

Grundlagen Programmierung

Theorie-Skript Teil 2

Anwendungsentwicklung

Objektorientierte Programmierung



Name:

Klasse:

Datum:

Inhaltsverzeichnis:

1	Motivation für Objektorientierte Programmierung (OOP)	3
2	Anwendungsfalldiagramme (Use Cases)	5
3	Klassen / Objekte	6
3.1	Definition und Verwendung	6
3.2	Beispielklasse Auszubildender	7
3.3	Methoden allgemein	8
3.4	Konstruktor	9
3.5	Destruktor	9
3.6	Set- und get-Methoden	9
4	Beziehungen zwischen Klassen (Objekten)	10
4.1	Übersicht Klassenbeziehungen	10
4.2	Klassenbeziehungen im Detail	11
	Vererbung	11
	Schnittstellen	12
	Assoziationen	12
	Aggregation	15
	Komposition	16

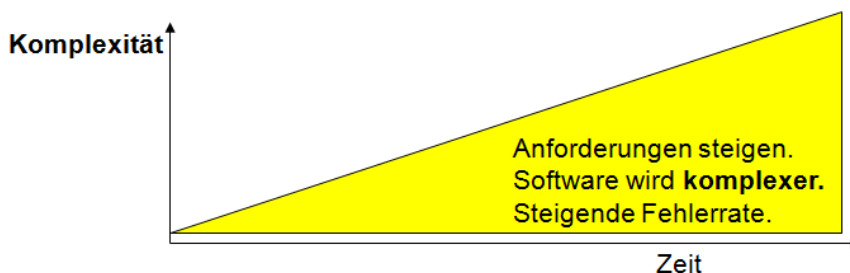
1 Motivation für Objektorientierte Programmierung (OOP)

Die Idee der Objektorientierung ist über 30 Jahre alt und fast ebenso lange liegt die Entwicklung Objektorientierter Programmiersprachen zurück. Zu den bekanntesten Sprachen gehören C++, Visual C# und Java.

Prozedurale Programmierung: Trennung von Daten und Funktionen

Im Allgemeinen erfolgt die Unterteilung von Informationen in zwei voneinander unabhängigen Kategorien: Funktionen und Daten. Die meisten Aufgaben lassen sich mit dieser Betrachtungsweise bewältigen. Diese strukturierten Software-Entwicklungsmethoden finden aber schnell ihre Grenzen, wenn die Komplexität der Software zunimmt. Wenn z.B. eine Variable zusätzlich benötigt wird, müssen Funktionen, die speichern, erfassen oder drucken, angepasst werden. Ab einer bestimmten Größe der Software ist die Suche welche Funktionen wo anzupassen sind aber Sisyphusarbeit. Die strikte Trennung zwischen Daten- und Funktionssicht macht die Beschreibung und Realisierung komplexer Software nicht mehr oder nur zu hohen Kosten beherrschbar.

Motivation für Objektorientierung



Wesentliche Anforderungen an die Entwicklung objektorientierter Programmiersprachen sind:

Anforderungen	Vorteile
Software muss teilbar sein.	Arbeitsteiliges Arbeiten und Wiederverwendbarkeit wird vereinfacht.
Software muss übersichtlich sein.	Einfacher zu erweitern und zu verändern (an einer zentralen Stelle im Programm). Änderungen erzeugen keine Fehler an anderer Stelle im Programm.
Dinge der realen Welt müssen einfach und direkt abbildbar sein.	Komplexe Begriffe aus der Anwendungswelt, wie z.B. ein Kunde, sollten ohne Übersetzung und Zerstückelung in dem Quellcode benutzt werden können. Kein Bruch der Begriffe zwischen Anforderung und Realisierung. evtl.: Die Anforderungsbeschreibung des Problems soll möglichst ohne Bruch realisiert werden.

Objektorientierte Programmierung: Verknüpfung von Daten und ihren Funktionen

Bei der objektorientierten Software-Entwicklung werden nicht nur die Daten und Funktionen beschrieben, sondern auch deren "innere" Beziehung. Funktionen und Daten werden miteinander verknüpft und zusammen an einer Stelle zentral definiert. Realisiert wird diese Verknüpfung durch die Möglichkeit eigene Datentypen, sogenannte Klassen, zu erstellen. Die

Variablen, die den Datentyp der Klasse besitzen, nennt man „Objekte“ oder „Instanzen“. Eine Klasse ist daher ein abstrakter Oberbegriff für Dinge, die eine gemeinsame Struktur und/oder ein gemeinsames Verhalten haben.

Ein Objekt ist ein zur Laufzeit eines Programms vorhandenes Exemplar einer Klasse, für das Speicherplatz zur Verfügung gestellt ist. Man sagt auch: Ein Objekt ist eine Instanz einer Klasse. Wird ein Exemplar einer Klasse erzeugt, so spricht man von Instanziierung.

