# Allgemeine Konzepte der OOP

Die objektorientierte Programmierung (OOP) ist eine seit vielen Jahren eingeführte Art und Weise des Programmierens, die sich wesentlich von der sog. strukturierten Programmierung unterscheidet.

Die strukt. Programmierung (auch prozedurale Programmierung genannt) trennt zwischen den Daten und den Methoden, die diese Daten manipulieren. Die objektorientierte Programmierung führt diese beiden Aspekte wieder in Form von Klassen zusammen. Die Klasse ist der Container für Daten, die logisch gesehen zusammengehören und einen realen/irrealen Gegenstand unseres Systems abbilden. Sie beinhaltet auch sämtliche Methoden, die diese Daten manipulieren sollen.

## Wichtige Konzepte

* Klasse / Objekt / Instanziierung
* Konstruktor/Destruktor
* Statische Attribute und Methoden
* Attribute / Methoden / Properties
* Geheimnisprinzip (public/private/protected/internal/….)

\* Vererbung

* Basisklasse / Abgeleitete KLasse
* Einfach-, Mehrfachvererbung
* Abstrakte Klasse / Interface
* Überschreiben von Methoden/Polymorphie / Late vs. early Binding
* Beziehungen zwischen Klasse
* Assoziation/Aggregation/Komposition
* using / Navigierbarkeit

### Klasse / Objekt /Instanziierung

Die Klasse ist der grundlegende Begriff in der OOP.

Ausgehend von der jeweiligen Problemstellung versucht man, die zusammengehörenden Informationen in einem gemeinsamen Container, der Klasse eben, zu speichern.

Die Klasse erhält einen Namen, die zu speichernden Informationen bezeichnet man als Attribute der Klasse.

Die Manipulation, d.h. das Schreiben, Lesen und Ändern der Daten, obliegt ebenfalls dem Verantwortungsbereich der Klasse. Neben diesen Aufgaben kann eine Klasse noch weitere Fähigkeiten besitzen. Diese Fähigkeiten werden in der Klasse durch Methoden definiert.

Eine Klasse besteht deshalb zumindest aus 3 Bereichen.

1. Dem Klassennamen
2. Der Liste der Attribute
3. Der Liste der Methoden

Ein typisches Beispiel für eine Klasse könnte wie folgt aussehen:

|  |  |
| --- | --- |
| Quellcode | Klassendiagramm |
|  |  |

Neben den wirklichen Fähigkeiten, im obigen Beispiel sprechen, muss eine Klasse häufig über Verwaltungsmethoden verfügen, die eine Manipulation der internen Attribute ermöglichen. Man nennt diese Methode häufig getter/setter-Methoden. Da ein Methodenname lediglich einmal genutzt werden darf, muss man deshalb häufig 2 Methodennamen benutzen.

Innerhalb der Klasse gibt es häufig eine spezielle Methode, den sog. Konstruktor. Sie hat den gleichen Namen wie die Klasse selbst und sie kann in verschiedenen Variationen vorliegen. Beim Erzeugen eines Objektes (s.u.) wird diese Methode als Erstes aufgerufen. Wird diese Methode nicht innerhalb der Klasse definiert, so wird ein sog. Standardkonstruktor benutzt. Er besteht lediglich aus dem Methodennamen ohne irgendwelche Parameter. Werden eigene Konstrukoren geschrieben, so muss der Standardkonstruktor explizit definiert werden, sonst ist er nicht mehr vorhanden.

In .NET gibt es deshalb noch eine 3. Art der Methoden, die sog. Properties. Hierbei handelt es sich um Methodennamen, die sowohl für den schreibenden als auch lesenden Zugriff geeignet sind. Eine Implementierung für das obige Beispiel könnte wie folgt ausehen:

|  |  |
| --- | --- |
| Quellcode | Klassendiagramm |
|  |  |

### Benutzung von Objekten / Konstruktor

Die Klasse definiert als Bauplan lediglich die Art der gehaltenen Informationen bzw. die Methoden, die zur Manipulation der Daten bzw. zur Funktionalität der Klasse notwendig sind. Doch wie kann man nun diese Klasse nutzen? Da wir ja Informationen zu einem realen Gegenstand des Systems erheben wollen, müssen wir den Bauplan der Klasse einem realen Objekt zuordnen. Nur ein reales Objekt kann Daten speichern. Dieser Vorgang wird Instanziierung genannt und läuft in folgenden Schritten ab:

* Deklaration einer Variablen vom Typ der Klasse
* Instanziierung des Objektes mit Hilfe des new-Operators.

class Programm  
{  
  
 public static void Main(string args)  
 {  
 //Deklaration  
 Person steinam;  
  
 //Instanziierung  
 //Standardkonstruktor wird genutzt  
 steinam = new Person();  
  
 //das geht auch in einem  
 Person sierl = new Person();  
  
 //hier wird der selbst geschriebene Konstruktor genutzt  
 Person wallner = new Person("Wallner");  
 wallner.sprechen("I like .net");  
 steinam.setName("Steinam");  
 //Aufruf des set-Teils der Property Familienname  
 sierl.Familienname = "Sierl";  
  
 //Aufruf des get-Teils der Property Familienname  
 string FamName = steinam.Familienname;  
   
 //Objekt steinam wird zerstört  
 //Aufruf des Destruktors  
 steinam = null;  
 }  
}

Der new-Operator erzeugt ein sog. Objekt der Klasse. Dieses Objekt ist einzigartig und ist nun in der Lage, die Informationen aufzunehmen und zu verarbeiten. Bei allen Instanziierungen wird immer ein Konstruktor aufgerufen. Er lautet wie der Name der Klasse und kann überladen sein, d.h. in verschiedenen Versionen existieren. Durch ihn ist das Objekt in der Lage, Zustände seiner Variablen bei der Erzeugung zu kontrollieren. Definiert man eigene Konstruktoren, so muss der parameterlose Standardkonstruktor ebenfalls angegeben werden, wenn man ihn zur Verfügung stellen will.

Erst nach dem Erzeugen kann man nun die Fähigkeiten des Objektes benutzen, d.h. man kann die Methoden der Klasse benutzen. Merke: Methoden werden auf Klassenebene definiert, aber auf Objektebene genutzt ! (Es gibt aber eine Ausnahme ! Welche ?)

