Verwalten von Objekten mit Hilfe von Container-Klassen am Beispiel eines Memory Spieles

Container-Klassen in Delphi

Michael Puff
mail@michael-puff.de

2010-03-26

Inhaltsverzeichnis

1 Vorwort

Oftmals steht man vor dem Problem, dass man eine unbestimmte Anzahl von Objekten einer Klasse irgendwie verwalten muss, zum Beispiel Adressen für eine Adressdatenbank oder Spieler für ein Spiel. Nun kann man sich einen eigenen Datentyp (in Delphi: `record`) entwickeln und diesen dann mit einem dynamischen Array verwalten. Sobald es aber sinnvoll ist, dass die Elemente auch eigene Methoden und Eigenschaften mitbringen; ein Spieler-Element könnte zum Beispiel die Methode `Move` besitzen, die einen Spielzug macht oder ähnliches. In diesem Fall kommt man mit einem eigenen Datentyp nicht mehr weit und man muss auf Objekte zurückgreifen, wenn man OOP konform arbeiten will.

Und wenn man dann schon mit Objekten arbeitet, warum dann nicht gleich den ganzen Schritt machen und statt des dynamischen Arrays eine Liste implementieren, mit der man diese Objekte verwalten kann? Zum einem bleibt man damit OOP konform und zum anderen hat man dann noch die Möglichkeit Methoden in der Liste zu implementieren, um alle oder einzelne Objekte innerhalb der Liste zu manipulieren. Nehmen wir als konkretes Beispiel mal ein einfaches Spiel, was jeder kennt: *Memory*. Ein Memory-Spiel besteht aus Karten, von denen jede doppelt vorhanden ist und einem Spielfeld, meist ein Tisch oder eine beliebige andere ebene Fläche, auf der die Karten verdeckt, meist im Quadrat, angeordnet werden. Ziel ist es nun Kartenpaare zu finden und aufzudecken. Man deckt ein Kartenpaar auf und wenn es übereinstimmt, werden die Karten vom Spielfeld genommen. Stimmen sie nicht über ein, werden sie wieder umgedreht. Soweit kurz zu den Regeln. Gucken wir uns nun ein mögliches Lösungskonzept an. Wir haben die Spielkarten und wir haben das Spielfeld. Wollte man dies nun mit eigenen Datentypen und dynamischen Arrays lösen, wird die Implementierung recht umständlich, wie wir gleich sehen werden, wenn wir die OOP konforme Lösung betrachten.

2 OOP konforme Memory-Spiel Lösung mit Listen

An Hand des im Vorwort angeführten Beispiels eines einfachen Memory-Spiels, will ich hier eine mögliche objektorientierte Lösung aufzeigen.

2.1 Container-Klassen

Delphi stellt drei Klassen zur Verfügung, die als Container für Objekte dienen können:

TList: TList ist eine einfache Listenklasse, die eigentlich nur Array von Zeiger in einer Liste verwaltet und stellt Methoden zur Verfügung, um ein Zeiger der Liste hinzuzufügen, zu entfernen, zu verschieben usw.

TObjectList: TObjectList ist eine Erweiterung der TList-Klasse und besonders für Objekte geeignet, da sie auch den Speicherplatz der Objekte selber verwalten kann.

TCollection: TCollection ist eine spezielle Klasse, um Objekte vom Typ TCollectionItem zu verwalten. Näheres dazu kann m an in der Delphi-Hilfe nachlesen.

Für unsere Spielfeldklasse, die auch gleichzeitig die Container-Klasse für die Karten ist, würde sich die Klasse `TObjectList` anbieten, da sie uns einiges an Arbeit abnehmen kann, was die Speicherverwaltung betrifft. Ich habe mich jedoch für die einfachere Klasse `TList` entschieden, um zu zeigen, was man bei der Speicherverwaltung berücksichtigen muss.

