Objektorientierte Programmierung

08 - Polymorphie

Alexander Stuckenholz

Version 2022-05-04

Inhalt

HOCHSCHULE HAMM-LIPPSTADT

Wurzelklasse Object

Leitet in C# eine Klasse nicht explizit von einer anderen ab, wird sie implizit von der Basisklasse *Object* abgeleitet.

- Die Klasse *Object* wird durch das .Net-Framework selbst definiert.
- Die Klasse Object ist somit die ultimative Wurzelklasse in C#.

Da alle Klassen implizit von *Object* ableiten, können auch alle Objekte in den Basistyp *Object* konvertiert werden.

```
1  var s = new Student("Hannelore", "Klimcak", 12345);
2  var o = (Object) s;
```

Auf *o* können dann aber nur Methoden aufgerufen werden, die durch die Klasse *Object* auch definiert wurden.

Methoden der Klasse Object

HOCHSCHULE HAMM-LIPPSTADT

Die Klasse *Object* bietet einige Low-Level-Dienste an.

- Jedes Objekt in C# verfügt über diese Methoden:
- *GetType*: Gibt Laufzeitinformationen zum Objekt zurück.
- *Equals*: Überprüft, ob zwei Objekte verweisgleich sind.
- *GetHashCode*: Erzeugt einen Hash-Code eines Objektes.
- *ToString*: Erzeugt eine String-Repräsentation des Objektes.

Alle diese Methoden können in den Kindklassen auch überschrieben werden.

- Insbesondere die Methode *ToString()* ist dafür prädestiniert.
- Wann immer ein Objekt Text umgewandelt werden soll, wird diese Methode benutzt.

Das base-Schlüsselwort

HOCHSCHULE HAMM-LIPPSTADT

Auch wenn in einer Kindklasse eine Methode verborgen oder überschrieben wurde, kann in dem Objekt dennoch auf die geerbte Methode zugegriffen werden.

- Hierfür kann das Schlüsselwort `base` benutzt werden.

Ähnlich zu `this` stellt `base` einen Objektverweis dar.

- Im Objekt einer Kindklasse verweist `base` auf das *Basisobjekt*.
- In einem Student-Objekt können wir so auch die Ausgabe-Routine der Person aufrufen.

```
1  public class Student : Person
2  {
3      // Der Rest fehlt hier...
4
5      public new void Ausgeben()
6      {
7          base.ausgeben();
8          Console.WriteLine("Matrikelnummer: {0}", Matrikelnummer);
9      }
10 }
```

Abstrakte Klassen

Klassen können Schnittstellen implementieren.

- Dabei definieren Schnittstellen lediglich Methodensignaturen, also einen Vertrag.
- Die Klasse muss die Schnittstelle dann aber mit einer eigenen Implementierung ausfüllen.

Die Vererbung ist hier noch stärker.

- Erbt eine Klasse von einer Basisklasse, so wird nicht nur ein Vertrag an die Kindklasse weitergegeben.
- Auch die Implementierung der Methoden wird vererbt.

Abstrakte Klassen vermischen diese beiden Prinzipien.

- Abstrakte Klassen können für einige Methoden lediglich die Signaturen definieren (wie eine Schnittstelle).
- Sie können für andere Methoden aber auch die Implementierung mitliefern.

Grafikprogramm

HOCHSCHULE HAMM-LIPPSTADT

Stellen wir uns vor, wir wollten ein Grafikprogramm schreiben.

- Darin wollen wir einige Elemente, wie z.B. Kreise, Rechtecke, Sterne usw. anbieten.

Alle diese Elemente haben gewisse Gemeinsamkeiten.

- Jedes dieser Elemente besitzt z.B. eine Bildschirmposition in Form von X/Y-Koordinaten.
- Entsprechend können wir eine Basisklasse einführen: *GrafischesElement*.

Jedes Element muss auch später auf dem Bildschirm gezeichnet werden.

- Wie die einzelnen Elemente aber gezeichnet werden müssen, unterscheidet sich aber.
- Ein Dreieck sieht ganz anders aus, als ein Stern.
- Daher kann die Basisklasse nur vorgeben, dass eine Methode Zeichne() geben soll.
- Nicht aber, wie diese Methode implementiert werden soll.
- Die Methoden Zeichne() besitzt in der Klasse *GrafischesElement* daher nur eine Signatur.

