# Mastermind

objektorientiert, mit Delphi

Patrick Breuer

November 2006

## Inhaltsverzeichnis

# 1 Spielbeschreibung

Ein Spieler spielt gegen den Computer. Der Computer wählt zufällig eine Folge von vier farbigen Steckern aus sechs möglichen Farben. Dabei dürfen mehrere Stecker dieselbe Farbe haben. Der Spieler hat die Aufgabe, diese Steckerfolge in maximal zwölf Versuchen zu erraten. Jeder Versuch wird durch den Computer mit Hilfe einer Folge von maximal vier schwarzen oder weißen Steckern auf folgende Weise bewertet:

- Zunächst wird für jeden Stecker, der mit der richtigen Farbe an die richtige Stelle gesetzt ist, ein schwarzer Stecker gesetzt.
- Dann wird für jeden Stecker, dem ein gleichfarbiger Stecker aus der verbleibenden Zufallsfolge entspricht, ein weißer Stecker gesetzt.
- Die schwarzen und weißen Stecker werden stets von links nach rechts gesetzt, so dass nicht erkennbar ist, welche Farbstecker richtig gesetzt sind bzw. welche eine richtige Farbe haben.

# 2 Die Benutzeroberfläche (GUI)

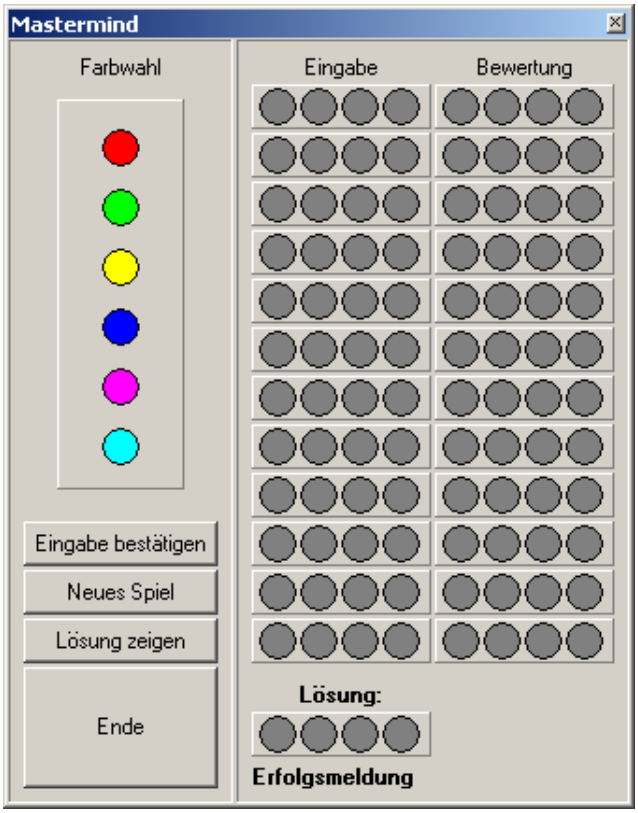
Die Benutzeroberfläche enthält in der linken Hälfte ein „Farbpanel“, in dem der Spieler die Farbe des Steckers auswählt, der als nächstes gesetzt werden soll. Darunter befindet sich die Schaltfläche „Eingabe bestätigen“, durch die der Computer aufgefordert wird, die gesetzte Steckerfolge zu bewerten. Die rechte Hälfte der Benutzeroberfläche ist in zwei Bereiche unterteilt: Links werden vom Spieler Farbstecker gesetzt, rechts setzt der Computer die Bewertungsstecker. Ganz unten wird nach Abschluss eines Spiels die Zufallsfolge angezeigt, darunter eine Meldung über den Spielerfolg.

Ein erster Entwurf der Benutzeroberfläche kann im Entwurfsmodus erstellt werden und wie unten angegeben aussehen.

# 3 Ein erstes Klassendiagramm

Die graphische Benutzeroberfläche dient der Interaktion zwischen Spieler und Computer. Sie wird durch die Klasse *TSpielfeld* in der Unit *UGUI* definiert. Für jede Spielrunde sind insgesamt acht Steckplätze notwendig, die durch *Shapes* realisiert werden, die wiederum auf *Panels* angeordnet sind. Die Klasse *TSteckerpanel* in der Unit *USteckerpanel* definiert später eine Folge von vier Steckplätzen. Sie wird für den ersten GUI–Entwurf noch nicht benötigt.

Der Spielablauf wird durch die Klasse *TSpielverwaltung* in der Unit *USpielverwaltung* definiert. Zur Speicherung der vom Spieler festgelegten Steckerfolge und zur anschließenden Auswertung des Spielzuges dient die



Benutzeroberfläche

Klasse *TSteckerfolge* in der Unit *USteckerfolge*. Schließlich wird ein einzelner Stecker durch die Klasse *TStecker* in der Unit *UStecker* definiert.

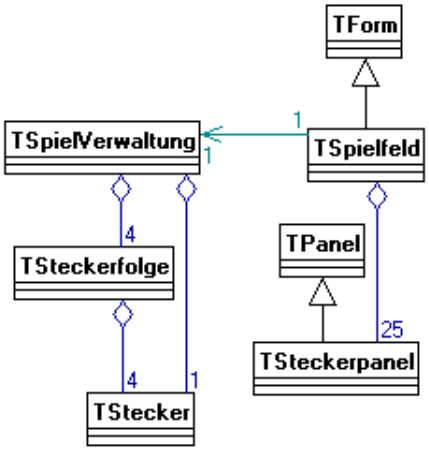
# 4 GUI mit Komponenten–Arrays

Werden alle Steckerpanels und alle Shapes für die Steckplätze im Entwurfsmodus auf dem Formblatt angeordnet, kann nur über ihren jeweiligen Namen — einen „normalen“ Variablenbezeichner — auf sie zugegriffen werden. Dies verhindert die Verwendung der Namen in Zählschleifen und erschwert dadurch die Spielsteuerung. Günstiger ist es, ein Feld von Steckerpanels zu deklarieren und auf den einzelnen Panels die Steckplätze als Feld von Shapes zu realisieren.

