Objektorientierte Programmierung

07 - Vererbung

Alexander Stuckenholz

Version 2022-05-04

Inhalt

Hochschulverwaltung

HOCHSCHULE HAMM-LIPPSTADT

Stellen wir uns vor, wir müssten eine Anwendung entwerfen, um eine Hochschule zu verwalten.

- Neben vielen Anderen Dingen finden wir an einer Hochschule auch Dozenten und Studierende.
- Damit wir solche Objekte erzeugen können, benötigen wir entsprechende Klassen.

Wir erstellen also die beiden Klassen *Student* und *Dozent*.

- Beide Klassen benötigen Eigenschaftsfunktionen für Vor- und Nachname.
- Der Student hat eine Matrikelnummer, der Dozent ein Lehrgebiet.

```
1  class Student
2  {
3      public string Vorname { get; set; }
4      public string Nachname { get; set; }
5      public int Matrikelnummer { get; set; }
6  }
```

```
1  class Dozent
2  {
3      public string Vorname { get; set; }
4      public string Nachname { get; set; }
5      public string Lehrgebiet { get; set; }
6  }
```

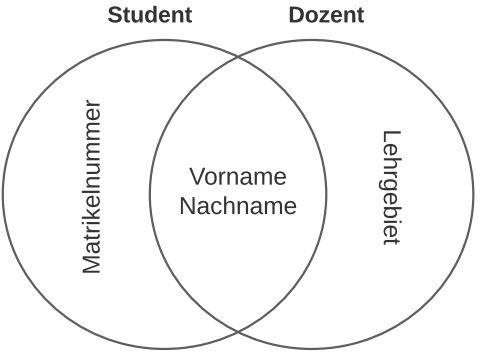
Gemeinsamkeiten

Wir stellen fest, dass Student und Dozent einigen Programmcode gemeinsam haben.

- Beide Klassen besitzen Eigenschaftsmethoden für Vor- und Nachnamen.

Dass Student und Dozent solche Gemeinsamkeiten aufweisen ist natürlich kein Zufall.

- Studenten und Dozenten haben ja gewisse Ähnlichkeiten.
- Beide sind menschliche Wesen!



Redundanzen

HOCHSCHULE HAMM-LIPPSTADT

Redundanter Programmcode ist aus vielen Gründen sehr schlecht!

- Redundanter Programmcode ist schwer zu lesen und zu verstehen.
- Redundanter Programmcode ist schwer zu warten und zu pflegen!

Redundanzen im Programmcode sollten unbedingt vermieden werden!

- *Don't repeat yourself (DRY)* ist eine wichtige Basisregel, um qualitativen Programmcode zu erzeugen! (Siehe [1, S. 48])

Die Frage ist aber, wie wir hier diesen redundanten Programmcode vermeiden können?

- Um Ähnlichkeiten bei Klassen zusammenzuführen, bietet die Objektorientierung ein bestimmtes Konzept an.
- Die **Vererbung**.

Vererbung

Bei der Vererbung vererbt eine **Basisklasse** (auch Super-, Ober- oder Elternklasse) alle Eigenschaften an eine oder mehrere **Kindklassen**.

- Man spricht auch davon, dass man neue Klassen aus einer Basisklasse **ableitet**.

Durch die Vererbung besitzen die Kindklassen alle Eigenschaften der Basisklasse.

- Alle Objektvariablen, Eigenschaften und Methoden der Basisklasse.

Die Kindklassen können diese Elemente benutzen, ohne sie selbst definieren zu müssen.

- Vererbung hilft also dabei, Programmcode wiederzuverwenden und Redundanzen zu vermeiden, siehe auch [2, S. 68]

Die Kindklassen können zusätzliche Elemente definieren.

- Neue Variablen, Eigenschaften und Methoden.
- Die Kindklassen erweitern die Basisklassen dann um weitere Elemente.

Person

Wir können nun eine gemeinsame Basisklasse *Person* schaffen.

- Dort können wir solche Elemente sammeln, die Student und Dozent gemeinsam haben.
- Wir verschieben also die beiden Eigenschaftsmethoden für Vor- und Nachname in diese Klasse.

```
1  public class Person
2  {
3      public string Vorname { get; set; }
4      public string Nachname { get; set; }
5  }
```

Student und Dozent

HOCHSCHULE HAMM-LIPPSTADT

Die Klassen *Student* und *Dozent* können wir nun von der Klasse *Person* ableiten.

