

Objektorientierte Programmierung

06 - Objektbeziehungen

Alexander Stuckenholtz

Version 2022-05-04

Inhalt

- 1 Vereinsverwaltung
- 2 Objektbeziehungen
- 3 Tic Tac Toe
- 4 Zusammenfassung und Aufgaben

In den letzten Abschnitten haben wir die *Bruch*-Klasse realisiert.

- Dadurch konnten wir viele konkrete *Bruch*-Objekte erzeugen.
- Mit Hilfe von Methoden konnten wir dadurch die Bruchrechnung im Rechner abbilden.

Wir wollen nun noch ein anderes Beispiel umsetzen.

- Wir wollen damit beginnen, eine Verwaltungssoftware für einen Sportverein zu implementieren.

Ein Sportverein hat bekanntlich Mitglieder.

- Jedes Mitglied hat einen Vor- und einen Nachnamen.
- Zudem noch ein Geburtsdatum.

Um Mitglied-Objekte erzeugen zu können, benötigen wir also die *Mitglied*-Klasse.

Die *Mitglied*-Klasse besitzt Eigenschaftsmethoden für Vor- und Nachnamen, sowie das Geburtsdatum.

- In den Set-Teilen der Eigenschaftsmethoden benötigen wir keine besondere Prüfung der übergebenen Werte.
- Entsprechend können wir auch die abgekürzte Schreibweise nutzen.

```
1  class Mitglied
2  {
3      public string Vorname { get; set; }
4      public string Nachname { get; set; }
5      public DateTime Geburtsdatum { get; set; }
6
7      public Mitglied(string v, string n, string g)
8      {
9          Vorname = v;
10         Nachname = n;
11         Geburtsdatum = Convert.ToDateTime(g);
12     }
13 }
```

Nun reicht die *Mitglied*-Klasse allein aber nicht aus, um eine Verwaltungssoftware für einen Sportverein umzusetzen.

- Der Sportverein selbst besitzt ja auch Eigenschaften.
- Er hat einen Namen, er *kennt* einen Vorsitzenden und seine Mitglieder.

In der objektorientierten Programmierung ist alles ein Objekt.

- Also auch der Sportverein.

Für den Sportverein erstellen wir dementsprechend auch eine Klasse.

- Das ist notwendig und sinnvoll.
- Auch dann, wenn wir zur Laufzeit nur ein einziges Objekt dieser Klasse besitzen werden.

Klasse Sportverein

Für alle Daten, die ein Objekt der Klasse *Sportverein* verwalten soll, werden Objektvariablen benötigt.

- Der Name ist eine einfache Zeichenkette vom Typ `string`.
- Dazu erstellen wir eine Eigenschaftsmethode.

Der Vorsitzende und die Mitglieder werden allerdings über Objektreferenzen abgebildet.

```
1  class Sportverein
2  {
3      private Mitglied vorsitzender;
4      private List<Mitglied> mitglieder = new List<Mitglied>();
5
6      public string Name { get; set; }
7
8      public Sportverein(string n)
9      {
10         Name = n;
11     }
12 }
```

Die Objektvariable *vorsitzender* kann eine Objektreferenz auf ein einzelnes *Mitglied*-Objekt aufnehmen.

- Die Objektvariable ist zunächst vor dem Zugriff von außerhalb des Objekts geschützt (*private*).
- Über eine Eigenschaftsmethode können wir aber erneut den Zugriff gewähren.
- Im Set-Teil können wir dann prüfen, ob die übergebene Objektreferenz auch auf ein Objekt verweist.

```
1 public Mitglied Vorsitzender
2 {
3     get { return vorsitzender; }
4     set
5     {
6         if (value == null)
7             throw new Exception("Der Verein muss einen Vorsitzenden haben!");
8
9         vorsitzender = value;
10    }
11 }
```

Die Objektvariable *mitglieder* ist ein Objekt der Klasse *List*.

- Das Objekt kann beliebig viele Objektreferenzen auf Mitglied-Objekte verwalten.
- Auch diese Objektvariable ist sinnvollerweise zunächst privat.

Um dem Sportverein Mitglieder hinzufügen zu können, erstellen wir eine Methode:

```
1 public void MitgliedHinzufuegen(Mitglied m)
2 {
3     mitglieder.Add(m);
4 }
```

Der Sportverein nimmt dann die übergebene Objektreferenz in seiner eigenen Objektvariable auf.

- Der Sportverein *kennt* dadurch seine Mitglieder.

Nachdem wir so dem Sportverein Mitglieder bekannt machen können, wollen wir den Sportverein auch fragen können, welche Mitglieder er kennt.