Dazu sind folgende Anpassungen notwendig:

- Im Interface–Abschnitt der Unit *UGUI* wird die Klasse *TSpielfeld* um die Deklaration eines zweidimensionalen Feldes *Steckerpanel* erweitert:

```
Steckerpanel: array[1..13,1..2] of TSteckerpanel;
```



Ein erstes Klassendiagramm

Der erste Index steht für die Zeile, der zweite für die Spalte. Steckerpanel[13,1] soll später zur Ausgabe der Lösung dienen, Steckerpanel[13,2] wird nicht benötigt.

- Im Interface–Abschnitt der Unit *USteckerpanel* wird die Klasse *TSteckerpanel* um die Deklaration eines Konstruktors erweitert. Die Parameterliste soll gegenüber dem Konstruktor der Klasse *TPanel* auch Variablen für die Position und die Größe des zu erzeugenden Panels enthalten. Eine weitere Abweichung gegenüber *TPanel* betrifft den Parameter, der den Besitzer des Objekts angibt. Der Besitzer ist das übergeordnete Objekt. Werden alle Steckerpanels auf einem Panel *pnSpielbrett* angeordnet, ist der Besitzer also selbst ein Objekt der Klasse *TPanel* (im Gegensatz zur allgemeineren Klasse *TComponent* im Konstruktor der Klasse *TPanel*).

```
type TSteckerpanel=class(TPanel)
  constructor create (Besitzer:TPanel; x,y,b,h:integer);
end;
```

- Im Implementation–Abschnitt wird der Konstruktor als Methode aufgeschrieben. Die eigentliche Erzeugung des Objekts übernimmt der Konstruktor der „Elternklasse“, der mit dem Schlüsselwort *inherited* aufgerufen wird. Neben den Eigenschaften *Left*, *Top*, *Height* und *Width* muss auch die Eigenschaft *Parent* mit einem Wert belegt werden. Sie gibt an, auf welcher Komponente des Formblatts das Panel erscheinen soll, und erst durch diese Festlegung wird das Panel zur Laufzeit sichtbar.

```
constructor TSteckerpanel.create (Besitzer:TPanel;
                                  x,y,b,h:integer);
begin
  inherited create (Besitzer);
  Left:=x;
  Top:=y;
  Width:=b;
  Height:=h;
  Parent:=Besitzer;
end;
```

- Im Implementation–Abschnitt der Unit *UGUI* wird durch Doppelklick auf den Hintergrund des Formblatts die Methode *FormCreate* ergänzt (und gleichzeitig im Interface–Abschnitt deklariert und als Ereignisbehandlungsroutine des Ereignisses *OnCreate* festgelegt).

  In ihr muss für jedes zu erzeugende Steckerpanel der Konstruktor *create* mit entsprechenden Parametern aufgerufen werden. Dies geschieht durch eine geschachtelte Zählschleife, in der die Wertzuweisungen an $X$ und $Y$ in Abhängigkeit von *Zeile* und *Spalte* noch zu ergänzen sind. Dabei ist zu beachten, dass sich diese Koordinaten nicht auf das Formblatt beziehen, sondern auf die Komponente, auf der die Panels erscheinen, hier also auf das Panel *pnSpielbrett*. Der Ursprung des „Koordinatensystems“ liegt links oben.

```
procedure TSpielfeld.FormCreate(Sender: TObject);
var Zeile,Spalte,X,Y: integer;
begin
  for Zeile:=1 to 12 do
    for Spalte:=1 to 2 do
    begin
      X:=...; Y:=...;
      Steckerpanel[Zeile,Spalte]:=TSteckerpanel.create
                                  (pnSpielbrett,X,Y,96,24);
    end;
    Steckerpanel[13,1]:=TSteckerpanel.create(pnSpielbrett,
                                  ...,...,96,24);
end;
```

- Nun ist die Deklaration der Klasse *TSteckerpanel* so zu erweitern, dass jede Instanz vier Shapes enthält, die die Kreise darstellen. Um später mittels Zählschleife auf sie zugreifen zu können, werden sie wieder als Array realisiert. Das Zeichnen der Kreise soll eine Methode *zeichneKreis* übernehmen, die als Parameter die Farbe und die Stelle (zwischen 1 und 4) des zu zeichnenden Kreises benötigt.

```
Kreis: array[1..4] of TShape;
procedure zeichneKreis (Farbe:TColor; Stelle:integer);
```

- Im Implementation–Abschnitt der Unit *USteckerpanel* wird die Methode *zeichneKreis* notiert. Die Eigenschaft *Shape* bestimmt die Form, *Brush.Color* die Farbe. In der letzten Wertzuweisung wird der reservierte Bezeichner *Self* verwendet. *Self* bezeichnet immer dasjenige Objekt, für das eine Methode bearbeitet wird, hier also das Panel, das gerade mit Kreisen bestückt wird. Für die Eigenschaften *Left*, *Top*, *Width* und *Height* müssen noch geeignete Werte eingesetzt werden (Die Position bezieht sich wieder auf die linke obere Ecke des Panels, auf dem der Kreis gezeichnet wird, und hängt von der Stelle ab).

```
procedure TSteckerpanel.zeichneKreis (Farbe:TColor;
                                      Stelle:integer);
begin
  with Kreis[Stelle] do
  begin
    Shape:=stCircle;
    Brush.Color:=Farbe;
    Left:=...; Top:=...; Width:=...; Height:=...;
    Parent:=Self;
  end;
end;
```

- Als letztes wird der Konstruktor der Klasse *TSteckerpanel* so ergänzt, dass die vier Kreise erst als Objekte erzeugt und dann grau gezeichnet werden:

```
for Stelle:=1 to 4 do
begin
  Kreis[Stelle]:=TShape.Create(Self);
  zeichneKreis(clGray,Stelle);
end;
```