Parallel zum Konstruktor gibt es den sog. Destruktor. Er wird durch den folgenden Aufruf beschrieben:

~Klassenname()

Das Zerstören eines Objektes wird durch das NULL-Setzen der Objektreferenz bewirkt. Der tatsächliche Zeitpunkt des Zerstören eines Objektes hängt von mehreren Faktoren ab

* Anzahl der noch gültigen Referenzen
* Tatsächliches Freigeben des Speichers durch die Garbage Collection

Es kann deshalb nicht genau vordefiniert werden, ob und wann durch das Löschen einer Objektreferenz die Destruktor-Methode aufgerufen wird.

Beispiel:

public class Person  
{  
 public Person()  
 {  
 anzahl += 1;  
 }  
   
 ~Person()  
 {  
 Console.WriteLine("Ende");  
 Console.ReadLine();  
 }  
   
}  
  
  
class Program  
{  
 public static void Main(string[] args)  
 {  
 //2 Objekte werden erzeugt   
 Person steinam = new Person();   
 Person sierl = new Person();  
   
 //Ein Objekt wird zerstört  
 steinam = null;  
  
  
 //Wieviele Objekte werden angelegt ?  
 //Wieviele Objekte werden zerstört  
 Person meier = new Person();  
 Person mueller = meier;  
 meier = null;  
   
   
   
 Console.Write("Press any key to continue . . . ");  
 Console.ReadKey(true);  
 }  
}

### Statische Attribute/Methoden

Um in OOP arbeiten zu können, sind offenbar immer zunächst Objektinstanziierungen notwendig. Dies ist aber manchmal lästig, weil wir eigentlichnur eine Funktionalität brauchen bzw. redundante Informationen speichern wollen, die für alle gleich sind.

So könnte es z.B. sein, dass Schüler einer Klasse die Anzahl der Mitschüler in ihrer Klasse kennen sollen. Bei 32 Schülern würde dies bedeuten, dass 32 Objekte die gleiche Information in einer lokalen Instanzvariable halten müssten! Noch schlimmer, bei Änderungen der Schülerzahl durch Hinzukommen / Weggehen neuer Schüler müsste in allen Objekten dieser Wert geändert werden !

Für solche Fälle kennt die OOP die Möglichkeit sog. statischer Attribute bzw. Methoden. Diese werden bei der Klasse gehalten und durch das Schlüsselwort static deklariert.

Der Zugriff auf diese Werte ist sowohl über jedes Objekt als auch über die Klasse an sich möglich.

Die Aktualisierung solcher Informationen beim Zerstören solcher Objekte kann aber problematisch werden.

Folgendes Beispiel verdeutlicht die Situation:

Bestimmen Sie, welche Ausgabe jeweils von Console.WriteLine() produziert wird!

public class Person  
{  
 private static int anzahl=0;  
   
   
 public Person()  
 {  
 anzahl++;  
 }  
 ~Person()  
 {   
 //Zähler zurücksetzen  
 //wird das klappen  
 anzahl--;  
 Console.WriteLine("Ende");  
 Console.ReadLine();  
  
 }  
   
 //jedes Objekt kennt die Zahl der Personen  
 public int getAnzahl()  
 {  
 return anzahl;  
 }  
  
 //selbst die Klasse kann direkt gefragt werden  
 //ohne ein Objekt benutzen zu müssen  
 public static int getZahl()  
 {  
 return anzahl;  
 }  
}  
  
  
class Program  
{  
 public static void Main(string[] args)  
 {  
 //Statische Methode der Klasse wird genutzt  
 Console.WriteLine(Person.getZahl());  
   
 //Person wird erzeugt und nach Anzahl gefragt  
 Person steinam = new Person();  
 Console.WriteLine(steinam.getAnzahl());  
   
 //zweite Person erhöht die Zahl  
 Person sierl = new Person();  
   
 //Sierl kennt die neue Anzahl  
 Console.WriteLine(sierl.getAnzahl());  
  
 //kennt sie auch steinam  
 Console.WriteLine(steinam.getAnzahl());  
   
 //steinam wird zerstört   
 steinam = null;  
   
 Console.WriteLine(Person.getZahl());  
 Console.WriteLine(sierl.getAnzahl());   
 Console.Write("Press any key to continue . . . ");  
 Console.ReadKey(true);  
 Console.WriteLine(sierl.getAnzahl()); //hoffentlich 1 (GC)  
 }  
}

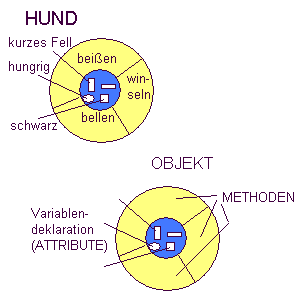
### Geheimnisprinzip

So wie Objekte im realen Leben auch nicht jedes Geheimnis nach draußen preisgeben, so gilt dies auch in der OOP. Das Objekt sollte prinzipiell den Zustand seiner Attribute versteckt halten, d.h. keinen direkten Zugriff auf seine Attribute erlauben.

Durch das Bereitstellen von entsprechenden getter/setter-Methoden bzw. Properties kann das Objekt den Zugriff auf seine Attribute kontrollieren.

Der generelle Zugriffsmöglichkeit auf Attribute und Methoden wird in der OOP über die Sichtbar-keiten definiert. Diese stehen vor dem Attribut bzw. der Methode und definieren den möglichen Zugriff von außerhalb auf die Attribute und Methoden. Falls keine Aussage im Quellcode getroffen wird, sol gelten die default-Sichtbarkeiten der jeweiligen Programmiersprachen, die sich durchaus unterscheiden können. Gängige Definitionen von Sichtbarkeiten sind:

* public: Öffentlich sichtbar, von überall aufruf- und damit manipulierbar
* private: Nur innerhalb des Objektes selbst benutzbar
* protected/internal: Nur innerhalb des gleichen Namespaces bzw. über Vererbungsmechanismen sichtbar.
* Ask your favorite programming language about more …..

In den Bildern sind die Variablen des Objektes im Zentrum, umschlossen von den Methoden. Dies deutet grafisch an, daß Variablen gegen Zugriffe von außen geschützt sind, d.h. Programme können nicht direkt auf die Variablen des Objektes zugreifen, son-dern müssen die entsprechenden Methoden auf-rufen. Dieser Schutz wird Kapselung genannt. Mit Kapselung werden oft Implementationsdetails versteckt.