- Dazu implementieren wir natürlich erneut eine eigene Methode.

```
1 public IEnumerable<Mitglied> AlleMitglieder()  
2 {  
3     return mitglieder;  
4 }
```

Als Antwort der Methode sollte nicht die Objektvariable *mitglieder* selbst zurückgegeben werden.

- Da dann ja die Objektreferenz selbst zurückgegeben wird, könnte der Nutzer die Liste dadurch dann manipulieren, z.B. komplett leeren.
- Stattdessen sollte ein Objekt des Typs *IEnumerable* zurückgegeben werden.
- Da die Klasse *List* diese Schnittstelle implementiert, kann sie implizit in diesen Typ konvertiert werden.

Mit der Klasse Mitglied können wir nun Mitglieder als Objekte abbilden.

- Die Klasse Sportverein überträgt das Konzept des Sportvereins selbst in den Rechner.

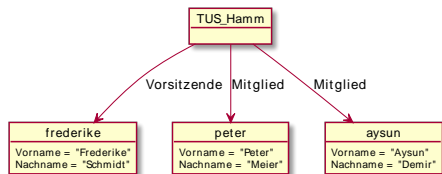
```
1 Sportverein s = new Sportverein("TUS Hamm");
2 s.Vorsitzender = new Mitglied("Erika", "Wischnowski", "3.2.2003");
3 s.MitgliedHinzufuegen(new Mitglied("Peter", "Pan", "6.8.1997"));
4
5 foreach (var m in s.AlleMitglieder())
6 {
7     Console.WriteLine(m.Vorname);
8 }
```

Durch die beiden Klassen können wir zur Laufzeit ein Geflecht aus Objekten erzeugen.

- Der Sportverein kennt seinen Vorsitzenden und seine Mitglieder.
- Der Sportverein kann die Methoden und Eigenschaften der Mitglied-Objekte nutzen.
- Dadurch können Aufgaben auf unterschiedliche Objekte verteilt werden.

Das Beispiel zeigt, wie **Objektbeziehungen** realisiert werden können.

- Auf dieser Basis interagieren Objekte in objektorientierten Systemen miteinander.
- In unserem Beispiel kann der Sportverein dann die Eigenschaften und Methoden des Mitglieds nutzen, um z.B. bestimmte Mitglieder zu filtern.



Solche Objektbeziehungen sind unidirektional, sie zeigen in eine bestimmte Richtung.

- Im Beispiel kennt der Sportverein seine Mitglieder.
- Die Mitglied-Objekte wissen aber nichts von einem Sportverein-Objekt.

Objekte und Objektbeziehungen bilden die **Grundlage der Objektorientierung**.

- Mit diesen Bausteinen lassen sich beliebige Strukturen im Rechner abbilden.
- Solche Strukturen aufzubauen zu können, ist der Kern der Objektorientierung.

Mithilfe von Objekten und Objektbeziehungen können beliebige Systeme entworfen werden.

- Sinnvolle Objektgeflechte können für alle Bereiche gefunden werden.
- Ganz egal, ob es sich um ein Computerspiel, ein Steuer- und Regelungssystem für ein Kernkraftwerk oder eine Finanzbuchhaltung handelt.

Entscheidend dabei ist, wie die Aufteilung auf Objekte gewählt wird.

- Über die entsprechenden Entwurfsprinzipien werden wir im weiteren Verlauf noch reden.

Das Spiel Tic Tac Toe kennen wir bereits.

- Bislang haben wir das Spiel allerdings noch nicht objektorientiert umgesetzt.
- Entsprechend konnten wir den Spielablauf, das Spielfeld und die Spieler nicht konzeptuell voneinander trennen.
- Alles war irgendwie miteinander vermischt.

Mithilfe der Objektorientierung können wir nun einen besseren Entwurf realisieren.

- Das Spielfeld und die Spieler setzen wir getrennt voneinander als Objekte um.

Ein weiteres Objekt bildet das Spiel und seinen Ablauf selbst ab.

- Dieses Objekt referenziert die anderen Objekte und nutzt sie um den Spielablauf zu steuern.

Wir beginnen mit dem Spielfeld und erstellen dazu eine gleichnamige Klasse.

- Um die Belegung des Spielfelds abzubilden, nutzen wir natürlich intern ein Array.
- Der Zugriff darauf wird aber ausschließlich über entsprechende Methoden gestattet.

```
1  class Spielfeld
2  {
3      private char[,] feld;
4
5      public Spielfeld()
6      {
7          // Hier das feld initialisieren
8      }
9
10     public void Setzen(int zeile, int spalte, char spielstein)
11     {
12         feld[zeile, spalte] = spielstein;
13     }
14 }
```

Weitere Methoden des Spielfelds

Das Setzen des Spielsteins basiert auf einer einfachen Zuweisung in einem Array.