Auch die farbigen Kreise zur Farbauswahl im linken Bereich des Spielfeldes werden sinnvollerweise als Array deklariert und erst zur Laufzeit der Anwendung als Objekte erzeugt. Hier erscheint es zunächst naheliegend, ganz analog zur Klasse *TSteckerpanel* mit je vier Kreisen eine neue Klasse *TFarbpanel* mit sechs Kreisen zu deklarieren. Während aber von der Klasse *TSteckerpanel* zur Laufzeit 25 Instanzen mit unterschiedlichen Ausprägungen bezüglich der Position erzeugt werden, würde von dieser neuen Klasse *TFarbpanel* nur eine einzige Instanz benötigt. Deshalb ist es günstiger, das Farbpanel im Entwurfsmodus einzufügen und die Kreise in die Deklaration des Spielfeldes aufzunehmen.

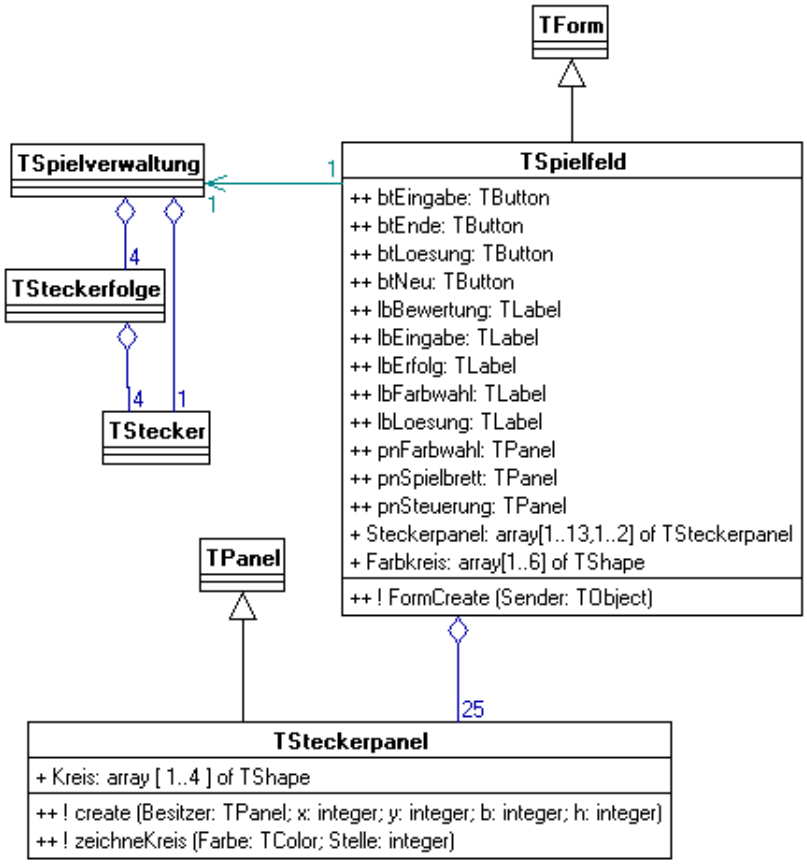
Alle notwendigen Ergänzungen beziehen sich daher auf die Unit *UGUI*.

- Zunächst werden die Kreise als Array deklariert, und zwar innerhalb der Deklaration des Spielfeldes:

```
Farbkreis: array[1..6] of TShape;
```

- Anschließend ist die Methode *FormCreate* zu erweitern. Die Farbkreise müssen durch einen Aufruf des Konstruktors der Klasse *TShape* erzeugt werden und einige ihrer Eigenschaften müssen mit Werten belegt werden. Dem Konstruktor wird dabei das Panel *pnFarbwahl* als Parameter übergeben, der wieder die Eigenschaft *Owner* festlegt. Die Eigenschaft *Top* wird in Abhängigkeit von *Zeile* festgelegt, damit die sechs Kreise untereinander erscheinen. Die Wertzuweisung an *Parent* ist wieder notwendig, damit jedes der Shapes auf dem Panel *pnFarbpanel* sichtbar wird.

```
for Zeile:=1 to 6 do
begin
  Farbkreis[Zeile]:=TShape.Create(pnFarbwahl);
  with Farbkreis[Zeile] do
  begin
    Shape:=stCircle;
    Left:=...; Top:=...; Width:=...; Height:=...;
    case Zeile of
      1: Brush.Color:=clRed;
      2: Brush.Color:=clLime;
      3: Brush.Color:=clYellow;
      4: Brush.Color:=clBlue;
      5: Brush.Color:=clFuchsia;
      6: Brush.Color:=clAqua;
    end;
    Parent:=pnFarbwahl;
  end;
end;
```




# 5 Farbauswahl

Die Farbe des zu setzenden Steckers soll durch Anklicken des entsprechenden Kreises im Farbpanel erfolgen. Im Gegensatz zu Buttons wird beim Anklicken eines Shapes jedoch nicht das Ereignis *OnClick* ausgelöst, sondern das Ereignis *OnMouseDown*. Als Ereignisbehandlungsroutine kann man eine beliebige Prozedur festlegen, für die jedoch die folgenden Parameter vorgeschrieben sind: *Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer*. Tritt das Ereignis *OnMouseDown* ein, werden alle Parameter automatisch mit Werten belegt: *Sender* gibt an, welches Objekt das Ereignis ausgelöst hat; *Button* kann die Werte *mbLeft*, *mbRight* und *mbMiddle* annehmen und gibt an, welche Maustaste gedrückt wurde; *Shift* enthält Informationen über den Zustand bestimmter Tasten der Tastatur und darüber, ob eine Maustaste gedrückt gehalten wird; $X$ und $Y$ geben die Position des Mauszeigers an.