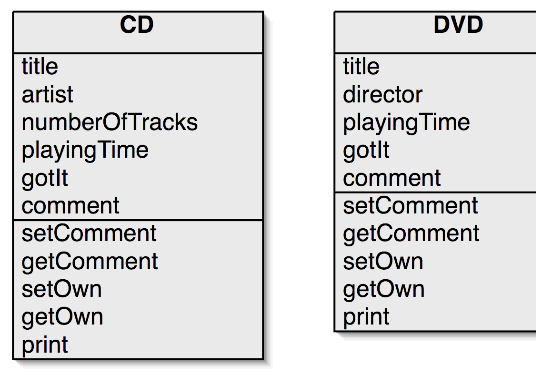
Ein Objekt hat eine nach außen definierte Schnittstelle, mit der andere Objekte kommunizieren können. Das Objekt kann die privaten Informationen und Methoden ändern, ohne daß andere Objekte betroffen sind. Zudem kann ein Objekt problemlos herumgereicht werden.

## Teil 2

## Der 2. Teil setzt sich mit folgenden Konzepten auseinander:

* Vererbung
  + Basisklasse / Abgeleitete KLasse
  + Einfachvererbung
  + Überschreiben von Methoden

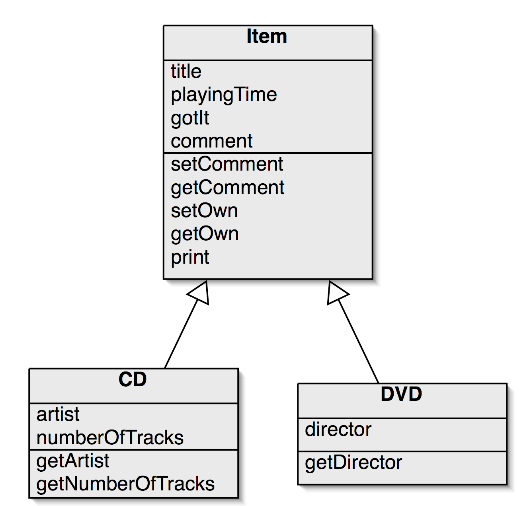
# Vererbung

Beim Erstellen von Klassen wird man häufig Abhängigkeiten/Gemeinsamkeiten zwischen Klassen erkennen. Irgendwie gehören sie zusammen aber doch auch wieder nicht. Die OOA löst diese Problematik durch das Konzept der Vererbung.

Es bedeutet, dass Klassen Attribute und Fähigkeiten anderer Klassen über-nehmen und gleichzeitig erweitern können.

Häufig wird man auch erst im Laufe der Analyse eine bestehende Klasse in mehrere Klassen unter-teilen wollen.

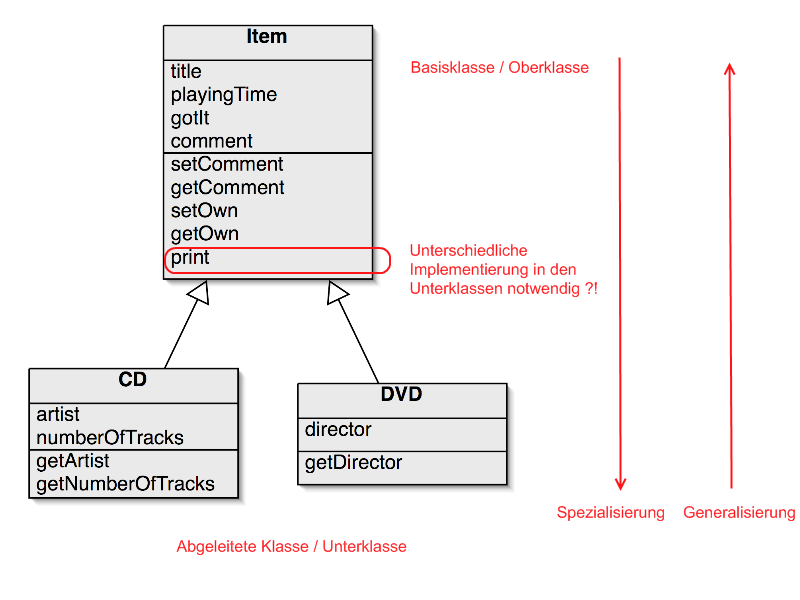
Aufgabe: Wie könnte man die dargestellten Klassen besser organisieren ?

Die Klasse CD und DVD verfügen über gemeinsame Attribute und Metho-den. Sie werden in einer Oberklasse Item ausgelagert. Zusätzliche Informa-tionen und Fähigkeiten bleiben in den spezialisierten Unterklassen.

Im Zusammenhang mit Vererbung existieren einige Begriffe:

* **Basisklasse**: Die Klasse, die alle Gemeinsamkeiten aufnimmt (in unserem Fall Item)
* **Abgeleitete Klasse / Unterklasse**: Die Klassen, die spezielle Attribute und Methoden aufnehmen (in unserem Fall CD und DVD)
* **Generalisierung**: Untersuchung des Vererbungsaspektes von den speziellen Klassen zu den allgemeinen Klassen

**Spezialisierung**: Untersuchung des Vererbungsaspektes von den allgemeinen Klassen zu den speziellen Klassen

Es sind folgende Dinge zu beachten:

* Informationen werden in Ober- und Unterklassen gehalten. Damit wird Redundanz vermieden.
* Die Unterklassen brauchen aber Zugriffsmöglichkeiten auf Attribute und Methoden der Oberklasse
* Die Unterklassen halten über den Vererbungsmechanismus den Vertrag der Oberklasse. Konkrete Instanzen der Unterklassen können überall dort benutzt werden, wo Instanzen der Oberklasse erwartet werden. In einem solchen Falle ist dann aber lediglich der Zugriff auf die Schnittstelle der Oberklasse möglich. Die Unterklassen sind somit vom gleichen Typ wie die Oberklasse. Eine DVD ist damit auch ein Item.
* Die Oberklasse hat im Allgemeinen keine Kenntnis darüber, dass es abgeleitete Unterklassen gibt.

## Implementierung des Beispiels in C#

Bei der Implementierung des obigen Beispiels ist zu beachten, dass

* insgesamt 3 Klassen erzeugt werden müssen
* die Unterklassen Zugriff auf Attribute und Methoden der Oberklasse haben müssen und dementsprechend Zugriffsmodifizier (internal) gewählt werden müssen
* gewisse Methoden in den Unterklassen überschrieben werden müssen. Hierzu muss in der Unterklasse mit dem Schlüsselwort virtual, in den Unterlassen mit dem Schlüsselwort override gearbeitet werden.