- Der Name der Basisklasse wird dazu hinter dem Doppelpunkt aufgeführt.
- Es wird also dieselbe Schreibweise genutzt, als würde die Klasse eine Schnittstelle implementieren.

```
1  public class Student : Person
2  {
3      public int Matrikelnummer { get; set; }
4  }
```

```
1  public class Dozent : Person
2  {
3      public string Lehrgebiet { get; set; }
4  }
```

Über die geerbten Elemente hinaus definieren beide Kindklassen noch zusätzliche Elemente.

- Zusätzlich zum Vor- und zum Nachnamen besitzt der Student noch eine Matrikelnummer.
- Der Dozent weist ein Lehrgebiet aus.

Klassen benutzen

Aus allen drei Klassen können nun Objekte erzeugt werden.

- Alle Objekte besitzen einen Vor- und einen Nachnamen.
- Nur Objekte der Klasse Student besitzen zusätzlich eine Matrikelnummer.
- Nur Objekte der Klasse Dozent besitzen zusätzlich ein Lehrgebiet.

```
1  var p = new Person() { Vorname = "Ingrid", Nachname = "Müller" };
2  var s = new Student() { Vorname = "Demir", Nachname = "Öztürk", Matrikelnummer = 12345 };
3  var d = new Dozent() { Vorname = "Ulrike", Nachname = "Demirel", Lehrgebiet = "Mathematik" };
```

Einfach- vs Mehrfachvererbung

HOCHSCHULE HAMM-LIPPSTADT

In C# darf eine Klasse nur von maximal einer Basisklasse ableiten.

- Dies wird als **Einfachvererbung** bezeichnet.

Andere Programmiersprachen erlauben auch die Mehrfachvererbung, z.B. C++.

- Dabei kann es aber zu Konflikten kommen.
- Bei namensgleichen Methoden in zwei Basisklassen muss geklärt werden, welche Implementierung die Kindklasse nutzen soll.

In C# darf eine Klasse aber beliebig viele Schnittstellen implementieren.

- Selbst bei Namensgleichheit mehrere Methoden kann es dabei nicht zu Konflikten kommen.
- Eine Schnittstelle sorgt dafür, dass eine bestimmte Methode vorhanden sein muss.
- Wie diese aber implementiert wird, ist der Klasse selbst überlassen.

Spezialisierung und Generalisierung

HOCHSCHULE HAMM-LIPPSTADT

Mithilfe der Vererbung können komplexe Hierarchien gebildet werden.

- In mehreren Ebenen können Klassen voneinander ableiten.
- Eine solche Vererbungshierarchie hat eine baumartige Struktur.

Je tiefer man in der Hierarchie absteigt, desto mehr Eigenschaften weisen die Klassen auf.

- In Vererbungsrichtung werden die Klassen dadurch spezieller (**Spezialisierung**).

Blickt man hingegen entgegen der Vererbungsrichtung, werden die Klassen immer allgemeiner.

- Klassen vereinen dann die Eigenschaften mehrerer Kindklassen.
- Sie verallgemeinern die Konzepte (**Generalisierung**).

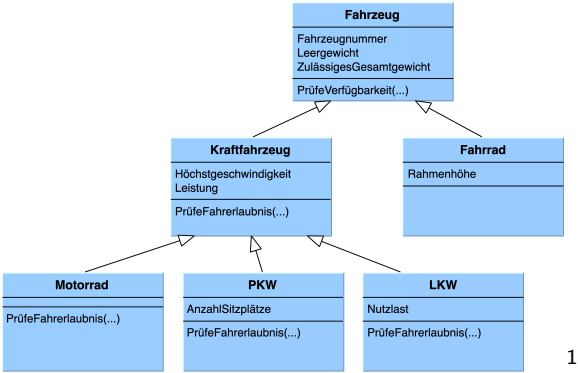
Vererbungshierarchie

Eine Vererbungshierarchie von mehreren Klassen kann auch grafisch dargestellt werden.

- Dazu wird meist ein UML-Diagramm genutzt (später mehr.)