- Zunächst wird eine Prozedur *Farbauswahl* mit entsprechenden Parametern als Methode der Klasse *TSpielfeld* deklariert:

```
procedure Farbauswahl(Sender: TObject; Button: TMouseButton;
                      Shift: TShiftState; X, Y: Integer);
```

- Im Implementation–Abschnitt der Unit *UGUI* wird diese Methode nun in einer ersten Version aufgeschrieben. Beim Anklicken soll der entsprechende Kreis größer dargestellt werden, damit jederzeit erkennbar ist, welche Farbe ausgewählt wurde. Die Größen– und Positionsangaben der anderen Farbkreise werden auf die ursprünglichen Werte zurückgesetzt, damit beim Wechsel der Farbe immer nur ein Kreis vergrößert ist. Für die Fallunterscheidung wird der Parameter *Sender* verwendet.

```
procedure TSpielfeld.Farbauswahl(Sender: TObject;
          Button: TMouseButton; Shift: TShiftState;
          X, Y: Integer);
var Zeile:integer;
begin
  for Zeile:=1 to 6 do
    if Sender=Farbkreis[Zeile] then  //vergrößern
      with Farbkreis[Zeile] do
      begin
        Left:=...;
        Top:=...;
        Width:=...;
        Height:=...;
      end
    else  //alle anderen zurücksetzen
      with Farbkreis[Zeile] do
      begin
        Left:=...;
        Top:=...;
        Width:=...;
        Height:=...;
      end;
end;
```

- Die Methode *Farbauswahl* wird nun als Ereignisbehandlungsroutine für jeden Farbkreis festgelegt. Weil die Farbkreise innerhalb der Methode *FormCreate* des Spielfeldes erzeugt werden, ist dort auch die folgende Zuweisung — an geeigneter Stelle — zu ergänzen:

```
OnMouseDown:=Farbauswahl;
```

Alle bisherigen Anpassungen bezogen sich nur auf die Darstellung der Farbauswahl innerhalb der Benutzeroberfläche. Nun geht es darum, die Farbwahl auch der Spielverwaltung zur Verfügung zu stellen.

- Die Deklaration der Klasse *TSpielverwaltung* (in der Unit *USpielverwaltung*) wird um einen parameterlosen Konstruktor *Create* und um eine Eigenschaft *AktuellerStecker* erweitert. Der aktuelle Stecker ist ein Objekt der Klasse *TStecker*.
- Die Deklaration der Klasse *TStecker* (in der Unit *UStecker*) wird um eine Eigenschaft *Farbe* vom Typ *TColor* erweitert. *TColor* ist ein vordefinierter Typ (in der Unit *Graphics*), der Bezeichner für verschiedene Farbwerte zur Verfügung stellt.
- Im Konstruktor der Spielverwaltung wird der aktuelle Stecker als Objekt der Klasse *TStecker* erzeugt.
- In der Methode *FormCreate* des Spielfeldes wird die Spielverwaltung als Objekt der Klasse *TSpielverwaltung* erzeugt.

Nach diesen Vorarbeiten ist es nun möglich, die Methode *Farbauswahl* so zu erweitern, dass die Farbe des angeklickten Kreises an die Eigenschaft *Farbe* des aktuellen Steckers übergeben wird:

```
for Zeile:=1 to 6 do
  if Sender=Farbkreis[Zeile] then
  begin
    with Farbkreis[Zeile] do
    begin
      ...
    end;
    Spielverwaltung.AktuellerStecker.Farbe:=
                                  Farbkreis[Zeile].Brush.Color;
  end
  else
    with Farbkreis[Zeile] do
    begin
      ...
    end;
```

# 6 Positionswahl

Als nächstes geht es darum, das Setzen eines Steckers zu ermöglichen. Nach Wahl einer Farbe soll ein Stecker des Spielbretts beim Anklicken die gewählte Farbe annehmen. Dazu wird — wie zuvor bei den Kreisen für die Farbwahl — eine entsprechende Ereignisbehandlungsmethode *Positionswahl* ergänzt und dem Ereignis *OnMouseDown* zugewiesen. Sie zeichnet den angeklickten Kreis in der Farbe des aktuellen Steckers neu.

- Zunächst wird analog zur Methode *Farbauswahl* eine Methode *Positionswahl* der Klasse *TSpielfeld* (mit den gleichen Parametern) deklariert.

- Innerhalb der Methode muss für jede der vier Stellen in jedem Steckerpanel der linken Spalte überprüft werden, ob der entsprechende Kreis das Ereignis ausgelöst hat. Ist dies der Fall, wird der Kreis in der Farbe des aktuellen Steckers der Spielverwaltung neu gezeichnet. Die Variable *AktuelleFarbe* vom Typ *TColor* dient nur dazu, den Aufruf der Methode *zeichneKreis* zu verkürzen.

```
AktuelleFarbe:=Spielverwaltung.AktuellerStecker.Farbe;
for Zeile:=1 to 12 do
  for Stelle:=1 to 4 do
    if Sender=Steckerpanel[Zeile,1].Kreis[Stelle] then
      Steckerpanel[Zeile,1].zeichneKreis(AktuelleFarbe,
                                                Stelle);
```

- Zuletzt muss nun diese Methode *Positionswahl* den Kreisen als Ereignisbehandlungsmethode für das Ereignis *OnMouseDown* zugewiesen werden. Dazu wird die Methode *FormCreate* des Spielfeldes erweitert, denn dort werden die Steckerpanels mit ihren Kreisen erzeugt.

```
for Stelle:=1 to 4 do
  Steckerpanel[Zeile,Spalte].Kreis[Stelle].OnMouseDown:=
                                                Positionswahl;
```

Damit ist die Funktionalität der Benutzeroberfläche weitgehend fertig gestellt. In den folgenden Arbeitsschritten geht es darum, die internen Abläufe, also die Rolle des Computers beim Spielen, zu implementieren.

