Max-Eyth-Schule	**Klassendiagramm im ObjektOrientierten Design**	Klasse:
Tag:		Name:

Das Klassendiagramm in der Entwurfsphase

Am Ende der Analysephase sollten alle fachlichen Attribute und Methoden in den Klassen modelliert sein. Nach der Analyse- kommt die Entwurfsphase. Das Klassendiagramm aus der Analysephase wird mit weiteren Informationen angereichert. Alle Attribute sollten einen Typ haben, Methoden sollten, falls nötig, Parameter und Rückgabetypen haben. In vielen Fällen wird eine Anwendungsklasse ergänzt, eine Klasse welche das System repräsentiert und die Steuerung der Anwendung übernimmt. Nach dem Entwurf kommt die Implementierung. Am Ende der Entwurfsphase soll das Klassendiagramm so detailliert sein, dass es als Vorlage für die Implementierung der Klassen dienen kann. Jede Klasse des Klassendiagrams wird in einer objektorientierten Programmiersprache implementiert.

Sichtbarkeit

Spätestens jetzt sollte die Sichtbarkeit von Attributen und Methoden bestimmt werden (vgl. Infoblatt Klassendiagramm OOA). Zur Erinnerung: Innerhalb einer Klasse sind alle Methoden und Attribute sichtbar. Um die Datenkapselung (das Geheimnisprinzip) umzusetzen und damit die (Nicht-)Sichtbarkeit von Attributen, und Methoden nach außen hin, auch im Klassendiagramm auszudrücken, werden "+"-,"#"- und "-"-Zeichen eingefügt, mit der Bedeutung:

+	**public** / öffentlich	für andere Klassen sichtbar und benutzbar
-	**private** / privat	no public viewing, Element ist für andere Klassen nicht sichtbar
#	**protected** / privat	Elemente verhalten sich wie private-Elemente werden nur bei Ableitung der Klasse anders behandelt (siehe AB Vererbung-Übung Nr. 5)

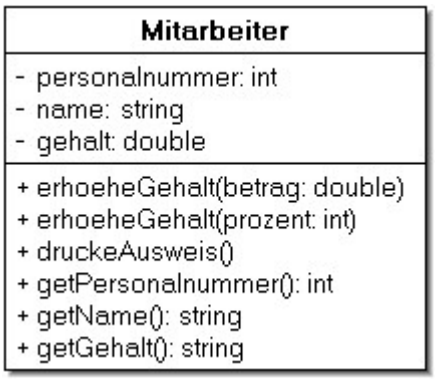
In der Regel sind Attribute privat und Methoden öffentlich.

Set- und get-Methoden (auch setter und getter genannt)

Das Konzept der Datenkapselung sieht vor, Attribute zu privatisieren und öffentliche Methoden zum Lesen und ggf. auch zum Schreiben dieser Attribute anzubieten. Für jedes Attribut wird eine Lese- und ggf. eine Schreibmethode deklariert, für die es ein festes Namensschema gibt. Die Lesemethoden heißen `get<Attributname>` und liefern das Attribut zurück. Die Schreibmethoden heißen `set<Attributname>`, haben einen Parameter mit gleichem Typ wie das Attribut und schreiben den Wert des Parameters in das Attribut.

Beispiel: Für das Attribut `name` mit dem Typ `string` gibt es die Methoden

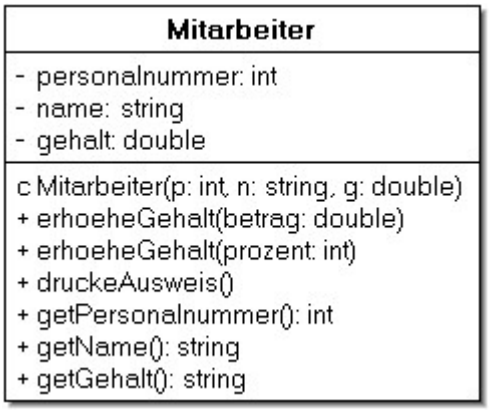
- `string getName()` ohne Parameter, mit Rückgabetyp `string`
- `void setName(string name)` mit dem Parameter `name` vom Typ `string` wie das Attribut, der dem Attribut `name` zugewiesen wird, sie hat keinen Rückgabetyp



Die getXXX()-Methoden heißen Getter, die setXXX()-Methoden Setter. Die Methoden sind öffentlich, und der Typ der Getter-Rückgabe muss der gleiche wie der Parametertyp des Setters sein. Bei boolean-Attributen darf es (und muss es in manchen Fällen auch) statt getXXX() alternativ isXXX() heißen, z. B. isFull().

Die set- und get-Methoden erscheinen häufig nicht im Klassendiagramm, da sie bekannt sind und das Diagramm nur unnötig aufblähen würden.