Das folgende Beispiel zeigt ein solches UML-Diagramm.

- Die Klasse Fahrzeug ist die Basisklasse.
- Alle anderen Klassen sind abgeleitet.



1

[1] Bildquelle: Cactus26 (https://commons.wikimedia.org/wiki/File:InheritancePgmExample.svg), „InheritancePgmExample“, https://creativecommons.org/licenses/by-sa/3.0/legalcode

Erkenntnisse aus dem Beispiel

Aus der Vererbungshierarchie des Beispiels können einige Sachverhalte abgeleitet werden:

- Ein Kraftfahrzeug ist ein Fahrzeug.
- Ein Motorrad ist ein Kraftfahrzeug.
- Ein Fahrzeug ist jedoch kein LKW.

Die Basisklasse *Fahrzeug* stellt generelle Eigenschaften bereit.

- Diese Eigenschaften werden von allen Klassen in der Hierarchie geteilt.
- Alle Klassen besitzen die Attribute Fahrzeugnummer, Leergewicht, ...
- Alle Klassen besitzen die Elementfunktion *PrüfeVerfügbarkeit()*.

Die Klasse *Kraftfahrzeug* spezialisiert die Klasse *Fahrzeug* z.B. um die Eigenschaft Leistung.

- Ein Kraftfahrzeug ist demnach ein spezielles Fahrzeug.

Typumwandlung in Vererbungshierarchien

HOCHSCHULE HAMM-LIPPSTADT

Objekte einer Kindklasse bieten alle Methoden an, die auch die Basisklasse besitzt.

- Objekte der Kindklasse lassen sich also genauso benutzen, wie Objekte der Basisklasse.
- Ein Objekt einer Kindklasse kann daher in den Typ der Basisklasse umgewandelt werden (engl. *up-casting*).

```
1  var s = new Student() { Vorname="Maria", Nachname="Eitekin", Matrikelnummer=12345 };
2  var p = (Person)s;
```

Ein Objekt der Basisklasse kann aber nicht in ein Objekt der Kindklasse umgewandelt werden.

- Eine fehlende Matrikelnummer einer Person kann ja nicht hinzuerfunden werden.
- Folgender Versuch einer Typumwandlung wird daher vom Compiler mit einer Fehlermeldung verweigert.

```
1  Person p = new Person() { "Elrike", "Üzgür" };
2  Student s = (Student) p;
```

Konstruktoren und Vererbung

Durch Vererbung werden Variablen und Methoden an die Kindklassen weiter gegeben.

- Konstruktoren und der Destruktor werden allerdings nicht vererbt.
- Ansonsten könnte die korrekte Initialisierung von Objekten von Kindklassen nicht sicher gestellt werden.

Die Klasse *Person* kann um einen Konstruktor erweitert werden:

```
1  public class Person
2  {
3      public string Vorname { get; set; }
4      public string Nachname { get; set; }
5
6      public Person(string vorname, string nachname)
7      {
8          Vorname = vorname;
9          Nachname = nachname;
10     }
11 }
```

Konstruktor der Basisklasse verwenden

Da die Basisklasse *Person* nun einen Konstruktor besitzt, müssen auch die Kindklassen einen Konstrultor definieren.

- Ansonsten ist das Projekt nicht übersetzbar.

Man kann aber einen Teil der Arbeit an den Konstruktor der Basisklasse delegieren.

- Dazu wird das Schlüsselwort `base` benutzt, der auf das Objekt der Basisklasse verweist.

```
1  public class Student : Person
2  {
3      public int Matrikelnummer { get; set; }
4
5      public Student(string vorname, string nachname, int matrikelnummer) : base(vorname, nachname)
6      {
7          Matrikelnummer = matrikelnummer;
8      }
9  }
```

Adresse

HOCHSCHULE HAMM-LIPPSTADT

Wir wollen eine neue Klasse einführen, um die Wohnadresse einer Person abbilden zu können.

```
1  class Adresse
2  {
3      public string Strasse { get; set; }
4      public int PLZ { get; set; }
5      public string Ort { get; set; }
6
7      public Adresse(string strasse, int plz, string ort)
8      {
9          this.Strasse = strasse;
10         this.PLZ = plz;
11         this.Ort = ort;
12     }
13 }
```

Von dieser Klasse können wir natürlich erneut leicht Objekte erzeugen.

```
1  var w = new Adresse("Im Rosenhang 12", 59063, "Hamm");
```

Wohnadresse

HOCHSCHULE HAMM-LIPPSTADT

Objekte vom Typ Adresse sind für sich allein allerdings nicht besonders hilfreich.