Betrachtet man das Problem, die Implementierung eines Memory-Spiels, von der objektorientierten Seite, ergibt sich eine Lösung des Problems, wie man die Spielregeln umsetzen könnte, fast von alleine. Eine Karte wird in einem Karten-Objekt abgebildet und unser Spielfeld wird unser Container für die Karten. Unserer Karten-Klasse geben wir zwei Eigenschaften mit: *Wert* und *Status*. Der Wert entspricht dem aufgedruckten Symbol und der Status, ob die Karte auf- oder zugedeckt ist. Dies könnte man auch noch mit einem Record lösen. Aber es wäre bestimmt praktisch, wenn die Klasse, die die Karten verwaltet eine Benachrichtigung bekäme, wenn eine Karte umgedreht wird, um entsprechend darauf reagieren zu können. Also geben wir unserer Karten-Klasse noch ein Ereignis `OnFlip` mit, welches ausgelöst wird, wenn sich der Status der Karte ändert, sie also umgedreht wird. Die Container-Klasse, unser Spielfeld kann dann entscheiden, was passieren soll: Ist es die erste Karte, darf noch eine aufgedeckt werden, ist es die zweite, müssen die Werte der Karten verglichen werden um dann zu entscheiden, was weiter passieren muss. Daraus ergibt sich eigentlich schon welche Methoden, Ereignisse und Eigenschaften die Container-Klasse besitzen muss. Unser Spielfeld, muss im Prinzip nur eine Karte umdrehen können und ein Ereignis auslösen,

wenn ein Zug zu ende ist, um den „Schiedsrichter“ benachrichtigen zu können, der dann entscheidet, was weiter passieren soll. Ich habe der Container-Klasse für unser Spielfeld noch ein paar zusätzliche Methoden gegönnt, um bestimmte Aufgaben etwas zu vereinfachen. Als da wären zum Beispiel alle Karten aufdecken oder wieder ein ganzes Kartenpaar umzudrehen, wenn sie nicht gleich waren.

2.2 Die Container-Klasse TMemoryField

Wir leiten unsere Spielfeldklasse nicht direkt von der Klasse `TList` ab, sondern einfach nur von der Basisklasse für alle Klassen `TObject`. Zum einem, weil wir noch ein paar zusätzliche Methoden implementieren wollen und zum anderen weil wir so die Möglichkeit haben selber zu bestimmen, welche Methoden mit welchen parameteren nach aussen hin sichtbar sind. So können wir sicher stellen, dass man in der Liste nur Objekte unserer Kartenklasse ablegen kann.

Dazu implementieren wir eine Klasse mit einer „inneren“ Liste (`FCards: TList`), welche dann schließlich unsere Karten-Objekte aufnimmt. Da wir letztendlich nur Karten der Liste hinzufügen können müssen, reicht es wenn wir eine nicht öffentliche, nach aussen nicht sichtbare Methode `Add(Card: TCard)` implementieren, die eine Karte unserer inneren Liste `FCards` hinzufügt. Nicht sichtbar deswegen, weil das Spielfeld und die Karten ja von der Klasse selber aufgebaut und erzeugt werden. Fehlt nur noch eine Eigenschaft, um auf die Objekte in unserer Liste zugreifen zu können. Die übliche Bezeichnung für diese Eigenschaft ist `items` mit der dazugehörigen Getter-und Setter-Methode. Somit wäre unser Container eigentlich schon fast fertig implementiert. Fehlt nur noch eins, das Freigeben der Objekte in der Liste, wenn unsere Spielfeld-Klasse selber freigegeben wird. Somit sähe unsere Spielfeld-Klasse bisher wie folgt aus:

```
TMemoryField = class(TObject)
private
  FCards: TList;
  // ...;
  // ...;
  function GetItem(Index: Integer): TCard;
  procedure SetItem(Index: Integer; const Value: TCard);
  procedure Add(Card: TCard);
  // ...;
  // ...;
public
  constructor Create(CountCards: Integer; Parent: TWinControl; Width:
     Integer; Padding: Integer);
  destructor Destroy; override;
  property Items[Index: Integer]: TCard read GetItem write SetItem;
  // ...;
  // ...;
end;
```