- Dennoch ist es sehr sinnvoll, dies in Form einer eigenen Methode zu implementieren.
- Wenn die Methode später benutzt wird, versteht man durch die Benennung sehr schnell, was gemacht wird.

Um mit dem Spielfeld vernünftig interagieren zu können, werden weitere Methoden benötigt.

- Sinnvoll ist z.B. eine Methode zur Ausgabe auf der Konsole: `void Ausgeben() { ... }`

Auch werden mehrere Prüfungen benötigt, z.B.:

- ob ein Platz im Spielfeld belegt ist: `public void IstBelegt(int zeile, int spalte) { ... }`
- ob ein bestimmter Spieler gewonnen hat: `public bool HatGewonnen(char spielstein) { ... }`
- ob irgendein Spieler gewonnen hat: `public bool HatIrgendwerGewonnen() { ... }`
- ob noch irgendein Feld frei ist: `public bool IstEinFeldFrei() { ... }`

Es sollte nicht schwierig sein, diese Methoden zu implementieren.

Das Spielfeld allein macht noch kein Tic Tac Toe.

- Wir benötigen noch zwei Spieler, die ihre Spielsteine auf dem Spielfeld ablegen können.
- Dies sind natürlich auch Objekte, die wiederum das Spielfeld-Objekt benutzen.

Später wollen wir dafür sorgen, dass ein Mensch gegen den Computer spielen kann.

- Zunächst wollen wir das Spiel aber so umsetzen, dass zwei Computergegner gegeneinander spielen.

Entsprechend erstellen wir eine Klasse *ComputerSpieler*.

- Zur Laufzeit erzeugen wir dann zwei Objekte dieser Klasse.
- Sie benutzen ein Objekt der Klasse Spielfeld, um ihre Spielsteine dort abzulegen.

Jedes Objekt der Klasse *ComputerSpieler* muss sich seinen Spielstein merken.

- Also benötigen wir eine entsprechende Eigenschaft, in der wir dann ein X oder ein O ablegen können.

```
1  class Spieler
2  {
3      public char Spielstein { get; set; }
4
5      public Spieler(char spielstein)
6      {
7          // Der eigene Spielstein (X oder O) wird dem Konstruktor übergeben
8          Spielstein = spielstein;
9      }
10 }
```

Die wichtigste Methode eines Spielers ist *Ziehe()*.

- Als Parameter wird ein Spielfeld-Objekt übergeben.
- Der *ComputerSpieler* legt seinen Spielstein einfach auf einem zufälligen freien Feld ab.

```
1 public void Ziehe(Spielfeld feld) {
2     if (!feld.IstEinFeldFrei())
3         return;
4
5     while (true) {
6         int zeile = rnd.Next(0, 3);
7         int spalte = rnd.Next(0, 3);
8
9         if (!feld.IstBelegt(zeile, spalte)) {
10             feld.Setzen(zeile, spalte, Spielstein);
11             return;
12         }
13     };
14 }
```

Zu guter Letzt müssen wir diese Objekte noch miteinander zu einem Spiel verbinden.

- Auch dazu erstellen wir eine neue Klasse *TicTacToe*.
- Ein Objekt der Klasse kennt ein Spielfeld und zwei Objekte der Klasse *ComputerSpieler*.

```
1  class TicTacToe
2  {
3      private Spielfeld feld;
4      private ComputerSpieler spieler1 = new ComputerSpieler('X');
5      private ComputerSpieler spieler2 = new ComputerSpieler('O');
6      private Spieler aktueller_spieler;
7
8      ...
9  }
```

Wir führen zudem noch ein weiteres Feld `aktueller_spieler` ein.

Der Spielverlauf wechselt bekanntlich zwischen den beiden Spielern hin und her.

- Dieses Verhalten können wir so abbilden, dass das Feld `aktueller_spieler` abwechselnd auf eines der beiden Spieler-Objekte verweist.
- Diesen Wechsel können wir mit einer eigenen Methode abbilden:

```
1 private void WechsleSpieler()  
2 {  
3     if (aktueller_spieler == spieler1)  
4         aktueller_spieler = spieler2;  
5     else  
6         aktueller_spieler = spieler1;  
7 }
```

Diese Mechanik können wir überhaupt nur deswegen abbilden, weil wir nun Spieler-Objekte besitzen.

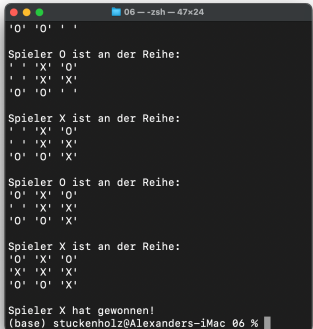
Der eigentliche Spielablauf kann nun in einer Methode *StarteSpiel()* abgebildet werden.