- Wir müssen in der Lage sein, ein solches Objekt mit Personen in Verbindung zu setzen.
- Dies können wir mithilfe einer Objektbeziehung erreichen.
- Die Klasse *Person* erweitern wir dazu um eine Eigenschafstmethode Wohnadresse.
- Diese nimmt eine Referenz auf ein Objekt vom Typ *Adresse* auf.

```
1  class Person
2  {
3      // Der Rest fehlt hier...
4      public Adresse Wohnadresse { get; set; }
5  }
```

Einem Objekt der Klasse Person können wir nun zur Laufzeit eine Wohnadresse zuweisen:

```
1  var p = new Person("Ingrid", "Müller");
2  p.Wohnadresse = new Adresse("Im Rosenhang 12", 59063, "Hamm");
```

Objektbeziehungen und Vererbung

HOCHSCHULE HAMM-LIPPSTADT

Die Klassen *Student* und *Dozent* sind von der Klasse *Person* abgeleitet.

- Beide Klassen erben also alle Variablen, Eigenschaften und Methoden der Basisklasse.
- Dazu gehört auch die Eigenschaftsmethode *Wohnadresse*.
- Entsprechend können wir auch Objekten der Klassen *Student* und *Dozent* nun eine Wohnadresse zuweisen.

```
1  var s = new Student("Demir", "Öztürk", 12345);
2  s.Wohnadresse = new Adresse("Schlehenstraße 26", 59063, "Hamm");
3
4  var d = new Dozent("Elvira", "Kosmolski", "Physik");
5  d.Wohnadresse = new Adresse("Marker Allee 123", 59063, "Hamm");
```

Bei der Vererbung wird auch die Möglichkeit für Objektbeziehungen weiter vererbt.

- Erlaubt die Basisklasse bestimmte Objektbeziehungen, können auch die Kindklassen diese Objektbeziehungen aufbauen.

Private Objektvariablen und Vererbung

Nehmen wir an, die Klasse Person würde noch weitere Objektvariablen deklarieren, z.B. die Schuhgröße.

```
1  class Person
2  {
3      // Der Rest fehlt hier
4
5      private double schuhgroesse;
6  }
```

Obwohl die Klassen Student und Dozent Kindklassen sind und alle Variablen erben, dürfen wir dort aber nicht auf die Schuhgröße zugreifen.

- Der Compiler würde die Übersetzung des Projekts mit einem Fehler abbrechen, würden wir dies versuchen.
- Wenn Variablen privat sind, dürfen selbst Objekte von Kindklassen diese nicht nutzen.

Schlüsselwort protected

HOCHSCHULE HAMM-LIPPSTADT

Neben den beiden Zugriffsmodifizierern `private` und `public` wurde speziell für die Vererbung noch eine dritte Variante eingeführt: `protected`

- Der Zugriffsmodifizierer `protected` bewirkt, dass die abgeleiteten Klassen Zugriff auf die sonst geschützten Elemente erhalten.

Der Zugriffsmodifizierer `protected` ist also eine Kombination aus `private` und `public`.

- Ein als `protected` markiertes Element ist `public` für alle Objekte der Kindklassen.
- Es ist aber `private` für alle anderen Objekte außerhalb der Vererbungshierarchie.

Ändern wir den Zugriffsmodifizierer der Schuhgröße in der Klasse *Person* von `private` auf `protected`, können die Objekte der Kindklassen diese Objektvariable nutzen.

```
1 protected double schuhgroesse;
```

Methoden vererben

Wir wollen nun in der Lage sein, Objekte der Klassen *Person*, *Student* und *Dozent* auf der Konsole auszugeben.