### Implementierung der Oberklasse

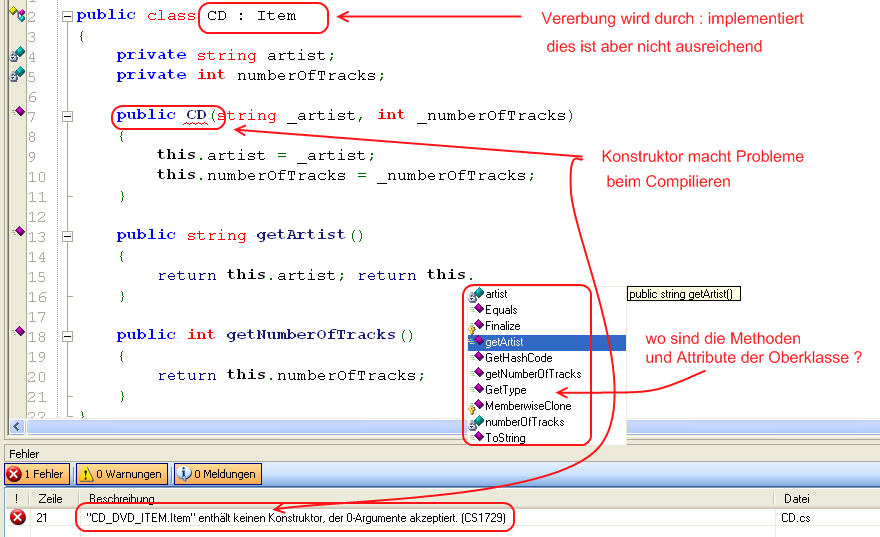
|  |  |
| --- | --- |
| Quellcode | Kommentar |
|  | Die Klasse Item hält 4 Attribute und 2 getter/setter-Methoden. Der Konstruktor wird verwendet, um Titel und Spielzeit zu setzen. Die print()-Methode gibt den Titel und die Spielzeit aus. |

### Implementierung der Unterklasse CD

Eine erste, naive, Implementierung sieht wie folgt aus:

public class CD : Item  
{  
 private string artist;  
 private int numberOfTracks;  
   
 public CD(string \_artist, int \_numberOfTracks)  
 {  
 this.artist = \_artist;  
  
  
 this.numberOfTracks = \_numberOfTracks;  
 }  
   
 public string getArtist()  
 {  
 return this.artist;  
 }  
   
 public int getNumberOfTracks()  
 {   
 return this.numberOfTracks;  
 }  
}

Dies führt aber zu folgenden Problemen:



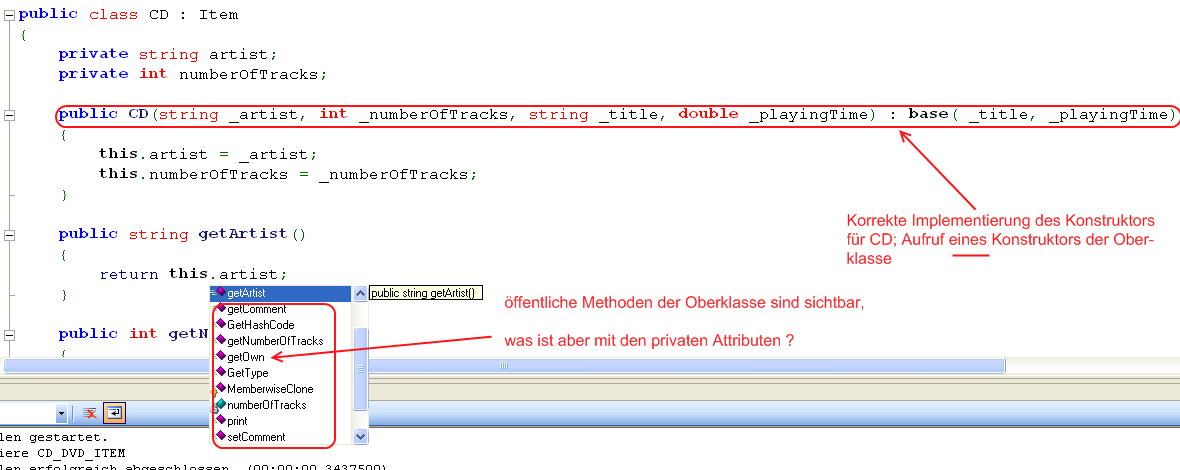
#### Konstruktorproblematik

Beim Erzeugen einer Unterklasse wird es aufgrund der Vererbung auch notwendig, eine Instanz der Oberklasse zu erzeugen. Da diese jedoch in unserer Implementierung keinen parameterlosen Konstruktor zulässt, erzeugt der Compiler einen Fehler. Hätten wir einen parameterlosen Konstruktor, hätte der Compiler keinen Fehler gemeldet.

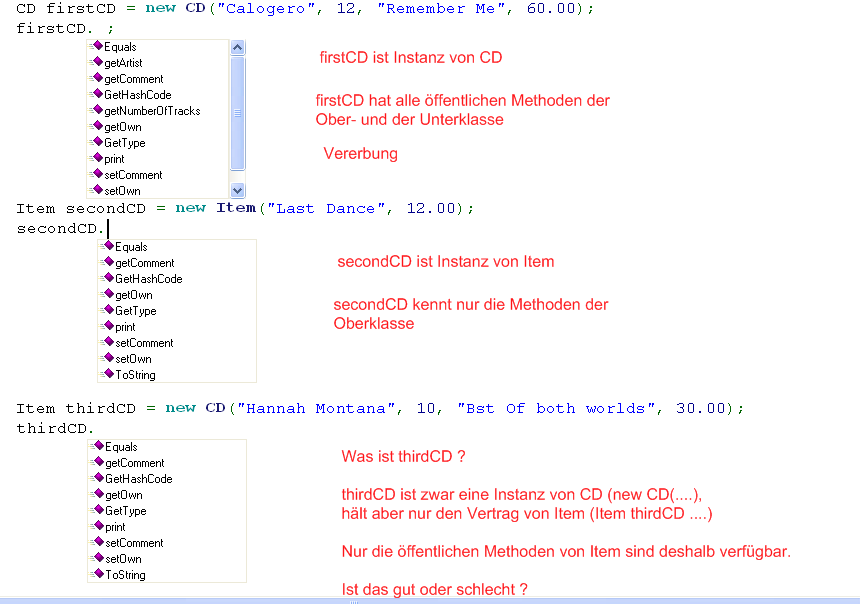
Daran ist prinzipiell auch nichts Verwerfliches, da wir in der Oberklasse über den Konstruktor das Setzen von Zuständen implementiert haben, die für unser Verhalten wichtg sind (Titel und Spielzeit). Auch eine CD sollte über diese Informationen verfügen. Der Konstruktor von CD muss deshalb diese Informationen an den Konstruktor der Oberklasse Item weiterreichen.