# 7 Erzeugung der Zufallsfolge

- Zunächst wird die Klasse *TSpielverwaltung* um ein Array *Vorgabe* von vier Farben (*TColor*) erweitert, das die Farbvorgabe für einen Spieldurchlauf darstellt. Um den Datentyp *TColor* verwenden zu können, muss in der *Uses*–Zeile die System–Unit *Graphics* angegeben werden.

- Nun wird die Klasse *TSpielverwaltung* um eine parameterlose Methode *initialisieren* ergänzt, die zufällig vier Farben im Feld Vorgabe speichert. Diese Methode wird später noch für weitere Aufgaben der Spielsteuerung benötigt.

  Als erste Anweisung der Methode wird der Zufallszahlengenerator mit Hilfe des Befehls *randomize* „neu gemischt“. Dann werden durch die

Funktion *random* Zufallszahlen erzeugt, die anschließend den sechs verschiedenen Farben zugeordnet werden. Die Funktion *random* kann sowohl mit als auch ohne Parameter verwendet werden. Ohne Parameter liefert sie eine Zufallszahl $q$ mit $0 \leq q < 1$. Gibt man einen positiven, ganzzahligen Parameter $n$ an, ist auch die Zufallszahl ganzzahlig und es gilt $0 \leq q < n$.

Für die Erzeugung der Zufallszahlen sowie die Zuordnung der Farbwerte an die vier Feldvariablen verwendet man sinnvollerweise eine geeignete Zählschleife und in dieser eine Mehrfach-Fallunterscheidung mit *case*.

```
Farbe:=random(6);
case Farbe of
  0: Vorgabe[Stelle]:=clRed;
  1: Vorgabe[Stelle]:=clLime;
  2: Vorgabe[Stelle]:=clYellow;
  3: Vorgabe[Stelle]:=clBlue;
  4: Vorgabe[Stelle]:=clFuchsia;
  5: Vorgabe[Stelle]:=clAqua;
end;
```

- Zuletzt wird der Konstruktor der Spielverwaltung um einen Aufruf der Methode *initialisieren* erweitert.

Beim Starten der Anwendung wird nun zwar eine zufällige Folge von Farben als Vorgabe für einen Spieldurchlauf festgelegt, man kann aber nichts davon sehen. Um die Richtigkeit des zuvor Programmierten zu überprüfen, ist es an dieser Stelle sinnvoll, vorübergehend eine Ausgabe der Zufallsfarben einzubauen.

- Am Ende der Methode *FormCreate* des Spielfeldes kann man Aufrufe der Methode *zeichneKreis* einfügen, so dass die Zufallsfarben im Lösungspanel angezeigt werden (ebenso in der *OnClick*–Methode des Buttons *btNeu*, wenn die Spielverwaltung zuvor initialisiert wird):

```
for Stelle:=1 to 4 do
  Steckerpanel[13,1].zeichneKreis(
                  Spielverwaltung.Vorgabe[Stelle],Stelle);
```

# 8 Bewertung

Im letzten Abschnitt wurde *Vorgabe* als Array vom Typ *TColor* definiert. Das ist eine angemessene Möglichkeit, wenn es nur darum geht, Zufallsfarben zu speichern. Im Modell des Spiels handelt es sich aber um eine Folge von vier

Steckern, die in jeder Spielrunde mit der getippten Steckerfolge verglichen wird. Daher ist es sinnvoll, die Klasse *TSteckerfolge* so zu definieren, dass sie vier Stecker enthält, und dann *Vorgabe* als Objekt der Klasse *TSteckerfolge* zu deklarieren.

- ```
  type TSteckerfolge=class
    private
    public
      Stecker:array[1..4] of TStecker;
      constructor create;
  end;
  ```

- Der Konstruktor muss dafür sorgen, dass beim Erzeugen eines Objekts der Klasse *TSteckerfolge* auch die vier enthaltenen Stecker erzeugt werden:

  ```
  constructor TSteckerfolge.create;
  var Stelle:integer;
  begin
    for Stelle:=1 to 4 do
      Stecker[Stelle]:=TStecker.create;
  end;
  ```

Nach dieser Ergänzung ist jetzt es jetzt möglich, die Deklaration von *Vorgabe* in der Klasse *TSpielverwaltung* entsprechend anzupassen:

```
Vorgabe: TSteckerfolge;
```

Dies macht nun weitere Anpassungen notwendig:

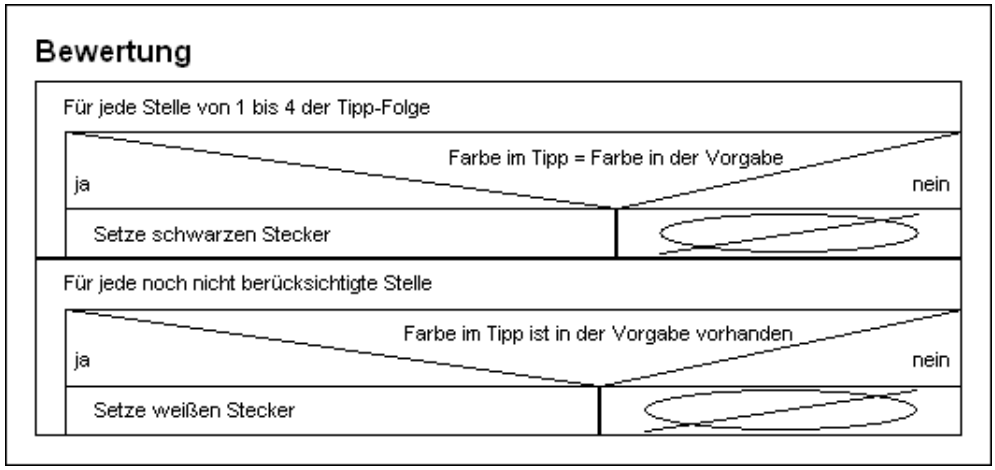
- Im Konstruktor der Spielverwaltung muss *Vorgabe* als Instanz der Klasse *TSteckerfolge* erzeugt werden.