Vorteile (Prinzipien der Objektorientierten Programmierung (OOP)) sind:

- **Abstraktion (→Klassen).** Es können eigene benutzerdefinierte Datentypen (Klassen) erstellt werden, die z. B. auch die Realisierung komplexer Anforderungsbegriffe (wie z. B. einen Kunden mit seinen Daten und Verhalten) ermöglichen.
- **Kapselung (→Schutz, Vereinfachung).** Zugriffe auf diese mächtigen Klassen werden an zentraler Stelle definiert und sind nur über spezielle Funktionen kontrolliert möglich. Damit erhält man Black-Box Bauelemente, d. h. Implementierungsinformationen werden verdeckt und nur die Information wie diese Bauelemente "einzuklinken" sind (über Parameter, Rückgabewerte) sind öffentlich. Dieses *information hiding* entbindet den Programmierer auch von der Notwendigkeit sich damit zu beschäftigen (einzuarbeiten).
- **Vererbung (→Wiederverwendbarkeit).** Allgemeine Informationen können in Basis Klassen zusammengefasst werden und auf andere Klassen übertragen werden (Bildung von Klassenhierarchien). Solche Basisklassen sind gute Kandidaten um in diversen SW Projekten wiederverwendet zu werden. Spezialfälle sind Abstrakte Klassen und Schnittstellen.
- **Polymorphie (→Flexibilität).** Innerhalb von Klassenhierarchien können bezogen auf die beteiligten Klassen verschiedene Versionen ein und derselben Funktion existieren. Bei der Verwendung eines Objektes wird automatisch die passende Version ausgeführt.

Bei richtiger Verwendung dieser Möglichkeiten wird die Softwareentwicklung damit **weniger fehleranfällig** und bereits geschriebener Code **wiederverwendbar**. Dies ist vor allem bei großen Softwareprojekten unabdingbar.

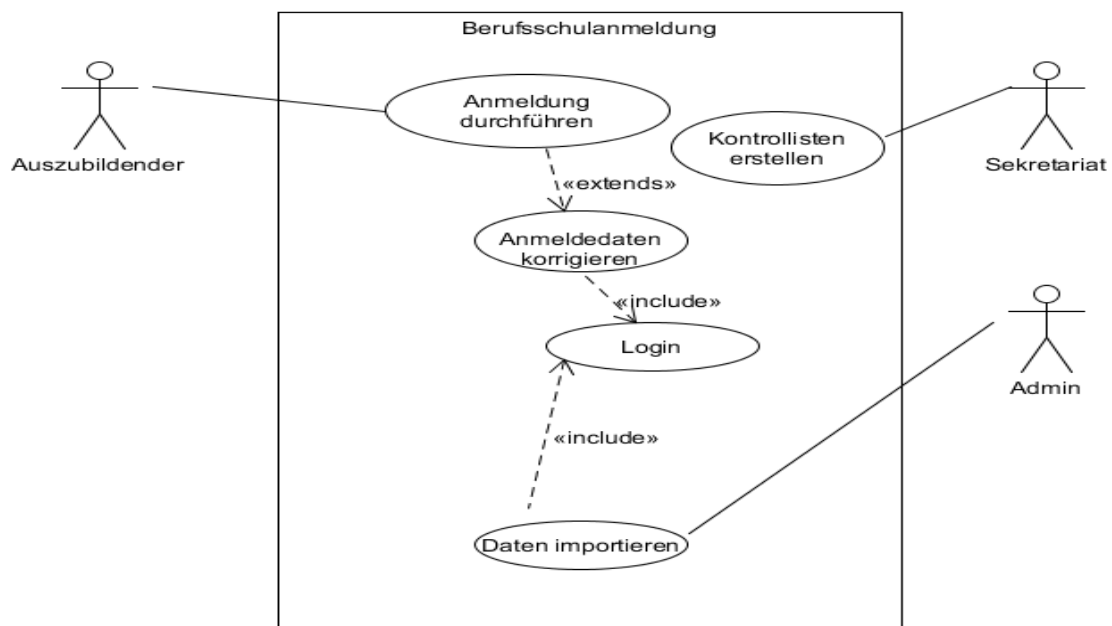
UML

In der Entwurfsphase der objektorientierten Programmierung werden grafische Notationen benutzt. Dazu gibt es einen weltweiten Standard, die so genannte UML (Unified Modeling Language).

2 Anwendungsfalldiagramme (Use Cases)

Use Case Diagramme werden zur Festlegung der Anforderungen an ein Softwaresystem (Anforderungsanalyse, Requirements Engineering) eingesetzt. Use Case Modelle sind leicht verständlich und ein gutes Kommunikationsmittel zwischen Systemanalytiker, Anwender und Entwickler. Sie legen die Grenzen des Systems, die Akteure, die darauf zugreifen und die von der Software unterstützten Teilprozesse fest. Ein Use Case Diagramm stellt also eine grobe Skizze des Systems dar.

Das folgende Use Case-Diagramm zeigt, wie ein Softwaresystem die jährliche Erfassung der Stammdaten für die Auszubildenden am HNBK-Berufskolleg unterstützt.



3 Klassen / Objekte

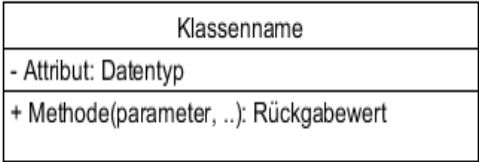
3.1 Definition und Verwendung

Ausgehend von den Anwendungsfällen des Use-Case-Diagrammes werden jetzt die für die Software benötigten Klassen und Objekte identifiziert.

Definitionen:

Klasse	Eine Klasse ist ein neuer, benutzerdefinierter Datentyp, der sowohl Daten als auch Funktionen beinhaltet.
Objekt (Instanz)	Ein Objekt (Instanz) ist eine Variable einer Klasse.