Konstruktoren



Die Attribute eines neuen Objekts können mit Werten versorgt werden, indem für jedes die entsprechende set-Methode aufgerufen wird. Das kann bei vielen Attributen etwas langwierig werden. Um das abzukürzen, gibt es eine spezielle Methode, die Konstruktormethode, auch kurz **Konstruktor** genannt. Ein Konstruktor hat den gleichen Namen wie die Klasse selbst, ist immer public und hat keinen Rückgabewert. Er wird bei der Objekterzeugung automatisch aufgerufen. Hat ein Konstruktor Parameter, werden diese zur Initialisierung der Attribute verwendet. Die Attribute müssen nicht mehr über einzelne set-Methoden mit Werten versehen werden, sondern erhalten durch einen Konstruktor-Aufruf ihre Werte.

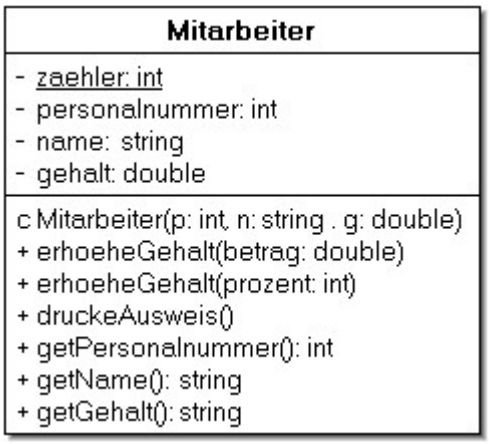
Häufig werden für die Parameter eines Konstruktors die gleichen Namen wie für die zugehörigen Attribute verwendet. Das hat den Vorteil, dass es keine Inflation von Variablennamen gibt. Es hat den Nachteil, dass die Schnittstellen der Konstruktoren ziemlich lang werden können.
Beispiel: `Mitarbeiter(personalnummer: int, name: string, gehalt: double)`

Es kann mehrere Konstruktoren, mit unterschiedlicher Anzahl von Parametern oder mit unterschiedlichen Parametertypen geben (vgl. Abschnitt Methoden überladen). Es ist zu überlegen, ob die Methode `setName()` noch Sinn ergibt, wahrscheinlich nicht.

Klassenattribute und Klassenmethoden

Die Personalnummer von Mitarbeitern soll fortlaufend vergeben werden. Die erste Mitarbeiterin erhält die Nr. 1, der zweite Mitarbeiter die 2, die dritte die 3 usw. Die Personalnummer ist ein Attribut des Mitarbeiter-Objekts, die letzte vergebene Personalnummer kann kein Attribut eines Objekts sein, das ergibt keinen Sinn. Die Klasse muss wissen welche Nummer zuletzt vergeben wurde, damit das nächste neue Objekt die nächste Zahl als Personalnummer bekommt.
Wie kann dieser Zähler modelliert werden? - in einem normalen Attribut? - nein
Es muss ein Attribut geben, das zur Klasse gehört und das bei jeder Erzeugung eines neuen Mitarbeiter-Objekts um eins erhöht wird und dann als Personalnummer an das Mitarbeiter-Objekt weitergegeben wird.

Wie die Überschrift vermuten lässt:
Dieser Zähler gehört zur Klasse Mitarbeiter, nicht zu einem bestimmten Objekt der Klasse. Er wird als **Klassenattribut** modelliert und erscheint im UML-Klassendiagramm unterstrichen. In C++ heißen solche Attribute statische Attribute/Variablen, weil sie durch das Schlüsselwort **static** gekennzeichnet sind.



Die Klasse Mitarbeiter erhält nun ein weiteres Attribut, ein Klassenattribut `zaehler` vom Typ ganze Zahl zum Durchzählen der Objekte.

Klassenattribute wurden am Beispiel einer Personalnummer erklärt, es ist aber ein allgemeines Konzept und kann für viele unterschiedliche Zwecke eingesetzt werden.

Analog zu Klassenattributen, Attributen mit einem Wert pro Klasse, gibt es auch Klassenmethoden.

Sie haben den Vorteil, dass sie ohne ein Objekt zu erzeugen ausgeführt werden können.

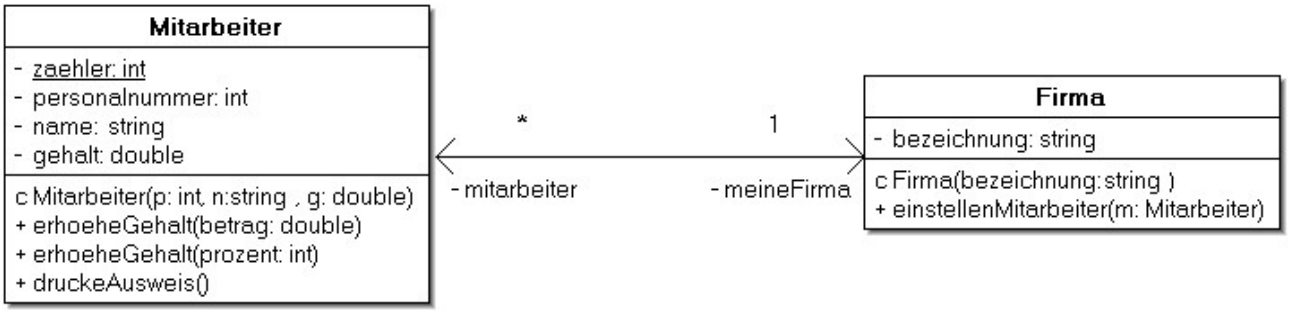
Alle Attributwerte eines Objekts in einem Text – `toString()` –Methode

(Ist in Java üblich und wird im Zentralabitur häufig verwendet !!!)