Abstrakte Klasse in C#

HOCHSCHULE HAMM-LIPPSTADT

Eine Klasse, die für mindestens eine Methode nur eine Signatur besitzt, ist abstrakt.

- Sowohl die Klasse selbst, als auch alle nicht implementierten Methoden sind mit dem Schlüsselwort `abstract` markiert.

```
1  abstract class GrafischesElement
2  {
3      public int X { get; set; }
4      public int Y { get; set; }
5
6      public void Bewege(int delta_x, int delta_y)
7      {
8          this.X += delta_x;
9          this.Y += delta_y;
10     }
11
12     public abstract void Zeichne();
13 }
```

Abstrakte Klassen benutzen

HOCHSCHULE HAMM-LIPPSTADT

Von einer abstrakten Klasse können keine Objekte erzeugt werden.

- Eine abstrakte Klasse dient ausschließlich dazu, gemeinsame Funktionalität zu sammeln.
- Über die Vererbung wird diese Funktionalität dann an die Kindklassen weiter gegeben.
- Abstrakte Klassen dienen als Basisklasse in Vererbungshierarchien.

Abstrakte Methoden müssen in einer Kindklasse implementiert werden.

- Die zu implementierenden Methoden müssen dazu überschrieben werden.
- Dazu wird erneut das Schlüsselwort `override` benutzt.

```
1  class Dreieck : GrafischesElement
2  {
3      public override void Zeichne()
4      {
5          Console.WriteLine("Das Dreieck wird gezeichnet");
6      }
7  }
```

Typkonvertierung und abstrakte Klassen

Wir können nun natürlich Objekte der Klasse Dreieck erzeugen.

- Rufen wir die Methode Zeichne() auf, wird der Text »*Das Dreieck wird gezeichnet*« auf der Konsole ausgegeben.

```
1  Dreieck d = new Dreieck() { X=10, Y=15 };
2  d.Zeichne();
```

In einer Vererbungshierarchie können wir Objekte in den Typ der Basisklasse konvertieren.

- Ein *Dreieck* können wir demnach in ein *GrafischesElement* umwandeln.
- Auf diesem Objekt können wir dann auch die Methode Zeichne() aufrufen.
- Es wird dann die Implementierung der Dreieck-Klasse benutzt.

```
1  GrafischesElement g = d;
2  g.Zeichne();
```

Polymorphie

Dieses Verhalten haben wir bereits im letzten Abschnitt beobachtet und ist dennoch bemerkenswert.

- Ein Objekt sieht aus wie ein *GrafischesElement*, verhält sich zur Laufzeit aber wie ein *Dreieck*.

Dieses Verhalten wird auch als **polymorphes Verhalten** bzw. **Polymorphie** bezeichnet.

- Das Wort Polymorphie kommt aus dem Griechischen und bedeutet Vielgestaltigkeit.
- Die Polymorphie ist ein sehr wichtiges Konzept der Objektorientierung.

Durch die Polymorphie können wir überall dort, wo ein *GrafischesElement* benötigt wird, auch ein *Dreieck* verwenden.

- Die Basisklasse *GrafischesElement* definiert im übertragenen Sinn die Steckdose.
- Zur Laufzeit können in diese Steckdose unterschiedliche Geräte (Objekte) eingesteckt werden, z.B. ein Dreieck.

Tic Tac Toe

HOCHSCHULE HAMM-LIPPSTADT

Wir wollen an einem Beispiel sehen, wie die Polymorphie genutzt werden kann.

- In einem vorhergehenden Abschnitt haben wir uns mit dem Spiel Tic Tac Toe befasst.
- Wir wollen das Spiel nun so erweitern, dass wir das Spiel mit unterschiedlichen Arten von Spielern spielen können (menschlicher Spieler oder Computerspieler)

Bislang haben wir lediglich eine Klasse *ComputerSpieler* erstellt.

- Diese Klasse setzt einen automatisierten Spieler um.
- Der Spielstein wird jeweils zufällig auf ein freies Feld abgelegt.
- Die Klasse besitzt dazu die Methode *Ziehe()*.

Wir wollen nun auch eine Klasse *MenschlicherSpieler* umsetzen.

- Ein solcher Spieler soll nach Koordinaten für den nächsten Zug fragen.
- Wenn der Platz frei ist, wird der Spielstein dort abgelegt.

Basisklasse für Spieler

Die Klassen *ComputerSpieler* und *MenschlicherSpieler* haben Gemeinsamkeiten.