- In der Methode *initialisieren* müssen die Wertzuweisungen angepasst werden. Innerhalb der *Case*–Anweisung wird nun der Eigenschaft *Farbe* des jeweiligen Steckers der Steckerfolge *Vorgabe* ein Farbwert zugewiesen.

- Wurde zum Testen eine Anzeige der Zufallsfarben vorgesehen, muss auch die Anweisung zum Zeichnen der Kreise in den Methoden *FormCreate* und *btNeuClick* des Spielfeldes angepasst werden. Der zu übergebende Parameter für die Farbe ist jetzt die Eigenschaft *Farbe* des jeweiligen Steckers der Steckerfolge *Vorgabe*.

Nach diesen Änderungen sollte die Anwendung wieder wie vorher „laufen“. Nun gilt es, den Algorithmus für die Bewertung der getippten Steckerfolge zu entwerfen.
```

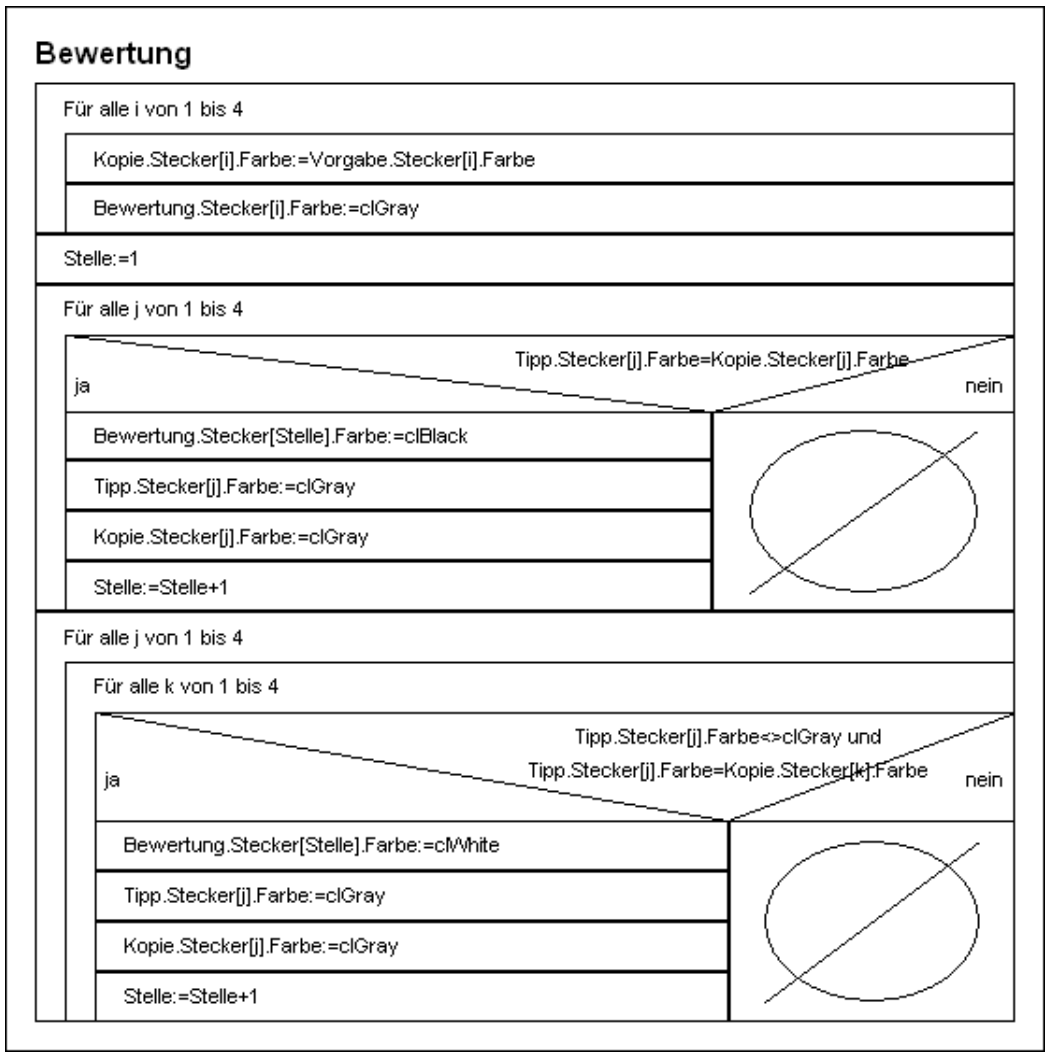
Damit die Position der schwarzen und weißen Stecker keine Rückschlüsse auf die durch sie bewerteten Farbstecker zulässt, werden immer zuerst die schwarzen und dann die weißen Stecker gesetzt. Ein erster allgemein formulierter Algorithmus sieht demnach folgendermaßen aus:




Während die erste Schleife leicht zu programmieren ist, ergibt sich in der zweiten das Problem, dass nur noch diejenigen Farbstecker berücksichtigt werden dürfen, die zuvor noch nicht mit einem schwarzen Stecker bewertet wurden. Mit einem Trick lässt sich das Problem lösen: Sobald ein schwarzer oder weißer Stecker gesetzt wurde, löscht man in der getippten und in der vorgegebenen Steckerfolge die Farben der jeweiligen Stecker, indem man die Eigenschaft *Farbe* z. B. auf *clGray* setzt. Für die weitere Bewertung sind dann nur noch diejenigen Stecker relevant, die nicht grau sind.

Da die vorgegebene Steckerfolge in allen Spielrunden zur Verfügung stehen muss, ist es sinnvoll, bei der Bewertung mit einer Kopie der Folge zu arbeiten. Man benötigt also drei weitere Steckerfolgen: *Tipp* enthält die Farben der vom Spieler gesetzten Stecker, *Kopie* entspricht der vom Computer vorgegebenen Steckerfolge, und *Bewertung* enthält die schwarzen und weißen Stecker, die anschließend im rechten Steckerpanel des Spielfeldes angezeigt werden.

Dies ergibt die folgende Präzisierung des Algorithmus, bei der vorausgesetzt wird, dass *Tipp*, *Kopie* und *Bewertung* analog zu *Vorgabe* als Objekte der Klasse *TSteckerfolge* deklariert sind.




Dieser Algorithmus wird in einer neu zu erstellenden Methode *bewerten* der Spielverwaltung implementiert. Die Methode wird aufgerufen von der Methode *btEingabeClick* des Spielfeldes, in der auch für die Speicherung der getippten Farben und die Ausgabe der Bewertungsfolge gesorgt wird. Bis die Spielsteuerung fertig gestellt ist, erfolgt die Auswertung und Bewertung nur in der ersten Zeile des Spielbretts.