Und die zu gehörigen implementierten Methoden:

```
constructor TMemoryField.Create(CountCards: Integer; Parent: TWinControl;
    Width: Integer; Padding: Integer);
begin
  inherited Create;
  // ...;
  FCards := TList.Create;
  // ...;
end;

procedure TMemoryField.Add(Card: TCard);
begin
  FCards.Add(Card);
end;

function TMemoryField.GetItem(Index: Integer): TCard;
begin
  Result := FCards.Items[Index];
end;

procedure TMemoryField.SetItem(Index: Integer; const Value: TCard);
begin
  FCards.Items[Index] := Value;
end;

function TMemoryField.GetCount: Integer;
begin
  Result := FCards.Count;
end;

destructor TMemoryField.Destroy;
var
  i                 : Integer;
begin
  if FCards.Count > 0 then
  begin
    for i := FCards.Count - 1 downto 0 do
    begin
      TObject(FCards.Items[i]).Free;
    end;
  end;
  FCards.Free;
  inherited;
end;
```

Im Konstruktor wird unsere innere Klasse erzeugt: `FCards := TList.Create;` und im Destruktor, nach Freigabe aller enthaltenen Objekte:

```
for i := FCards.Count - 1 downto 0 do
begin
  TObject(FCards.Items[i]).Free;
end;
```

wieder freigegeben: `FCards.Free;`.

Die Setter-Methode `SetItem` nimmt nur ein Objekt vom Typ `TCard` als Parameter an, so dass sicher gestellt ist, dass auch wirklich nur Objekte vom Typ `TCard` in die Liste aufgenommen werden können. Entsprechend gibt die Methode `GetItem` auch nur ein Objekt vom Typ `TCard` zurück.

2.3 Die Spiel-Methoden und zugehörigen Ereignisse

Das zentrale Ereignis ist die „OnClick“-Methode

```
procedure Click(Sender: TObject); reintroduce;
```

der Klasse `TCard`, welche die Klick-Methode der Vorfahrenklasse `TPanel` implementiert. In dieser Methode wird einfach nur der entsprechende Status gesetzt, nämlich „aufgedeckt“ und das Ereignis `OnFlip` ausgelöst, auf welches die Container-Klasse dann entsprechend reagiert:

```
procedure TMemoryField.OnFlip(Card: TCard);
begin
  Inc(FCountFlips);
  if FCountFlips = 1 then
    FFirstCard := Card
  else
    FSecondCard := Card;
  if FCountFlips = 2 then
  begin
    if Assigned(OnEndTurn) then
      OnEndTurn(FFirstCard.Value = FSecondCard.Value);
    FCountFlips := 0;
  end;
end;
```

Hier wird einfach die Häufigkeit des Ereignisses `OnFlip` gezählt und wenn es zwei mal aufgetreten ist, eine Runde ist zu ende, wird das Ereignis `OnEndTurn` der Klasse `TMemoryField` ausgelöst. Das Ereignis `OnEndTurn` liefert als Parameter auch gleich noch mit, ob die Karten den gleichen Wert haben, also ein Paar bilden, oder nicht:

```
TOnEndTurn = procedure(AreEquale: Boolean) of object;
```

Auf dieses Ereignis wiederum reagiert unsere Schiedsrichter-Klasse, die in diesem Beispiel auch gleich die Klasse unseres Formulares ist:

```
procedure TForm1.OnEndTurn(AreEqual: Boolean);
begin
  Panel1.Enabled := False;
  if AreEqual then
  begin
    if CheckBox1.Checked then
      Field.HideCouples
```

```
    else
      Field.FlipCouples(csFound);
  end
  else
  begin
    Delay(1000);
    Field.FlipCouples(csBlind);
  end;
  if IsGameOver then
    ShowMessage(rsGameOver);
  Panel1.Enabled := True;
end;
```

Mit diesen zwei, drei Methoden und Ereignissen hätten wir dann schon ein einfaches Memory-Spiel ohne großen Aufwand implemntiert. Alle anderen Methoden dienen eigentlich nur dazu den Umgang etwas zu vereinfache bzw. zu erleichtern und um den Spielkomfort zu erhöhen, deswegen will ich an dieser Stelle nicht weiter auf sie eingehen.