Modellierung von Klassen mithilfe des UML-Klassendiagramms:

UML Klassendiagramm	Erläuterung
 <p>The diagram shows a rectangular box representing a class. It is divided into three horizontal sections. The top section is labeled 'Klassenname'. The middle section is labeled '- Attribut: Datentyp'. The bottom section is labeled '+ Methode(parameter, ..): Rückgabewert'.</p>	<p>Klassenname: gültiger Bezeichner, am Anfang ein Großbuchstabe</p> <p>Attribute: auch Instanzvariablen, Elementvariablen, oder Membervariablen. Keine oder mehrere Attribute sind möglich.</p> <p>Methoden: auch Elementfunktionen oder Memberfunktion. Mindestens der vom Compiler zur Verfügung gestellte Standardkonstruktor ist vorhanden.</p> <p>Zugriffsmodifikatoren:</p> <ul style="list-style-type: none">- = Private, auf diese Elemente dürfen nur eigene Elementfunktionen direkt zugreifen.+ = Public, auf diese Elemente darf auch von außen mit dem . (Punkt)-Operator zugegriffen werden.# = Protected, nur eigene Elemente und "Erben" dürfen darauf zugreifen (das ist beim Thema Vererbung wichtig).

Standardmäßig sollten Attribute private und die Methoden public gesetzt werden. (Prinzip der Kapselung in der OOP).

3.2 Beispielklasse Auszubildender

Auf Basis des Use-Case-Diagrammes „Anmeldung“ wird die Klasse Auszubildender identifiziert und in C# umgesetzt.

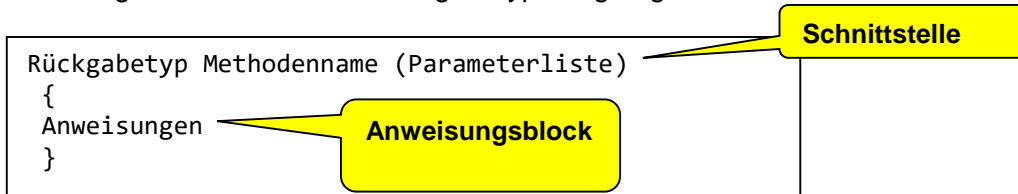
UML- Klassendiagramm	VC# Code
<pre> classDiagram class Auszubildender { -id: long -name: String -vorname: String -startDate: Date -endDate: Date +Auszubildender() +Auszubildender(long id, String name, String vorname) +~Auszubildener() +setId(long id): void +getId(): long +calcAusbildungsdauer(): Date } class Azubi1 { <<instanceOf>> id = 1 name = "Meier" vorname = "Harald" startDate = "1.07.2015" endDate = "1.07.2018" } Azubi1 --> Auszubildender : «instanceOf» </pre> <p>Hinweis: die Darstellung des Standardkonstruktor Auszubildender() in UML ist optional, get- und set-Methoden werden in der Regel nicht dargestellt!</p>	<pre> //Vor class Programm (Klasse definieren) class Auszubildender { public long id; ... public void setId(long id) { this.id= id; } ... public string getId() { return id; } ... public Auszubildender() { } public Auszubildender(long id, String name, String vorname) { this.id = id; this.name = name; this.vorname = vorname; } public ~Auto() { Console.WriteLine("Ende"); } }; //In static void Main(string[] args) // Objekte der Klasse erzeugen und benutzen Auszubildender a1 = new Auszubildender(); //Objekt wird erzeugt a1.setId(1); Console.WriteLine(a1.getId()); Auszubildender a2 = new Auszubildender(); a1 = a2; /*a1 bekommt den Verweis von a2. Damit wird die Adresse, wo das Objekt a1 zu finden ist, überschrie- ben. Ein Objekt ohne Verweis, kann nicht mehr ge- nutzt werden und ist daher ein Kandidat für den Garbage Collector (Objekt-Loeschung)*/ </pre>

In den folgenden Abschnitten erläutern wir, das Aussehen von Methoden und spezielle Methoden.

3.3 Methoden allgemein

Definition einer Methode:

Die zur Methode gehörenden Anweisungen stehen in geschweiften Klammern. Die sog. Schnittstelle enthält die Information wie die Methode heißt, welche Eingabe (Parameter) notwendig sind und welcher Rückgabetyt festgelegt ist.



Name einer Methode:

Der Name der Methode ist im Rahmen der Regeln für gültige Bezeichner (z. B. keine Sonderzeichen) frei wählbar. Ein guter Methodenname sollte aussagekräftig sein und einen Hinweis auf die Aufgabe der Methode geben. Methodennamen beginnen mit einem Kleinbuchstaben. Über ihren Namen kann die Methode später aufgerufen werden.

Parameter:

Parameter sind lokale Variablen der Methode, denen bei Aufruf der Methode Werte übergeben werden. Jeder Parameter wird mit Datentyp und Name definiert, die einzelnen Parameterdefinitionen werden durch Komma getrennt. Bekommt eine Methode keine Eingabe, bleibt die Parameterliste leer. Es sind 0 bis N Parameter möglich.