```
for Stelle:=1 to 4 do
  Spielverwaltung.Tipp.Stecker[Stelle].Farbe:=
                    Steckerpanel[1,1].Kreis[Stelle].Brush.Color;
Spielverwaltung.bewerten;
for Stelle:=1 to 4 do
  Steckerpanel[1,2].Kreis[Stelle].Brush.Color:=
                Spielverwaltung.Bewertung.Stecker[Stelle].Farbe;
```

# 9 Spielsteuerung

Wenn die Bewertung richtig funktioniert, kann die Anzeige der vorgegebenen Farben im Lösungspanel wieder deaktiviert werden.

Die Spielverwaltung muss nun zunächst so erweitert werden, dass ein Rundenzähler steuert, in welcher Zeile des Spielbrettes die Stecker bewertet werden. Dann wird die Methode *bewerten* angepasst, so dass in jeder Runde entschieden wird, ob die vorgegebene Steckerfolge erraten wurde oder das Spiel nach zwölf Versuchen erfolglos beendet ist.

- Als erstes wird in der Klasse *TSpielverwaltung* eine Eigenschaft *Runde* ergänzt. Sie erhält in der Methode *initialisieren* den Wert 1. Auf diese Weise wird gleichzeitig erreicht, dass der Rundenzähler bei jedem Neustart des Spiels (über den Button *btNeu*, dessen Klick–Methode die Methode *initialisieren* der Spielverwaltung aufruft) zurückgesetzt wird.

- Nun ist die Methode *btEingabeClick* anzupassen. Beim Speichern der getippten Farben und beim Anzeigen der Bewertung muss auf die der aktuellen Runde entsprechenden Steckerpanels zugegriffen werden. Außerdem muss der Rundenzähler um 1 erhöht werden, sobald die Bewertung angezeigt ist.

- Am Ende der Methode *btEingabeClick* soll auch gegebenenfalls die Anzeige der vorgegebenen Farben erfolgen, und zwar dann, wenn die richtige Farbfolge erraten wurde oder wenn die zwölfte Runde erfolglos beendet ist. Dazu wird die Klasse *TSpielverwaltung* um eine Eigenschaft *geloest* vom Typ *boolean* erweitert. Variablen dieses Typs können die beiden Werte *true* und *false* annehmen. Die Eigenschaft erhält in der Methode *initialisieren* den Wert *false*. In der Methode *bewerten* wird ihr Wert auf *true* gesetzt, wenn alle vier Bewertungsstecker schwarz sind.