- Wir können dazu eine Methode *Ausgeben()* in die Klasse *Person* einbauen.

```
1  class Person
2  {
3      // Der Rest fehlt hier
4
5      public void Ausgeben()
6      {
7          Console.WriteLine(Vorname + " " + Nachname);
8      }
9  }
```

Durch die Vererbung ist die Methode dann auf auch auf allen Kindklassen verfügbar.

```
1  var s = new Student("Demir", "Öztürk", 12345);
2  s.Ausgeben();
```

Methoden verbergen und überschreiben

HOCHSCHULE HAMM-LIPPSTADT

Die Methode *Ausgeben()* wird von der Klasse *Person* an alle Kindklassen weiter vererbt.

- Im Gegensatz zu einer Person besitzt der Student aber zusätzlich eine Matrikelnummer.
- Der Dozent zudem ein Lehrgebiet.
- Wenn wir *Ausgeben()* auf auf einem Studenten oder Dozenten aufrufen, sollten diese Infos auch mit ausgegeben werden.

In einer Kindklasse von Person können wir eine neue Methode *Ausgeben()* hinzufügen.

- Wir müssen dann aber festlegen, ob die geerbte Methode durch die neue Methode **verborgen** oder **überschrieben** werden soll.
- Die geerbte Methode wird mit dem Schlüsselwort `new` verborgen.
- Die geerbte Methode wird mit dem Schlüsselwort `override` überschrieben.

Bei beiden Varianten gibt es einen kleinen, aber feinen Unterschied.

Methode verbergen

In der Klasse Student fügen wir eine eigene Methode `Ausgeben()` hinzu.

- Mit dem Schlüsselwort `new` verbergen wir die geerbte Methode.

```
1  public new void Ausgeben()
2  {
3      Console.WriteLine("{0} {1} - {2}", Vorname, Nachname, Matrikelnummer);
4  }
```

Weiterhin kann auf den Objekten aller Klassen die Methode *Ausgeben()* aufgerufen werden.

- Beim Studenten wird nun die neue Methode genutzt.
- Diese gibt auch die Matrikelnummer aus.
- Konvertieren wir den Studenten in ein Objekt vom Typ *Person*, wird wieder nur der Vor- und Nachname ausgegeben.

```
1  var s = new Student("Demir", "Öztürk", 12345);
2  s.Ausgeben();   // gibt die Matrikelnummer mit aus
3  var p = (Person) s;
4  p.Ausgeben();   // gibt nur Vor- und Nachname aus
```

Methode überschreiben

HOCHSCHULE HAMM-LIPPSTADT

Anstelle die geerbte Methode zu verbergen, können wir sie auch überschreiben.

- Dazu muss aber die Methode *Ausgeben()* in der Klasse *Person* zunächst mit dem Schlüsselwort `virtual` gekennzeichnet werden.

```
1  public virtual void Ausgeben()
2  {
3      Console.WriteLine(Vorname + " " + Nachname);
4  }
```

Erneut fügen wir der Klasse Student eine neue Methode *Ausgeben()* hinzu.

- Dieses mal nutzen wir aber das Schlüsselwort `override`.

```
1  public override void Ausgeben()
2  {
3      Console.WriteLine("{0} {1} - {2}", Vorname, Nachname, Matrikelnummer);
4  }
```

Unterschied zwischen Verbergen und Überschreiben

HOCHSCHULE HAMM-LIPPSTADT

Das Schlüsselwort `virtual` aktiviert das sog. **Dynamische Binden**.

- Dadurch wird erst zur Laufzeit die Methode ermittelt, die auf einem Objekt aufgerufen werden soll.

Im Rahmen der Typkonvertierung können wir den Unterschied sehen.

- Erneut konvertieren wir ein Objekt der Klasse *Student* in den Typ seiner Basisklasse.
- Rufen wir nun die Methode *Ausgeben()* auf, wird noch immer die Methode der Ursprungsklasse genutzt.
- Das Objekt hat sich quasi *gemerkt*, welcher Typ es ursprünglich war.

```
1  var s = new Student("Demir", "Öztürk", 12345);
2  s.Ausgeben();   // gibt die Matrikelnummer mit aus
3  var p = (Person) s;
4  p.Ausgeben();   // gibt noch immer die Matrikelnummer mit aus
```

Das Double Dispatch Problem

HOCHSCHULE HAMM-LIPPSTADT

Dynamisches Binden wählt die passende Methode auf einem Objekt erst zur Laufzeit aus.