Dies erfolgt durch das Weiterleiten im Konstruktoraufruf:

public class CD : Item  
{  
 private string artist;  
 private int numberOfTracks;  
   
 //Weiterleiten an den Konstruktor der Oberklasse //Konstruktorsignatur der Oberklasse   
 public CD(string \_artist, int \_numberOfTracks, string \_title, double \_playingTime) : base( \_title, \_playingTime)  
 {   
 this.artist = \_artist;  
 this.numberOfTracks = \_numberOfTracks;  
 }



#### Vererbung als Vertrag

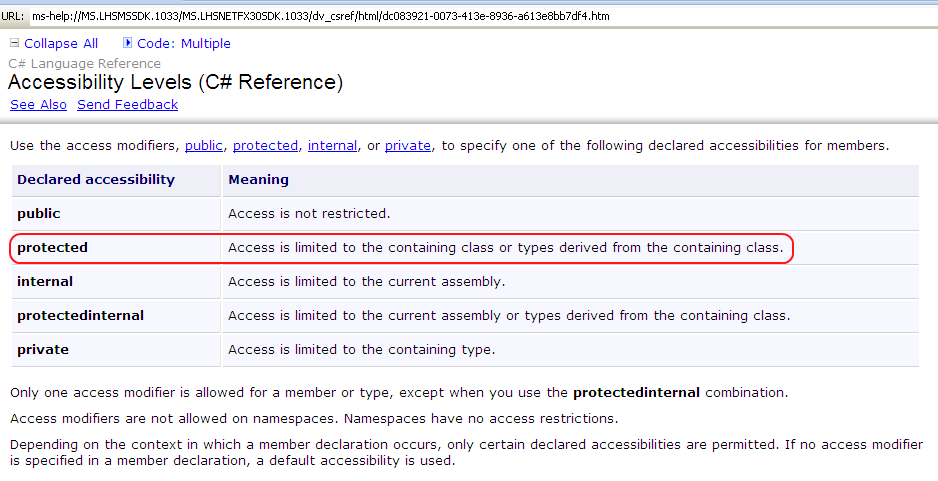


#### Zugriff auf Attribute und Methoden der Oberklasse

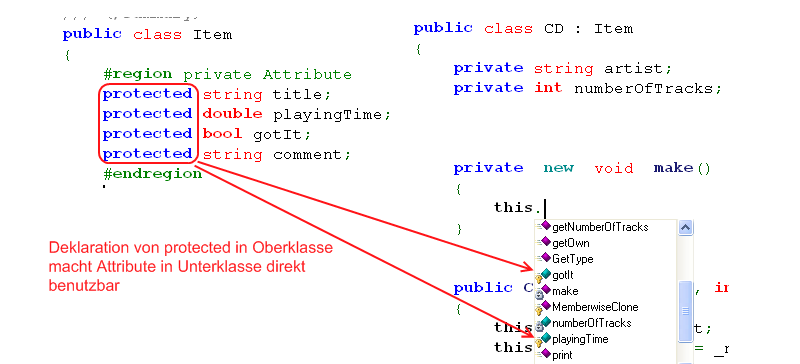
Durch die Vererbung wird zunächst der Zugriff auf öffentliche Methoden und Attribute gewährt. Private Attribute und Methoden sind durch das Prinzip der Kapselung weiter geschützt.

Will man den Unterklassen Zugriff auf diese Elemente gewähren, so müssen die Zugrifsmodifizierer der Oberklasse geändert werden.

In C# stehen folgende Modifizierer zur Verfügung:

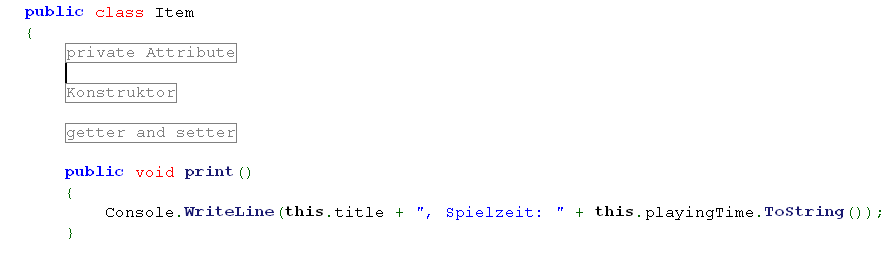


In unserem Beispiel könnten die bisher als private deklarierten Attribute der Oberklasse mit dem Schlüsselwort protected ersetzt werden.



#### Überschreiben von Methoden der Oberklasse

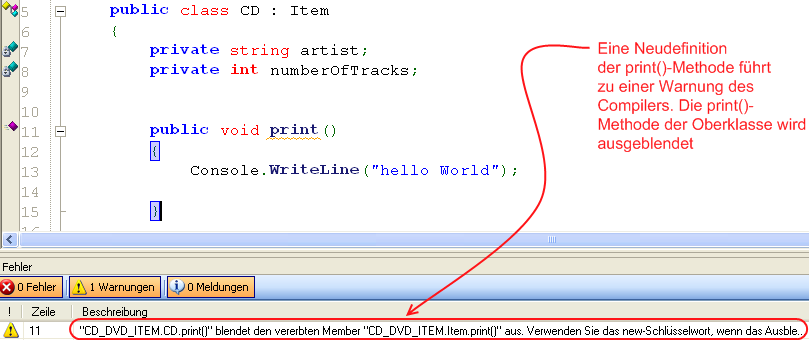
Die Methode print() der Klasse Item gibt zur Zeit den Titel sowie die Spielzeit aus.