```
Summe:=0;
for i:=1 to 4 do
  if Bewertung.Stecker[i].Farbe=clBlack then
    Summe:=Summe+1;
if Summe=4 then
  geloest:=true;
```

- Nun wäre es möglich, die Methode *btEingabeClick* um Anweisungen zu erweitern, die für die Ausgabe der vorgegebenen Farbfolge und einer Erfolgsmeldung sorgen, falls eine der oben genannten Bedingungen erfüllt ist. Die notwendigen Anweisungen sind jedoch dieselben, die

auch die Klick–Methode des Buttons *btLoesung* enthalten muss. Deshalb ist es geschickter, nur einen Aufruf der Methode *btLoesungClick* zu ergänzen.

- Innerhalb der Methode *btLoesungClick* kann jetzt neben der Anzeige der vorgegebenen Farbfolge auch eine Information über den Spielerfolg ergänzt werden, die von den Werten der Eigenschaften *Runde* und *geloest* der Spielverwaltung abhängig ist. Für diese Ausgabe war das Label *lbErfolg* vorgesehen.

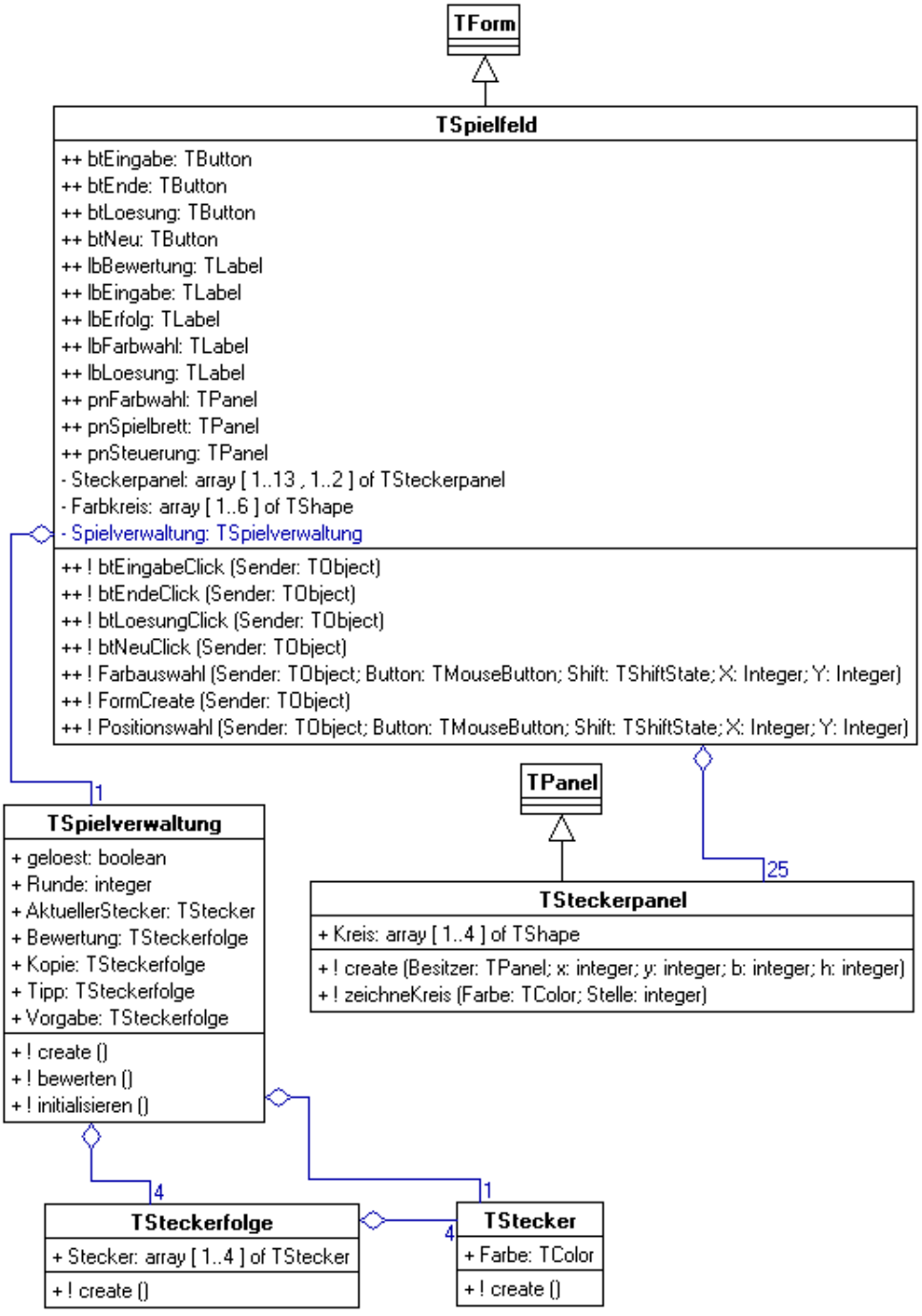
Zuletzt ist noch die Klick–Methode des Buttons *btNeu* zu vervollständigen.

- Die Methode *btNeuClick* enthält als erste Anweisung einen Aufruf der Methode *initialisieren* der Spielverwaltung. Dadurch werden die Werte der Eigenschaften *Runde* und *geloest* zurückgesetzt und eine neue Zufallsfolge von Farben erzeugt.
- Anschließend müssen noch alle Kreise auf dem Spielbrett neu gezeichnet werden und die Erfolgsmeldung muss wieder gelöscht werden.

## 10 Feinheiten

Das Spiel *Mastermind* ist nun der Vorlage weitgehend entsprechend modelliert und „spielbar“. Es gibt aber noch — mehr oder weniger sinnvolle — Möglichkeiten, die Anwendung weiter zu entwickeln:

- Verhinderung von Bedienungsfehlern
  - Sperrung nicht benötigter Steckerpanels bei der Positionswahl
  - Deaktivierung des Buttons *btEingabe*, solange nicht vier Stecker gesetzt sind
- Klick auf bereits gesetzte Steckerposition löscht den Stecker (färbt ihn wieder grau)
- Entfernung der (unschönen) Markierung eines Buttons nach dem Anklicken
- Positionswahl nur mit linker Maustaste; Anzeige eines Farbauswahl–Menüs bei Betätigung der rechten Maustaste über einer Steckerposition
- Für Experten: Rollentausch — Der Computer rät eine Farbfolge; der Benutzer bewertet mit schwarzen und weißen Steckern


TForm
TSpielfeld
++ btEingabe: TButton
++ btEnde: TButton
++ btLoesung: TButton
++ btNeu: TButton
++ lbBewertung: TLabel
++ lbEingabe: TLabel
++ lbErfolg: TLabel
++ lbFarbwahl: TLabel
++ lbLoesung: TLabel
++ pnFarbwahl: TPanel
++ pnSpielbrett: TPanel
++ pnSteuerung: TPanel
- Steckerpanel: array [ 1..13 , 1..2 ] of TSteckerpanel
- Farbkreis: array [ 1..6 ] of TShape
- Spielverwaltung: TSpielverwaltung
++ ! btEingabeClick (Sender: TObject)
++ ! btEndeClick (Sender: TObject)
++ ! btLoesungClick (Sender: TObject)
++ ! btNeuClick (Sender: TObject)
++ ! Farbauswahl (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X: Integer; Y: Integer)
++ ! FormCreate (Sender: TObject)
++ ! Positionswahl (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X: Integer; Y: Integer)
1
25
TPanel
TSpielverwaltung
+ geloest: boolean
+ Runde: integer
+ AktuellerStecker: TStecker
+ Bewertung: TSteckerfolge
+ Kopie: TSteckerfolge
+ Tipp: TSteckerfolge
+ Vorgabe: TSteckerfolge
+ ! create ()
+ ! bewerten ()
+ ! initialisieren ()
TSteckerpanel
+ Kreis: array [ 1..4 ] of TShape
+ ! create (Besitzer: TPanel; x: integer; y: integer; b: integer; h: integer)
+ ! zeichneKreis (Farbe: TColor; Stelle: integer)
4
1
TSteckerfolge
+ Stecker: array [ 1..4 ] of TStecker
+ ! create ()
4
TStecker
+ Farbe: TColor
+ ! create ()