Rückgabewert:

Über den Rückgabewert kann die Methode einen Wert an die aufrufende Methode (Main) zurückgegeben werden. Eine Methode, die keinen Rückgabewert liefert, muss mit dem Rückgabetyt void definiert werden. Es ist kein bis maximal genau 1 Rückgabewert möglich. Methoden, die einen Rückgabewert haben, müssen eine **return Anweisung** beinhalten. Die return Anweisung beendet gleichzeitig die Funktion.

```
//Beispiel für return
int summe (int z1, int z2)
{
  return z1 + z2;
}
```

Name und Parameterliste der Methode werden zusammen als **Signatur** bezeichnet.

Beispiele für Schnittstellen von Methoden:

```
void funk(): Methode ohne Rückgabewert und ohne Parameter
int funk(double d, double z): Methode mit int-Rückgabewert und zwei double Parameter
```

Es ist erlaubt mehrere Methoden gleichen Namens zu definieren, sofern die Parameter der Methoden sich in der Anzahl oder dem Typ unterscheiden, also der Signatur unterscheiden.

Die Definition mehrerer Methoden gleichen Namens bezeichnet man als **Überlagerung** (overloading).


```
// Überlagerte Methode
void initialisieren (int z1){...}
void initialisieren (int z1, double z2){...}
void initialisieren (double z2) {...}
```

Beim Aufruf einer überlagerten Funktion sucht der Compiler die Funktionsdefinition heraus, die am besten zu der Argumenten Liste passt. Die beim Funktionsaufruf übergebenen konkreten Werte bezeichnet man als **Argumente**.

3.4 Konstruktor

Jede Klasse hat mindestens einen Konstruktor. Der Konstruktor ist eine spezielle Methode, die keinen Rückgabetyt (auch nicht void!) besitzt und den Namen der Klasse trägt. Der Konstruktor erzeugt in Verbindung mit dem new Operator ein Objekt. Konstruktoren können überladen werden. Nicht selten geschieht dies, um Werte entgegenzunehmen, die den Attributen als Anfangswerte zugewiesen werden sollen.

Klassen, die keinen eigenen Konstruktor definieren, bekommen vom Compiler einen parameterlosen Standardkonstruktor gestellt. Aber Vorsicht: sobald ein überladener Konstruktor definiert wird, steht dieser nicht mehr automatisch zur Verfügung, sondern muss falls gewünscht in der Klasse zusätzlich definiert werden.

3.5 Destruktor

Ein Destruktor trägt den Namen der Klasse mit vorangestellter ~. Er hat keinen Rückgabetyt (auch nicht void!) und kann nicht überladen werden. Der Destruktor hat die Aufgabe bestimmte Anweisungen vor dem Löschen eines Objektes auszuführen. Beispielsweise kann eine von diesem Objekt geöffnete Datei wieder geschlossen werden. Der Destruktor wird in der Regel automatisch vom Garbage Collector ausgeführt. Die Aufgabe des Garbage Collectors besteht darin, Objekte, die nicht mehr benötigt werden (Objekte ohne Verweis) im Speicher zu löschen. Wann der Garbage Collector während des Programmablaufs läuft, kann nicht beeinflusst werden. Ein Aufruf des Garbage Collectors findet aber z. B. immer statt, wenn die Methode Main beendet wird.

3.6 Set- und get-Methoden

Attribute (Membervariablen) sollten private gesetzt sein. Um einen Zugriff auf diese Daten zu ermöglichen, werden also Methoden benötigt. Eine get-Methode hat als Aufgabe, den Wert eines Attributes zu liefern. Die Methode muss daher einen Rückgabewert vom Typ des Attributes besitzen. Parameter werden nicht benötigt. Eine set-Methode setzt den Wert eines Attributes. Ein Rückgabewert ist nicht notwendig. Als alleiniger Parameter wird der zu setzende Wert benötigt, der denselben Typ wie das Attribut besitzt. Besitzen Methodenvariablen denselben Namen wie ein Attribut, ist das Attribut mit this. vor dem Namen anzusprechen.

```
//In static void Main(string[]
args)
// Objekte der Klasse erzeugen
und benutzen

    Auszubildender a1 = new
Auszubildender();
    //Objekt wird erzeugt
(6) a1.setId(1);
(7) Con-
sole.WriteLine(a1.getId());
```

4 Beziehungen zwischen Klassen (Objekten)

Definition: Beziehungen zwischen Klassen

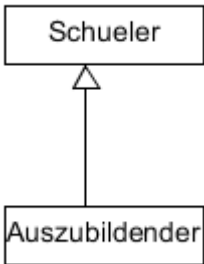
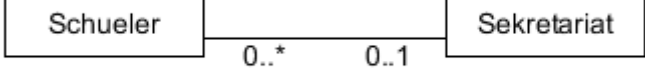
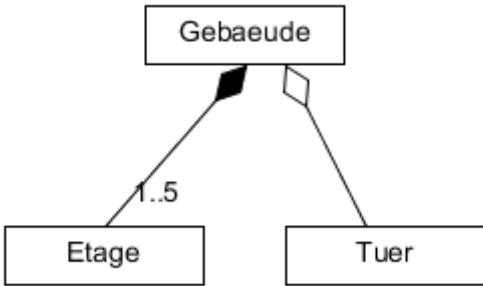
Man versteht unter den Beziehungen zwischen Klassen die gegenseitige Nutzung von Attributen und / oder Methoden anderer Klassen.