- Der Programmcode liest sich fast wie eine Spielanleitung.

```
1  public void StarteSpiel() {
2      feld = new Spielfeld();
3
4      do {
5          WechsleSpieler();
6          Console.WriteLine("Spieler " + aktueller_spieler.Spielstein + " ist an der Reihe:");
7          aktueller_spieler.Ziehe(feld);
8          feld.Ausgeben();
9
10         if (feld.HatGewonnen(aktueller_spieler.Spielstein)) {
11             Console.WriteLine("Spieler " + aktueller_spieler.Spielstein + " hat gewonnen!");
12             break;
13         }
14     }
15     while (feld.IstEinFeldFrei());
16
17     if (!feld.HatGewonnen() && !feld.IstEinFeldFrei())
18         Console.WriteLine("Unentschieden!");
19 }
```

Die Klasse TicTacToe kann nun dazu genutzt werden, das eigentliche Spiel zu starten:

```
1 TicTacToe ttt = new TicTacToe();  
2 ttt.StarteSpiel();
```



```
'O' 'O' ' '  
  
Spieler 0 ist an der Reihe:  
' ' 'X' 'O'  
' ' 'X' 'X'  
'O' 'O' ' '  
  
Spieler X ist an der Reihe:  
' ' 'X' 'O'  
' ' 'X' 'X'  
'O' 'O' 'X'  
  
Spieler 0 ist an der Reihe:  
'O' 'X' 'O'  
' ' 'X' 'X'  
'O' 'O' 'X'  
  
Spieler X ist an der Reihe:  
'O' 'X' 'O'  
'X' 'X' 'X'  
'O' 'O' 'X'  
  
Spieler X hat gewonnen!  
(base) stuckenholz@Alexanders-iMac 06 %
```

Das Spiel wurde nun **modular** mithilfe objektorientierter Prinzipien realisiert.

- Die Spieler, das Spielfeld, und der Spielablauf wurden getrennt voneinander umgesetzt.
- Mithilfe von Objektbeziehungen können wir die Objekte zu einem **großen Ganzen** zusammensetzen.

Beim Entwurf der Klassen mussten wir jeweils überlegen, welche Aufgaben die Objekte **im Zusammenspiel** übernehmen müssen.

- Ist der Entwurf einigermaßen gelungen, können die Klassen dann unabhängig voneinander erweitert, getestet und ausgetauscht werden.
- Später können wir z.B. die aktuelle Spieler-Klasse durch eine andere ersetzen, ohne dass wir das Spielfeld usw. ändern müssen.

Der Schlüssel zum Erfolg ist dabei ein gutes Design der Klassen und der Zuschnitt der Aufgaben im Gesamtkontext.

Wir haben heute gelernt, dass...

- Objekte zur Laufzeit Beziehungen untereinander aufbauen können.
- solche Beziehungen auf Objektreferenzen basieren.
- man mit diesem Mechanismus komplexe Systeme aus vielen unterschiedlichen Objekten umsetzen kann.

Setzen Sie das TicTacToe-Spiel komplett um, ohne sich den Programmcode im Quellcode-Repository anzusehen.

- Implementieren Sie alle Methoden der Klassen.

Ersetzen Sie die Klasse *ComputerSpieler* durch eine neue Variante.

- Diese Variante soll einen menschlichen Spieler realisieren.
- In jedem Zug wird nach der gewünschten Position für den Spielstein gefragt.
- Bitte beachten: Ein Spielstein kann nur auf eine freie Position gesetzt werden.

Eine Mitfahrzentrale braucht eine neue Software, um Fahrten zu koordinieren.

- Das System soll dabei objektorientiert realisiert werden.

Erstellen Sie eine Klasse *Person*, mit entsprechenden Eigenschaften und Eigenschaftsmethoden.

- Erstellen Sie eine weitere Klasse *Fahrt*.
- Eine *Fahrt* hat einen Start- und einen Zielort (*string*).
- Eine *Fahrt* kennt einen Fahrer (*Person*) und mehrere Mitfahrer (*Personen*).
- Sorgen Sie dafür, dass nicht mehr als 3 Mitfahrer hinzugefügt werden können.

Nutzen Sie die Klassen, um mehrere Objekte der Klasse *Fahrt* zu erzeugen.

- Wie kann man dafür sorgen, dass ein Objekt der Klasse *Fahrt* nicht erzeugt werden kann, ohne einen Fahrer zu benennen?

Quellen I