Dieses Verhalten kann eventuell nicht das sein, was Instanzen der Klasse CD möchten. Doch wie kann die Ausgabe für Objekte der Klasse CD geändert werden, ohne eine andere Methode benutzen zu müssen. Um es allgemeiner auszudrücken:

Wie kann eine Unterklasse ein anderes Verhalten als das der Oberklasse implementieren?

Die Lösung besteht im Neudefinieren der Methode in den Unterklassen.



Um die Warnung des Compilers zu umgehen, gibt es zwei Möglichkeiten:

* Überschreiben mit Hilfe von new

public class CD : Item  
{  
 private string artist;  
 private int numberOfTracks;  
   
 public new void print()  
 {  
 Console.WriteLine("hello World");  
 }

Als Nebeneffekt ist jedoch zu bemerken, dass bei folgendem Quellcode nicht die Implementierung der Unterklasse aufgerufen wird.

Item newCD = new CD(....);  
newCD.print()

* Benutzen des Schlüsselwortes override

Dies setzt allerdings voraus, dass in der Oberklasse die jeweilige Methode als **virtual, abstract oder override** deklariert wurde.

public class CD : Item  
{  
 private string artist;  
 private int numberOfTracks;  
   
 //Zunächst rufen wir die print()-Methode der Oberklasse auf  
 //und fügen anschließend unsere eigene Implementierung hinzu  
 public override void print()  
 {  
 base.print();  
 Console.WriteLine("Artist: " + this.artist);  
 Console.WriteLine("Number of Tracks: " + this.numberOfTracks);  
   
 }  
 ....  
}  
  
public class Item  
{  
 ....  
 public virtual void print()  
 {  
 Console.WriteLine(this.title + ", Spielzeit: " + this.playingTime.ToString());  
 }  
}

Der 3. Teil setzt sich mit folgenden Konzepten auseinander:

* Vererbung

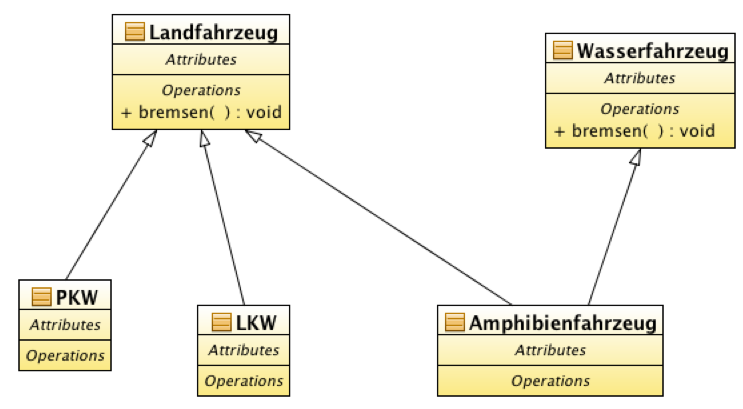
\* Mehrfachvererbung

* Abstrakte Klasse / Interface
* Überschreiben von Methoden/Polymorphie / Late vs. Early Binding

—————-

# Mehrfachvererbung

Mehrfachvererbung erweitert die grundsätzliche Vererbung um die Fähigkeit, von mehreren Oberklassen gleichzeitig ableiten zu können. Damit ist aber auch bereits die Problematik vorgegeben.

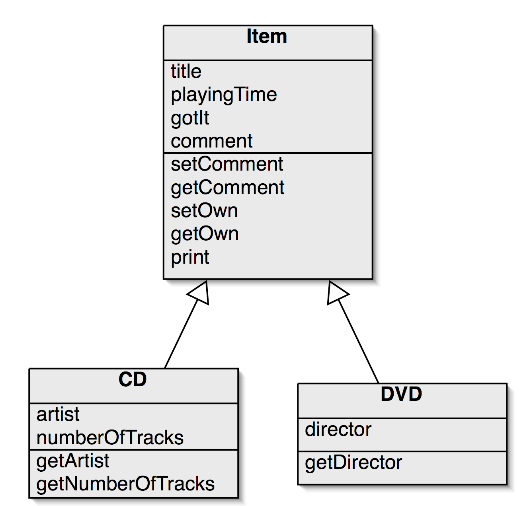


Welche Methode bremsen() sollen Amphibienfahrzeuge ausführen ??

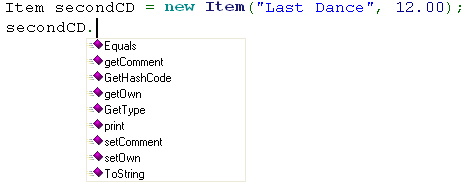
Aufgrund dieser Nebeneffekte haben sich die meisten Programmiersprachen vom Konzept der Mehrfachvererbung abgewendet, ohne jedoch die prinzipiellen Vorteile der Mehrfachvererbung mit Hilfe von Interfaces nicht zu implementieren.

# Abstrakte Klasse

Die Implementierung des vorherigen Beispiels CD/DVD hatte einen großen Nachteil:



Es gibt eigentlich keine Instanzen von Item, sie waren aber jederzeit erzeugbar.

****

Das Konzept der Abstrakten Klasse kann dieser Problematik begegnen.

* Definition der Klasse oder einer Methode als abstract
* Damit ist eine Instanziierung dieses Typs ausgeschlossen
* Klasse kann dennoch konkrete Implementierungen von Methoden sowie Attribute besitzen
* Methoden, von denen bekannt ist, dass Sie die Unterklassen überschreiben müssen, werden als abstrakt definiert. Sie bestehen lediglich aus der Methodensignatur.
* Damit müssen ableitende Unterklassen diese Methoden implementieren, ansonsten können keine Instanzen der Unterklassen gebildet werden.