4.1 Übersicht Klassenbeziehungen

Abhängig von der Nutzung der Attribute und Methoden unterscheidet man vier verschiedene Beziehungen:

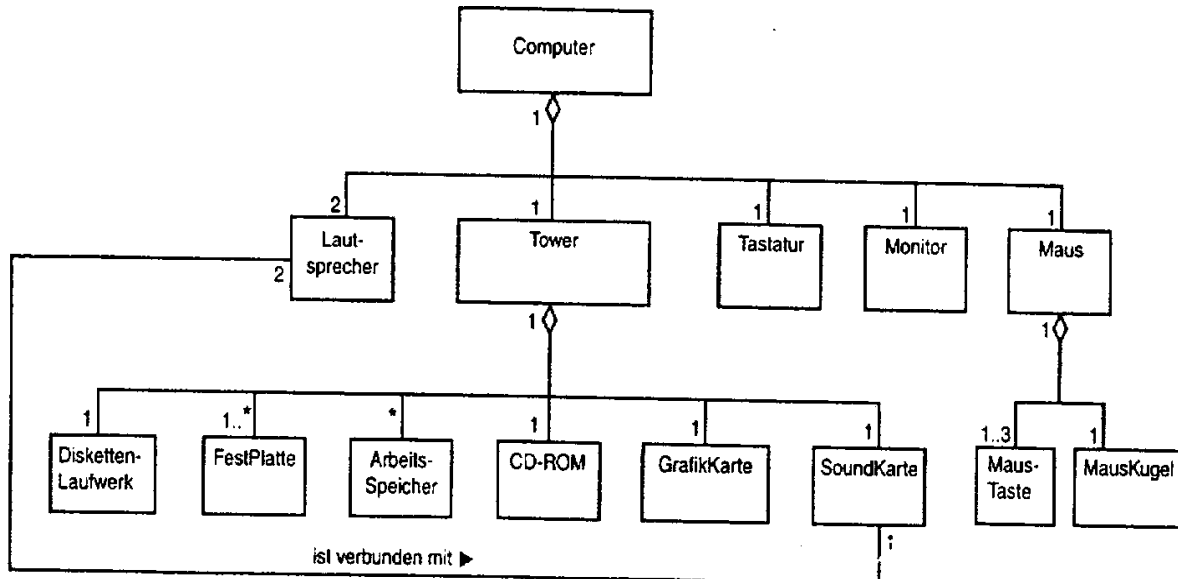
- die Vererbung (Spezialfälle: abstrakte Klassen, Schnittstellen)
- die Assoziation
- die Aggregation
- die Komposition

Beispiel 1:

Beziehung	Vererbung	Assoziation
Art der Beziehung	Ist ein	hat
Darstellung in UML	 <pre> classDiagram class Schueler class Auszubildender Auszubildender -- > Schueler </pre>	 <pre> classDiagram class Schueler class Sekretariat Schueler "0..*" -- "0..1" Sekretariat </pre>
Hinweis: Auf Angabe der einzelnen Methoden und Attribute ist aus Gründen der Übersichtlichkeit verzichtet worden.		Aggregation/Komposition
		hat / ist Teil von / besteht aus  <pre> classDiagram class Gebaeude class Etage class Tuer Gebaeude "1" *-- "1..5" Etage Gebaeude "1" *-- "1" Tuer </pre>

Beispiel 2:

Ein Computersystem besteht aus einer Kombination mehrerer unterschiedlicher Arten von Objekten: einem Tower, einer Tastatur, einer Maus, einem Monitor, einem CD-Rom-Laufwerk, einer Festplatte, einem Diskettenlaufwerk, einem Drucker, etc.. Innerhalb des Towers gibt es neben den Laufwerken einen Prozessor, eine Grafikkarte, eine Soundkarte, etc.. Alle Komponenten (Objekte) bilden zusammen das Computersystem-Objekt.



4.2 Klassenbeziehungen im Detail

Vererbung

Bei der Vererbung wird eine neue Klasse von einer bestehenden Klasse abgeleitet. Die abgeleitete Klasse erhält dadurch die Elemente (Attribute, Methoden) ihrer Basisklasse. Man spricht von einer Ist-ein-Beziehung. Die Basisklasse wird auch als Superklasse oder Elternklasse bezeichnet; die abgeleitete Klasse auch als Subklasse oder Kindklasse.

Fungiert eine abgeleitete Klasse selbst als Basisklasse ergibt sich eine Vererbungslineie. Werden von einer Klasse mehrere Klassen abgeleitet, entstehen Klassenhierarchien.

Wichtige Fakten zur Vererbung

- Die Basisklasse vererbt alle **als protected oder public gekennzeichneten** Elemente, **jedoch keine** Konstruktoren oder Destruktoren. Eine Klasse kann beliebig vielen anderen Klassen als Basis-Klasse dienen.
- Eine abgeleitete Klasse kann in vielen Programmiersprachen (C#, C++) nur eine direkte Basisklasse haben (keine Mehrfachvererbung).
- Die geerbten Elemente bilden in den Objekten der abgeleiteten Klasse ein **Unterobjekt**. Hierbei ist folgendes zu beachten:
 - Geerbte protected Elemente verhalten sich wie eigene private Elemente und können direkt benutzt werden, sind aber von außen geschützt.
 - Geerbte public Elemente verhalten sich wie eigene public Elemente
- Enthalten Basisklasse und abgeleitete Klasse Attribute gleichen Namens werden die Basisattribute verdeckt. Sie sind aber durch das Schlüsselwort **base** weiterhin verfügbar.