3 Ein Blick über den Tellerrand – Container-Klasse in C#

Abschliessend will ich noch mal etwas über den Tellerand schauen und zum Vergleich aufzeigen, wie man so etwas mit der .NET Sprache C# lösen könnte, wenn man keine vorgefertigte Klasse nutzen will. Im Prinzip kann unsere Container-Klasse identisch aufgebaut sein, nur eben mit der C# typischen Syntax für Eigenschaften:

```
class PersonenListe
    {
        private List<Person> innerList;

        public PersonenListe()
        {
            innerList = new List<Person>();
        }

        public void Add(Person person)
        {
            innerList.Add(person);
        }

        public int Count
        {
            get
            {
                return innerList.Count;
            }
        }

        public Person this[int index]
        {
            get
            {
                return (innerList[index]);
            }
            set
            {
                innerList[index] = value;
            }
        }

        public void RemoveAt(int index)
        {
            innerList.RemoveAt(index);
```

```
        }

        // benötigt für foreach
        public System.Collections.Generic.IEnumerator<Person>
            GetEnumerator()
        {
            return innerList.GetEnumerator();
        }

        public void Sort(IComparer<Person> comparer)
        {
            innerList.Sort(comparer);
        }
    }
```

Auffällig ist hier, wenn man mal von der C# spezifischen Syntax absieht, dass es keinen Destruktor gibt und nirgends etwas frei gegeben wird. Das liegt daran, dass C# eine Grabage collection besitzt, die automatisch dafür sorgt, dass nicht mehr benötigter Speicher wieder freigegeben wird. Insofern fällt unsere Klasse etwas einfacher aus, da man sich nicht mehr um das Freigeben des Speichers kümmern muss. Desweiteren wird eine generische Liste benutzt `innerList = new List<Person>();`, die eigentlich die Wrapper-Klasse überflüssig macht, da die Typensicherheit schon durch die generische Liste gegeben ist. Erst wenn man eine erweiterte Funktionalität implementieren will, zum Beispiel eine Prüfung vor dem Einfügen in der Liste oder ähnliches, würde eine Wrapper-Klasse wieder sinnvoll sein.

Die Klasse besitzt ausser dem noch eine zusätzliche Methoden, die mit der Verwaltung der Liste eigentlich nichts zu tun hat: `GetEnumerator()`. Diese Methode ist nötig damit man mit Hilfe einer `foreach`-Schleife durch die Einträge in der Liste iterieren kann.

Allerdings bietet C# schon extra für solche Probleme vorgefertigte Klassen an. Als da wäre zum Beispiel die Klasse `Collection` für eine benutzerdefinierte Aufliste und darum handelt es sich ja schließlich.

4 Schlussbemerkung

Wie man sieht handelt es sich hier um ein Programmiersprachen unabhängiges Konzept. Man spricht auch von „Design Patter“ oder einem Entwurfsmuster bzw. Lösungsmuster. Siehe dazu den Wikipedia Artikel „Entwurfsmuster“[1]. Hat man dies erst ein paar mal durch exerziert, fällt es auch nicht schwer das Prinzip auf andere Sprachen zu übertragen.

Wie man auch gesehen hat, ergibt sich die Lösung mehr oder weniger schon von alleine beim Entwurf der Klassen. Die Implementierung ist dann meist nur noch Tipparbeit, da man das Problem, wie schon gesagt während des Entwurfs der Klassen und Benennung der Methoden und Ereignisse gelöst hat.

[1] http://de.wikipedia.org/wiki/Entwurfsmuster