObj);
  ...
  procedure konstruieren; override;
  ...
```

Das Schlüsselwort **`abstract`** zeigt an, dass die Methode erst später, in einer abgeleiteten Klasse, implementiert wird.
Das Schlüsselwort **`virtual`** muss immer **vorangestellt** werden. Es besagt, dass die Methode in der abgeleiteten Klasse überschrieben werden wird. Zwar wird sie hier noch gar nicht implementiert, aber immerhin die Deklaration wird überschrieben.

Damit können auch die Prozeduren `zeichnen` und `loeschen` in den drei Nachkommen gelöscht und bereits hier deklariert und implementiert werden.

Zur obigen „Ankündigung“ **`virtual`** (Diese Prozedur wird später überschrieben) gehört der Partner, das Schlüsselwort **`override`** bei der Deklaration in der abgeleiteten Klasse, wo die Prozedur nun tatsächlich überschrieben wird.

3.8.2 Implementierung abstrakter Methoden

```
procedure TGeoObj.zeichnen;
begin
  canvas.Pen.Color:=farbe;
  canvas.Brush.Color:=farbe;
  konstruieren;
end;
```

Bei diesem Prozeduraufruf wird je nach Situation `TStrecke.konstruieren` oder `TKreis.konstruieren` verwendet.

```
procedure TStrecke.konstruieren;
begin
  canvas.moveto(xP,yP);
  canvas.lineto(xQ,yQ);
end;
```

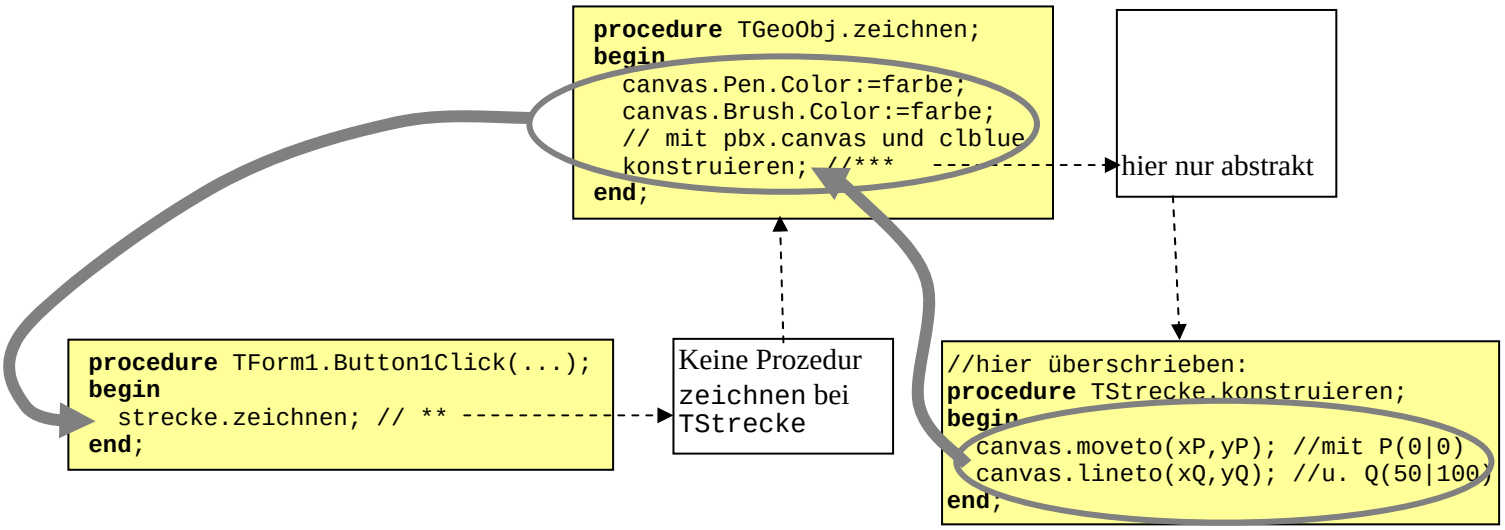
```
procedure TKreis.konstruieren;
begin
  canvas.Ellipse(xP-r,yP-r,xP+r,yP+r);
end;
```

3.8.3 Was bewirkt ein Aufruf im Hauptprogramm?

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  strecke:=TStrecke.create(pbx.canvas,clblue,0,0,50,100); // *
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  strecke.zeichnen; // **
end;
```

In der mit * gekennzeichneten Anweisung wird der Speicherplatz für eine ins Canvas einer Paintbox-Komponente `pbx` zu zeichnende, blaue Strecke mit dem Anfangspunkt P(0|0) und dem Endpunkt Q(50|100) bereitgestellt.
Da das Objekt `strecke` eine Instanz der Klasse `TStrecke` ist, prüft das System in **, ob die Klasse `TStrecke` über eine Methode `zeichnen` verfügt. Da dies nicht der Fall ist, wird als nächstes der Vorfahr `TGeoObject` untersucht, wo sich eine Methode `zeichnen` findet.
Diese ruft die abstrakte Methode `konstruieren` (***) auf. Also wird jetzt überprüft, ob die abstrakte Prozedur `konstruieren` in der von `TGeoObject` abgeleiteten Klasse `TStrecke` überschrieben wird. Dies ist der Fall, also richtet sich der Prozeduraufruf *** genau an die richtige, für eine Strecke spezifische Prozedur `konstruieren`.



Aufgabe 3.9:

Beenden Sie ggf. Lazarus. Kopieren Sie die Datei „uFigur.pas“ ein weiteres Mal und benennen Sie die Kopie in „uFigur3.pas“ um. Dies ist nunmehr die Version mit dem neuen Vorfahren `TGeoObject`, aber noch ohne abstrakte Methoden. Strukturieren Sie die Unit „uFigur“ nochmals um. Führen Sie dabei die abstrakte Methode `konstruieren` ein. Löschen Sie die Prozeduren `zeichnen` und `loeschen` aus den drei Nachkommen `TStrecke`, `TKreis` und `TQuadrat` und verschieben Sie sie in den Vorfahren `TGeoObject`.

Überlegen Sie selbst, wie Sie mit der Prozedur `verschieben` verfahren können.

Ihr Hauptprogramm (Ballonfahrt) müsste auch mit dieser Unit „uFigur“ noch immer funktionieren.