- **Objekterzeugung:** Bei der Erzeugung eines Objektes der abgeleiteten Klasse wird per Voreinstellung der **Standardkonstruktor der Basisklasse vor dem Konstruktor der abgeleiteten Klasse aufgerufen**. Dieses Konzept versagt aber, wenn kein Standardkonstruktor der Basisklasse vorhanden ist oder wenn ein spezieller Konstruktor anstelle des Standard-konstruktors verwendet werden soll. In solchen Fällen kann der gewünschte Spezial-Konstruktor der Basisklasse in der Initialisiererliste des Konstruktors der abgeleiteten Klasse aufgerufen werden: `KonstruktorAbgeleitet(p...) : KonstruktorBasis (p ...) { ... }` (siehe Codebeispiel)
- Eine Basisklasse kann als **Abstrakte Klasse**, von der keine eigenen Objekte erzeugt werden können, definiert werden.
Oder: Wenn von einer Basisklasse keine Objekte erzeugt werden, so nennt man diese Klasse „Abstrakte Klasse“.
In diesem Fall dient die Klasse ausschließlich der Bereitstellung von Attributen und Methoden. Einen weiteren Sonderfall stellen **Schnittstellen** (Interfaces) dar. Hier wird nur vorgegeben, welche Methoden und Eigenschaften zu implementieren sind. Es werden keine Anweisungen mitgegeben (siehe auch Abschnitt 4.2).
- Eine Basisklasse kann virtuelle Methoden enthalten, um die Möglichkeiten des Polymorphismus zu nutzen. Dies sind Methoden, die in der übergeordneten Klasse lediglich aus einer Signatur bestehen und keine Implementierung enthalten. Die Implementierung erfolgt erst in einer Kindklasse. Erst wenn alle virtuellen Methoden implementiert sind, können von einer Klasse Objekte erzeugt werden.

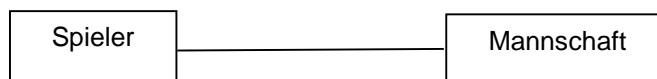
Schnittstellen

In Zusammenhang mit der Vererbung und den abstrakten Klassen gibt es außerdem das Konzept der Schnittstellen. Von einer Schnittstelle können ähnlich wie bei den abstrakten Klassen keine eigenen Objekte erzeugt werden. Schnittstellen kann man sich wie eine Klasse vorstellen, aber ohne Attribute und ohne Anweisungen. Eine Schnittstelle definiert also eine Reihe von Eigenschaften und Methoden, liefert aber keine Implementierung zu den Methoden oder den get/set Teilen der Eigenschaften. Durch die Verwendung einer Schnittstelle zeigt eine Klasse an, dass Sie die dort angegebenen Methoden und Eigenschaften implementiert. Eine Klasse kann mehrere Schnittstellen implementieren. Hierdurch wird eine Mehrfachvererbung möglich.

Assoziationen

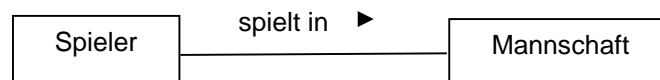
Eine Assoziation modelliert „hat“- Beziehungen zwischen Objekten gleichrangiger Klassen. Obwohl die Beziehung zwischen Objekten gemeint ist, wird sie in der Regel zwischen Klassen dargestellt.

Ein Beispiel: Eine Basketballmannschaft hat Spieler. Jeder Spieler hat eine Mannschaft.
Die UML-Darstellung für diese Assoziation sieht wie folgt aus:



Gerichtete Assoziation

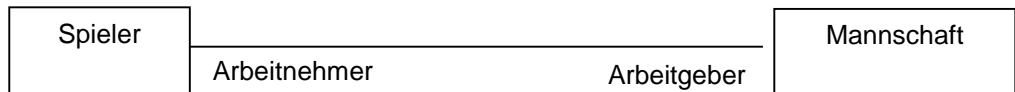
Die UML-Darstellung für diese gerichtete Assoziation sieht wie folgt aus:



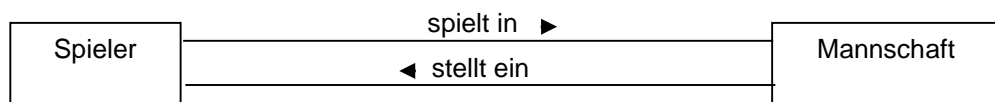
Die beiden Klassen werden durch eine Linie miteinander verbunden. Der Name der Assoziation steht oberhalb der Linie. Die Richtung der Assoziation wird durch ein ausgefülltes Dreieck dargestellt, dessen Spitze in die entsprechende Richtung zeigt.

Wenn eine Klasse mit einer anderen zusammenhängt, spielt jede dieser Klassen in der Assoziation eine Rolle. Diese **Rolle** wird in der UML-Darstellung an der Assoziationslinie vermerkt. Bei einer Profi-Basketballmannschaft ist z. B. der Spieler ein Arbeitnehmer, die Mannschaft der Arbeitgeber.