- Objekte beider Klassen verwalten ihren Spielstein in einer Eigenschaft.
- Objekte beider Klassen benötigen auch eine Methode *Ziehe()*, um ihren Zug umzusetzen.

Weisen Klassen solche Gemeinsamkeiten auf, ist es sinnvoll, eine Basisklasse zu schaffen.

- Die Methode *Ziehe()* ist abstrakt und muss durch jede Kindklasse selbst implementiert werden.

```
1  abstract class Spieler
2  {
3      public char Spielstein { get; set; }
4
5      public Spieler(char spielstein)
6      {
7          Spielstein = spielstein;
8      }
9
10     public abstract void Ziehe(Spielfeld feld);
11 }
```

ComputerSpieler

Die Klasse *ComputerSpieler* können wir nun so ändern, dass sie von der Basisklasse *Spieler* ableitet.

```
1  class ComputerSpieler : Spieler
2  {
3      private Random rnd = new Random();
4
5      public ComputerSpieler(char spielstein) : base(spielstein)
6      {
7      }
8
9      public override void Ziehe(Spielfeld feld)
10     {
11         // Der Programmcode hier ist noch der selbe, wie zuvor
12     }
13 }
```

MenschlicherSpieler

Auch die Klasse *MenschlicherSpieler* lässt sich nun leicht erstellen.

- Die Klasse leiten wir ebenso von der Basisklasse *Spieler* ab.
- In der Methode *Ziehe()* müssen wir lediglich den Nutzer nach Koordinaten fragen.

```
1  lass MenschlicherSpieler : Spieler
2  {
3      public MenschlicherSpieler(char spielstein) : base(spielstein)
4      {
5      }
6
7      public override void Ziehe(Spielfeld feld)
8      {
9          // Nutzer nach gültigen und freien X/Y-Koordinaten fragen
10
11         feld.Setzen(x, y, Spielstein);
12     }
13 }
```

TicTacToe

In der Klasse TicTacToe nutzen wir bislang zwei Objekte der Klasse *ComputerSpieler*.

- Hier können wir nun stattdessen zwei Objekte vom Typ *Spieler* benutzen.

```
1  class TicTacToe
2  {
3      private Spielfeld feld;
4      private Spieler aktueller_spieler;
5
6      public Spieler Spieler1 { get; set; }
7      public Spieler Spieler2 { get; set; }
8
9      // Der Rest bleibt gleich
10 }
```

Spieler1 und *Spieler2* müssen vor Beginn des Spiels Objekte zugewiesen bekommen.

- Da wir aus der abstrakten Klasse Spieler keine Objekte erzeugen können, können dies nur Objekte der Klassen *MenschlicherSpieler* oder *ComputerSpieler* sein.

Spiel starten

HOCHSCHULE HAMM-LIPPSTADT

Dies ermöglich uns, zur Laufzeit dynamisch festzulegen, wer gegen wen spielen soll.

- Zwei Computerspieler gegeneinander, ein Computerspieler gegen einen Menschen, ...
- Wir konfigurieren die Zusammenstellung des Spiels durch unterschiedliche Objekte, z.B.:

```
1  TicTacToe ttt = new TicTacToe();
2  ttt.Spieler1 = new ComputerSpieler('X');
3  ttt.Spieler2 = new MenschlicherSpieler('O');
4  ttt.StarteSpiel();
```

Jenachdem, welche Art von Objekten wir den Eigenschaften *Spieler1* und *Spieler2* zuweisen, verhält sich das Spiel dann anders.

- Das Spiel weiß nicht, um welche Art von Spieler es sich jeweils konkret handelt.
- Es ist auch nicht wichtig, da sichergestellt ist, dass jedes Objekt eine eigene Implementierung von *Ziehe()* besitzt.

Erkenntnisse

Im TicTacToe-Spiel haben wir die Polymorphie bewusst eingesetzt.

- Es ergibt sich dadurch die Möglichkeit, erst zur Laufzeit die konkrete Art des Spielers festlegen zu müssen.
- Die beiden Eigenschaften *Spieler1* und *Spieler2* stellen die Steckdosen dar, in die wir konkrete Objekte der Klassen *MenschlicherSpieler* oder *ComputerSpieler* einstecken können.

Technisch wird die Steckdose durch die abstrakte Klasse *Spieler* realisiert.