Eine Implementierung der Klasse Item als abstrakte Klasse kann wie folgt aussehen.

using System;  
  
namespace CD\_DVD\_ITEM  
{  
 public abstract class Item  
 {  
 #region private Attribute   
 protected string title;  
 protected double playingTime;  
 protected bool gotIt;  
 protected string comment;  
 #endregion  
   
 #region Konstruktor  
 public Item(string \_title, double \_playingTime){  
 this.title = \_title;  
 this.playingTime = \_playingTime;  
 }  
 #endregion  
   
 #region getter and setter  
   
 public string getComment(){  
 return this.comment;  
 }  
   
 public void setComment(string \_comment){  
 this.comment = \_comment;  
 }  
   
 public void setOwn(bool \_own){  
 this.gotIt = \_own;  
 }  
   
 public bool getOwn(){  
 return this.gotIt;  
 }  
 #endregion  
   
 public abstract void print();  
 }  
}

Da die Methode print() jetzt als abstract definiert wurde, müssen die konkreten Klassen CD und DVD diese Methode implementieren.

public class CD : Item  
{  
 private string artist;  
 private int numberOfTracks;  
   
 public override void print()  
 {  
 //das geht jetzt natürlich nicht mehr  
 //base.print();   
 Console.WriteLine("Artist: " + this.artist);  
 Console.WriteLine("Number of Tracks: " + this.numberOfTracks);  
 }

# Interface

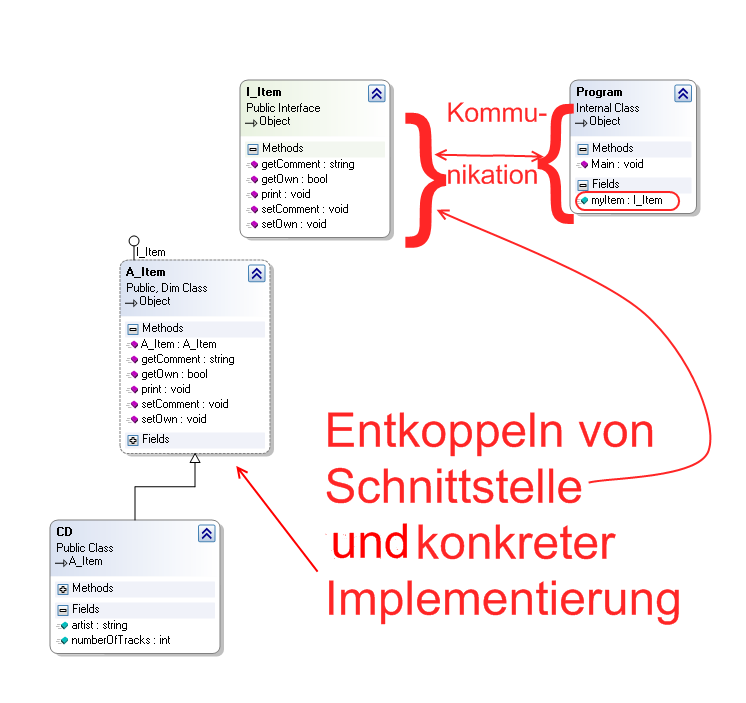
Das Konzept des Interfaces führt die Idee der abstrakten Klasse weiter. Während die abstrakte Klasse durchaus in der Lage ist, eigene Attribute und Implementierungen von Methoden zu besitzen, reduziert sich das Interface im Prinzip lediglich auf die Deklaration von leeren Methodensignaturen. Diese werden dann von konkreten Klassen implementiert. Eine Implementierung mehrerer Inter-faces ist möglich (siehe Mehrfachvererbung).

Interfaces sind dann das geeignete Konzept, wenn es darum geht, die Schnittstellen von Klassen herauszuarbeiten, d.h. die Frage zu beantworten:

Wer kommuniziert mit mir über welche Methoden ?

Die Implementierung der bisherigen (abstrakten) Klasse Item zu einem Interface sieht wie folgt aus:

using System;  
  
namespace CD\_DVD\_ITEM  
{  
 public interface I\_Item  
 {  
 string getComment();  
 void setComment(string \_comment);  
 void setOwn(bool \_own);  
 bool getOwn();  
 void print();  
 }  
}



# Überschreiben von Methoden/Polymorphie / Late vs. Early Binding

Polymorphie bedeutet „Vielgestaltigkeit“. In der OOP wird der Begriff zunächst dazu genutzt, um darzustellen, dass verschiedene Klassen gleiche Methoden besitzen können. Diese Methoden können aber veschieden implementiert sein und damit eben auch verschiedene Dinge tun.

Unter Late/Early Binding versteht man den Zeitpunkt, wann der Compiler bzw. die Runtime-Umgebung festlegen kann, welche Methode denn nun tatsächlich aufgerufen werden soll.

Die wahre Bedeutung der Polymorphie und des Late/Early Bindings erschließt sich durch die Vererbung und dem Nutzen von Klassen der Vererbungshierarchie von den jeweiligen Klienten.

Wir greifen das Beispiel der CD/DVD - Applikation auf und erweitern es um die Klasse DVD.

namespace CD\_DVD\_ITEM  
{  
 class Program  
 {  
 static I\_Item[] myItems = new I\_Item[10];  
 static CD myCD1;  
 static DVD myDVD1;  
   
 public static void Main(string[] args)  
 {  
 //Early Binding  
 //Bereits der Compiler kann erkennen, welche print()-Methode  
 //aufgerufen werden soll  
 //Konkreter Typ ist bekannt  
 myCD1 = new CD("Calogero", 12, "Remember Me", 60.00);  
 myCD1.print();  
   
 myDVD1 = new DVD("Steinam", "Hannibal Lector", 2.45);  
 myDVD1.print();  
 myItems[0] = myCD1;  
 myItems[1] = myDVD1;  
   
 //Vorbereiten des LateBindings  
 //Zufälliges Belegen der Slots des Arrays  
 Random number = new Random(1234);  
 for(int y = 2;y<=9;y++)  
 {  
 int zahl = number.Next(0,10);  
 if(zahl <=5)  
 {  
 myItems[y] = myCD1;  
 }  
 else  
 {  
 myItems[y] = myDVD1;  
 }  
 }   
   
 //Late-Binding  
 //Der Compiler kann eigentlich nicht erkennen, welche print()-Methode   
 // aufgerufen werden kann.Erst zur Laufzeit kann dies entschieden werden.   
 for(int i = 0; i<=9;i++)  
 {  
 myItems[i].print();  
 }  
 Console.ReadKey();  
 }  
 }  
}

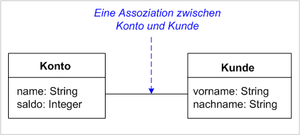
# Beziehungen zwischen Klassen

Während die Vererbung eine Typ-basierte-Beziehung darstellt, sind Beziehungen zwischen Klassen nicht nur auf diese Art herstellbar.

Objekte kommunizieren per Nachrichten mit anderen Objekten; häufig halten sie sich auch andere Obekte zur Erfüllung bestimmter Aufgaben.