Die Rollennamen oder Assoziationsnamen müssen angegeben werden, wenn zwischen zwei Klassen mehr als eine Assoziation besteht.¹



Manchmal kann ein Objekt in mehr als nur einer Weise mit einem anderen zusammenhängen. Ein Beispiel wäre, wenn die Mannschaft den Spieler einstellt:

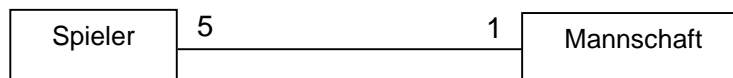


Kardinalität

Ein wichtiger Aspekt von Assoziationen zwischen Objekten ist die Multiplizität, auch **Kardinalität**² genannt. Diese besagt, wie viele Objekte einer einzelnen Klasse mit einem Objekt der assoziierten Klasse zusammenhängen.

Für unser obiges Beispiel bedeutet das: Eine Basketballmannschaft hat 5 Spieler, Reservespieler nicht mitgerechnet. Und jeder Spieler hat eine Mannschaft.

Die UML-Darstellung für diese Assoziation sieht wie folgt aus:



Bedeutung von Kardinalitäten³:

Kardinalität	Bedeutung
1	Verbindung zu genau einem Exemplar der verbundenen Klasse
0..1	Verbindung mit keinem bis maximal einem Exemplar der verbundenen Klasse
*	Verbindung mit keinem, einem oder mehreren Exemplar(en) der verbundenen Klasse
2..*	Verbindung mit 2 bis mehreren Exemplar(en) der verbundenen Klasse
5	Verbindung mit genau 5 Exemplaren der verbundenen Klasse
1, 5, 8	Verbindung mit einem, 5 oder 8 Exemplaren der verbundenen Klasse
1..7,10,13..*	Die Anzahl der zugeordneten Elemente der verbundenen Klasse ist ungleich 8, 9, 11, und 12.

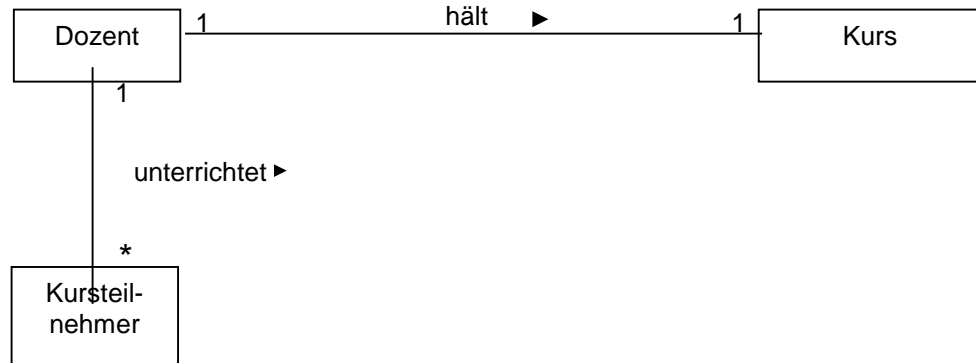
Beispiel Seminar:

¹ Vgl. Balzert, Helmut, *Lehrbuch der Softwaretechnik : Software Entwicklung*. 2. Auflage, S. 189.

² Vgl. Balzert, Helmut, *Lehrbuch der Softwaretechnik : Software Entwicklung*. 2. Auflage, S. 188.

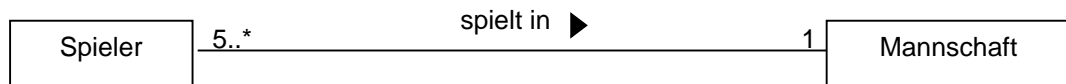
³ Vgl. Erler/Ricken, *UML*, 1. Auflage, S. 149

So wird z. B. eine typische Lehrveranstaltung von einem Dozenten durchgeführt. Der Kurs und der Dozent befinden sich in einer *eins-zu-eins*-Assoziation. Die Kursteilnehmer und der Dozent befinden sich in einer *mehrere-zu-eins*-Assoziation.



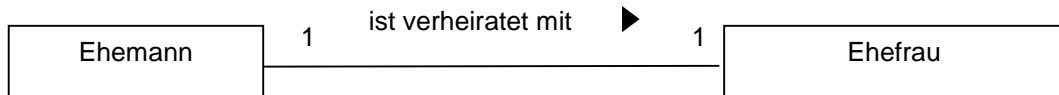
Beispiel Basketballmannschaft:

In einer Basketballmannschaft spielen 5 Spieler plus Reservespieler.

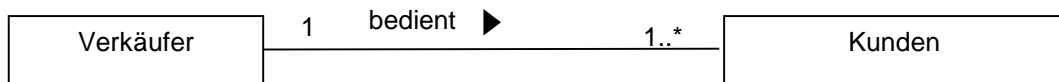


Weitere Beispiele für mögliche Kardinalitäten und ihre UML-Darstellung:

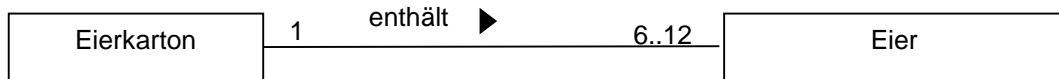
Ein Ehemann ist verheiratet mit einer Ehefrau:



Ein Verkäufer bedient ein bis mehrere Kunden:

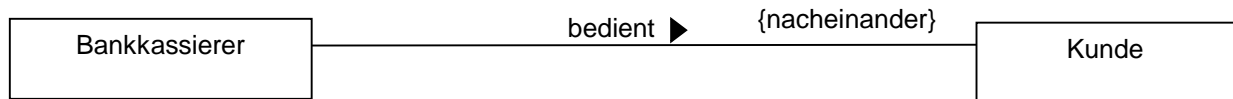


Ein Eierkarton enthält 6 bis 12 Eier:



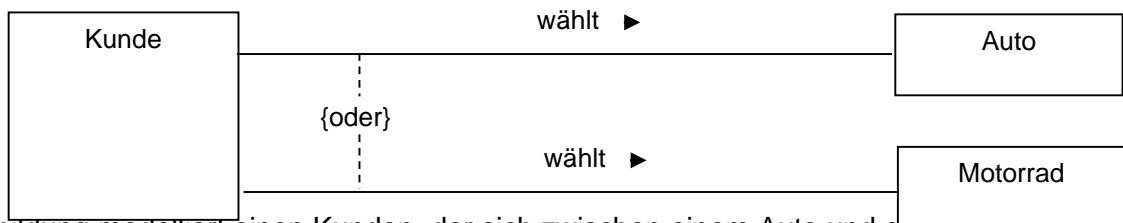
Einschränkungen für Assoziationen (Restriktionen)

Manchmal muss sich eine Assoziation zwischen zwei Klassen nach einer Regel richten. Diese Regel wird angezeigt, indem in der Nähe der Assoziationslinie eine Einschränkung angebracht wird. Die Einschränkung wird dabei in geschweifte Klammern gesetzt. Restriktionen können frei formuliert werden. Für häufig wiederkehrende Fälle ist es sinnvoll, sich Standards zu schaffen.⁴



Ein Bankkassierer bedient einen Kunden, aber jeder Kunde wird in der Reihenfolge bedient, wie er sich in die Schlange eingereiht hat.

Ein weiterer Einschränkungstyp ist die Oder-Beziehung. Diese wird durch ein {Oder} dargestellt, das auf die gestrichelte Linie geschrieben wird, die zwei Assoziationslinien miteinander verbindet.



Die Abbildung modelliert einen Kunden, der sich zwischen einem Auto und einem Motorrad entscheidet.

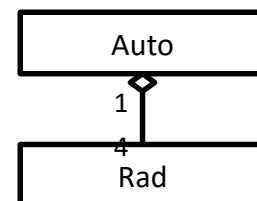
Aggregation

Eine Aggregation ist eine spezielle Assoziation. Die Komponenten und die Klasse, die sie bilden, stehen in einer Teile-Ganzes-Beziehung. Die Formulierung für eine Aggregation ist „besteht aus“ oder „ist Teil von“. Die „Teile“ einer Aggregation sind auch in einem anderen Objekt überlebensfähig.

Beispiel:

Ein Auto **hat** 4 Räder.

Die 4 Räder des VW Käfers (Seriennummer VWK185) können auch an den VW Käfer (Seriennummer VWK2000) montiert werden.



Eine Aggregation wird als Hierarchie dargestellt, indem die "Ganzes-Klasse" (z. B. das Computersystem, siehe unten) oben und die Bestandteile darunter gezeichnet werden. Das „Ganze“ wird mit einem „Teil“ durch eine Linie mit einer nicht ausgefüllten Raute auf der Seite des „Ganzen“ verbunden.⁵

⁴ Vgl. Balzert, Helmut, *Lehrbuch der Softwaretechnik : Software Entwicklung*. 2. Auflage, S. 192.

⁵ Vgl. Oestereich, Bernd, S. 269f

Komposition

Eine besondere Form der Aggregation ist die Komposition. Sie bedingt einen starken Zusammenhalt zwischen einem zusammengesetzten Objekt und seinen Teilen. Das Wesentliche bei einer Komposition ist, dass die Teile nur innerhalb des zusammengesetzten Objekts existieren. Wird das Objekt der "Ganzes-Klasse" gelöscht, dann müssen auch alle Teil-Objekte gelöscht werden.⁶ Jeder Bestandteil einer Komposition kann nur zu genau einem einzigen Ganzen gehören.

Die Formulierung für eine Komposition ist ebenfalls „*besteht aus*“ oder „ist Teil von“.

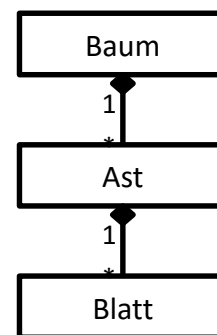
Das Symbol für ein Kompositum ist dasselbe wie für eine Aggregation, nur ist die Raute ausgefüllt.⁷

Beispiel:

Ein Baum ist eine Komposition. Er **besteht aus** Ästen und die Äste **haben** Blätter.

Die Äste eines Baums können absterben, bevor der Baum stirbt. Die Blätter eines Astes können absterben, bevor der Ast stirbt.

Wenn man hingegen den Baum zerstört, sterben auch die Äste. Stirbt der Ast, so sterben auch die Blätter.



⁶ Vgl. Balzert, Helmut, *Lehrbuch der Softwaretechnik : Software Entwicklung*. 2. Auflage, S. 195-200.

⁷ Schmuller, Joseph, S. 87.