- In den abgeleiteten Klassen überschreiben wir die Methode *Ziehe()*.
- Dazu wird das Schlüsselwort `override` genutzt.

Abstrakte Klassen sind ein Weg, um polymorphes Verhalten zu erhalten.

- Denselben Effekt können wir auch mithilfe von Schnittstellen erzeugen.

Polymorphie mit Schnittstelle

Wir wollen nun polymorphes Verhalten mithilfe einer Schnittstelle erzeugen.

- Wir führen dazu die Schnittstelle *ISpieler* ein.

Sie definiert, welche Methoden eine Klasse anbieten muss, um als Spieler zu gelten.

- Wie bereits zuvor benötigen wir eine Eigenschaft, um den Spielstein zu speichern.
- Zudem wird auch die Methode *Ziehe()* gebraucht.

```
1  interface ISpieler
2  {
3      char Spielstein { get; set; }
4      void Ziehe(Spielfeld feld);
5  }
```

Schnittstelle Implementieren

HOCHSCHULE HAMM-LIPPSTADT

Anstelle von einer Basisklasse abzuleiten, implementieren die beiden Klassen *MenschlicherSpieler* und *ComputerSpieler* nun die Schnittstelle *ISpieler*.

- Es ist dann aber etwas mehr zu tun, weil z.B. die Eigenschaftsmethode Spielstein in den Klassen selbst implementiert werden muss.

```
1  class MenschlicherSpieler : ISpieler
2  {
3      public char Spielstein { get; set; }
4
5      public MenschlicherSpieler(char spielstein)
6      {
7          Spielstein = spielstein;
8      }
9
10     // Der Rest bleibt gleich
11 }
```

Klasse TicTacToe anpassen

HOCHSCHULE HAMM-LIPPSTADT

In der Klasse *TicTacToe* müssen die Eigenschaften *Spieler1* und *Spieler2* angepasst werden.

- Der Datentyp der beiden Eigenschaften ist dann *ISpieler*.
- Den Eigenschaften können all jene Objekte zugewiesen werden, deren Klasse die Schnittstelle *ISpieler* implementiert.
- Dies ermöglicht erneut die dynamische Auswahl von passenden Objekten.

```
1  class TicTacToe
2  {
3      private Spielfeld feld;
4      private ISpieler aktueller_spieler;
5
6      public ISpieler Spieler1 { get; set; }
7      public ISpieler Spieler2 { get; set; }
8
9      // Der Rest bleibt gleich...
10 }
```

Zusammenfassung

HOCHSCHULE HAMM-LIPPSTADT

Wir haben gelernt, ...

- dass in C# die Klasse *Object* die ultimative Basisklasse ist.
- welche Methoden daher alle Objekte in C# besitzen.
- was es bedeutet, wenn man von Polymorphie spricht.
- wie man polymorphes Verhalten mithilfe Vererbung herstellen kann.
- wie man polymorphes Verhalten mithilfe von Schnittstellen herstellen kann.

Aufgabe 1

HOCHSCHULE HAMM-LIPPSTADT

Erstellen Sie eine Klasse *Grafikprogramm*.

- Die Klasse soll eine Menge von Objekten des Typs *GrafischesElement* verwalten können.
- Nutzen Sie dazu eine Objektvariable vom Typ *List*.
- Erstellen Sie eine Methode, um der Liste ein Objekt hinzufügen zu können.

Implementieren Sie die Klassen *Rechteck* und *Kreis* als Kindklassen des Klasse *GrafischesElement*.

Aufgabe 2

HOCHSCHULE HAMM-LIPPSTADT

Das Grafikprogramm soll die grafischen Elemente als Bilddatei speichern können.

- Bekanntlich existieren unterschiedliche Bildformate, z.B. Jpeg oder Png.
- Das Grafikprogramm soll zur Laufzeit genau ein solches Format kennen.
- Erstellen Sie zwei Dummy-Klassen für die Speicherung in einem Format, z.B. die Klassen *JpegFormat* und *PngFormat*.
- Beide Klassen sollen über die Methode *Speichern()* verfügen.

Sorgen Sie mithilfe von Polymorphie dafür, dass das Grafikprogramm dynamisch zur Laufzeit eines der beiden Formate zur Speicherung nutzen kann.

- Nutzen Sie entweder eine abstrakte Basisklasse oder eine Schnittstelle.

Quellen I