## Assozation

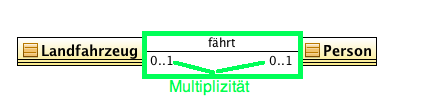
Unter Assoziation versteht man eine Beziehung zwischen zwei oder mehreren Typen. Untenstehende Abbildung zeigt dies in seiner einfachsten Form, einer Linie zwischen zwei Klassen. Sie drückt eine „Sich-Kennen“-Eigenschaft zwischen Objekten der jeweiligen Klasse aus.



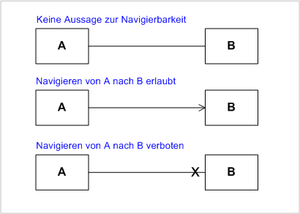
Die Linie kann durch verschiedene Elemente in ihrem Informationsinhalt erweitert werden.

### Multiplizität

Wie im Entity-Relationship-Modell kann die Assoziation um die Angabe erweitert werden, wieviele Objekte einer Seite jeweils mit einem Objekt der der anderen Seite in Beziehung stehen. Dies wird durch eine Zahl bzw. ein \* oder einer Min-Max-Notation zum Ausdruck gebracht.



### Navigierbarkeit

Eine Assoziation bildet eine Art Brücke zwischen zwei Typen: startet man bei der Instanz des einen beteiligten Typs kann man über eine Objektbeziehung zur Instanz des zweiten Typs navigieren. Die Navigierbarkeit von Assoziations-enden wird mit Hilfe eines > dargestellt und kann durch ein x eingeschränkt werden. Dabei unterscheidet man drei Arten, wie die Navigierbarkeit festgelegt werden kann:

* Keine Aussage (keine Pfeile)
* Erlaubte Navigation
* Nicht erlaubte Navigation

### Rollen

Gerade wenn es mehrere Assoziationen zwischen zwei Klassen gibt, kann der Informationsgehalt durch die Darstellung der Rolle erweitert werden. Die Rolle bezeichnet die Funktion, die ein Teilnehmer der Beziehung gegenüber der anderen Seite spielt. Sie wird an das jeweilige Assoziationsende geschrieben.

Im vorliegenden Beispiel gibt es zwei Beziehungen zwischen Bestellung und Adresse. Durch die unterschiedlichen Rollennamen kann man sie gut unterscheiden. Die beiden kleinen Dreiecke neben dem Assoziationsnamen unterstützen die Leserichtung.

**Adresse [ist] Rechnungsadresse für Bestellung**

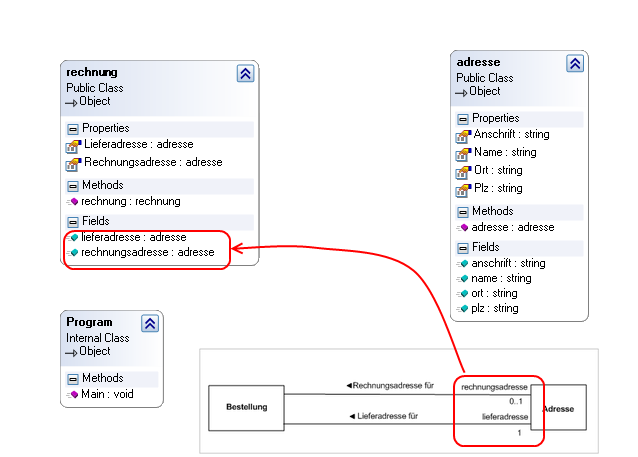
### Implementierung der Assoziation in einer Programmiersprache

Obenstehendes Beispiel von Adresse und Rechnung soll als Ausgangsgrundlage einer Implementierung dienen.

Folgende Grundüberlegungen können vorgenommen werden:

* Da keine Navigation angegeben ist, nehmen wir an, dass die Rechnung die Adresse kennt, nicht jedoch umgekehrt
* Ein Rechnungsobjekt hat 0 oder eine Rechnungs- bzw. Lieferadresse, d.h. sie muss jeweils eine Referenz auf maximal eines dieser beiden Adressen haben; dies kann durch eine entsprechende Referenzvariable erreicht werden.
* Die Rollennamen sind geeignete Namen für die Namen der Referenzvariablen.

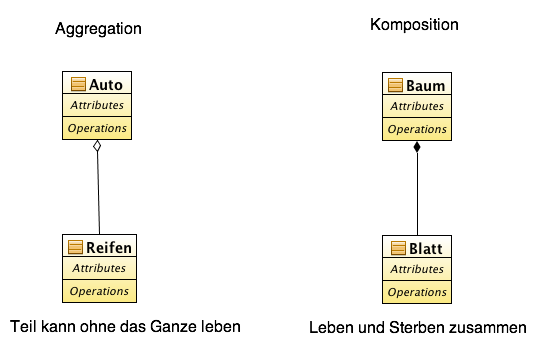
Aufgabe: Implementieren Sie untenstehende detaillierte Klassendiagrammansicht

****

## Aggregation/Komposition

Diese beiden Begriffe spiegeln eine besondere Art von Beziehungen zwischen Objekten wieder, nämlich sog. Teil-Ganzes-Beziehungen zwischen den Klassen. Es wird damit auch eine gewisse Abhängigkeit zwischen den Klassen ausgedrückt.

So ist das Ganze zur Erfüllung seiner Aufgaben von den Teilen abhängig (Aggregation), während zusätzlich dazu ein Teil ohne das Ganze nicht existieren kann (Komposition). In beiden Fällen kann das Ganze seine Aufgaben nur durch die Teile erfüllen, das Ganze wird damit erst durch seine Teile zu einem sinnvollen Objekt.



Aggregation und Komposition im Vergleich

Die Frage, ob eine Beziehung als Aggregation oder Komposition gestaltet werden soll, hängt häufig vom jeweiligen Problem ab und kann nicht immer gleich beantwortet werden.

Die Entscheidung für Aggregation und Komposition hat aber gewisse Konsequenzen für die Implementierung, insbesondere bei der Komposition:

* Das Ganze darf bei der Komposition die alleinige Referenz auf das Teil besitzen
* Deshalb muss das Ganze das Teil selbst erzeugen
* Das Ganze darf keine Referenzen auf das Teil an Klienten weitergeben, da sonst die Kontrolle verloren gehen kann.