- Bei überladenen Methoden gibt es (zumindest in C#) diese Dynamik aber nicht.

Im folgenden Beispiel existieren zwei überladene Methoden.

- Eine Methode erwartet ein Objekt vom Typ *Person*, eine andere ein Objekt vom Typ *Student*.

```
1  public static void TueEtwas(Person a) { Console.WriteLine("Person"); }
2  public static void TueEtwas(Student a) { Console.WriteLine("Student"); }
3
4  public static void Main(string[] args)
5  {
6    Person obj = new Student("Alexander", "Stuckenholz");
7    TueEtwas(obj);
8  }
```

Das Double Dispatch Problem

HOCHSCHULE HAMM-LIPPSTADT

Wir stellen fest, dass der Text «Person» ausgegeben wird.

- Wir würden aber erwarten, dass «Student» ausgegeben wird.
- Das Objekt wurde ja als Student erzeugt und dann nachträglich in eine Person umgewandelt.

Dies liegt daran, dass beim Überladen die passende Methode schon zur Übersetzungszeit festgelegt wird.

- Der Compiler sieht beim Übersetzen aber nur einen Parameter vom Typ *Person*.

Wird zur Laufzeit ein abgeleiteter Typ übergeben, kann dies nicht mehr festgestellt werden.

- Dieser Effekt ist als das **Double Dispatch Problem** bekannt.
- Andere Programmiersprachen, z.B. Lisp, zeigen hier ein anderes Verhalten.

Zusammenfassung

HOCHSCHULE HAMM-LIPPSTADT

Wir haben heute gelernt, ...

- aus welchen Gründen Redundanzen im Programmcode schlecht sind.
- wie man mit Vererbung bestimmte Redundanzen vermeiden kann.
- was mit dem Konzept der Vererbung in der objektorientierten Programmierung genau gemeint ist.
- was es bedeutet, wenn eine Kindklasse von einer Basisklasse abgeleitet ist.
- wie man Vererbung in C# umsetzt.
- was Vererbung mit Konstruktoren macht.
- wie sich Vererbung auf Objektbeziehungen auswirkt.
- welche Bedeutung der Zugriffsmodifizierer `protected` besitzt.
- wie man geerbte Methoden verbergen und erweitern kann.
- was Mehrfachvererbung ist und warum sie in C# nicht erlaubt ist.

Aufgabe 1

Pac Man ist ein sehr bekanntes Computerspiel.

- Auf einem Spielfeld sind verschiedene Elemente sichtbar, die untereinander eine Vererbungsbeziehung aufbauen.

Als Basisklasse existiert die Klasse Spielelement.

- Ein Spielelement hat eine X-/Y-Position.
- Wand und Pille sind jeweils Spielelemente.
- Zudem existiert die Klasse Figur, die ein Spielelement ist.
- Eine Figur besitzt die Methode bewege().
- Ein PacMan und ein Geist sind Figuren.

Stellen Sie diese Zusammenhänge grafisch dar.

Aufgabe 2

HOCHSCHULE HAMM-LIPPSTADT

Erstellen Sie die Klassen Spielfeld, Spielelement, Wand, Pille, Figur, PacMan und Geist in C#.

- Das Spielfeld *kennt* eine Menge von Wänden, eine Menge von Pillen, genau einen PacMan und genau vier Geister.
- Wie kann das in C# realisiert werden?

Überschreiben Sie die Methode Bewege() in den Klassen PacMan und Geist.

- Geben Sie jeweils eine Zeichenkette „PacMan bewegt sich“ bzw. „Geist bewegt sich“ auf der Konsole aus.

Welche Methoden werden in welchen Klassen noch gebraucht?

- Ist es noch schwierig, nach dieser Aufteilung das Spiel komplett fertig zu implementieren?

Quellen I

[1] Robert Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. Edition. Upper Saddle River, NJ: Prentice Hall, 1. Aug. 2008. 464 S. ISBN: 978-0-13-235088-4.

[2] Christian Silberbauer. *Einstieg in Java und OOP: Grundelemente, Objektorientierung, Design-Patterns und Aspektorientierung*. 2., akt. u. erw. Aufl. 2020 Edition. Springer Vieweg, 13. Aug. 2020. 168 S. ISBN: 978-3-662-61308-5.