Es ist praktisch eine Methode zu haben, die den Zustand eines Objekts, also die Werte aller Attribute, zusammen als ein Text erzeugt. Vereinbarungsgemäß heißt diese Methode `toString()`. Der erzeugte Text kann dann einfach in einer `cout`-Anweisung ausgegeben werden. Die `toString()` –Methode wird immer dann aufgerufen, wenn in einer Ausgabeanweisung einfach der Objektname angegeben ist. Z. B. `cout<<(m1)` ist gleichbedeutend mit `cout<<(m1.toString())`, wobei m1 ein Objekt der Klasse Mitarbeiter ist.

Assoziationen

Spätestens jetzt sollten alle Assoziationen vollständig beschrieben sein, mit Multiziplitäten und Rollennamen sowie der Richtungen. Diese Angaben machen klare Vorgaben für die Implementierung.



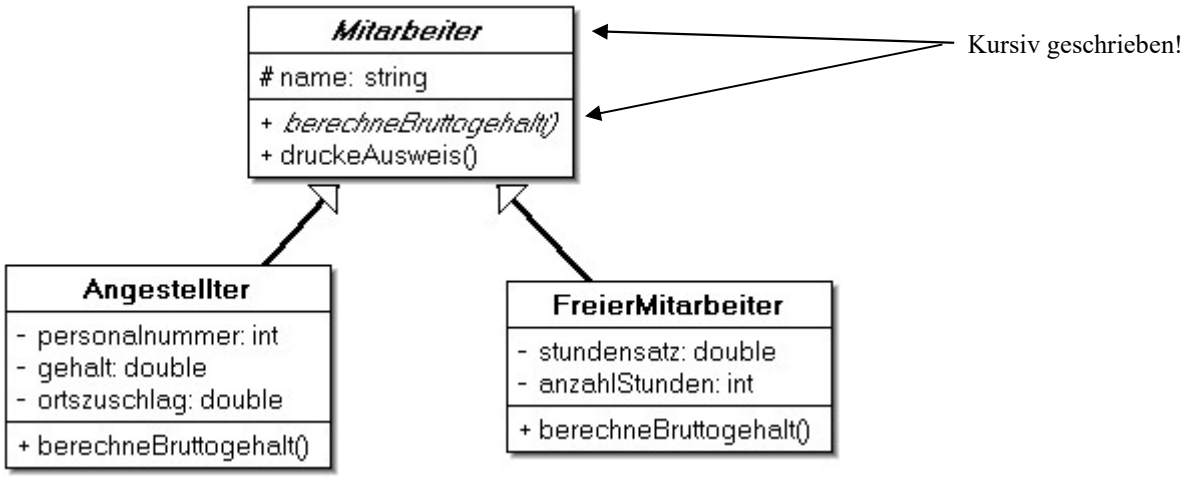
Eine **Assoziation** ist bestimmt durch …

- einen **Assoziationsnamen**, im Beispiel "arbeitet_in", der kann auch weggelassen werden
- die **Navigierbarkeit**, d. h. ob eine Assoziation in eine (unidirektional) oder in beide Richtungen existiert (bidirektional). Im Beispiel existiert die Assoziation "arbeitet in" in beide Richtungen, aus Sicht des Mitarbeiters wird die Firma als "meineFirma" modelliert, und aus Sicht der Firma sind die Mitarbeiter die "mitarbeiter". Zu einem Mitarbeiter kann seine Firma ermitteln werden und zu der Firma alle Mitarbeiter. Diese bidirektionale Assoziation wird als Linie mit zwei Pfeilspitzen dargestellt, gelegentlich auch als Linie ganz ohne Pfeilspitzen. Im nächsten Abschnitt gibt es Beispiele für unidirektionale Assoziationen, etwa Fan und Verein.
- "meineFirma" und "mitarbeiter" sind die **Rollennamen** der Objekte. Ein Rollenname besagt, welche Rolle ein Objekt für die Assoziation spielt. Diese Rollennamen sind für die Implementierung wichtig, sie werden zum Namen der Referenzattribute in der jeweiligen Klasse. Ein "-" vor dem Rollennamen bedeutet, dass es ein privates Referenzattribut ist.
- Für jede Richtung, in die eine Assoziation navigierbar ist wird die **Multiplizität** angegeben, im Beispiel der "*" auf der Mitarbeiter-Seite, er steht für beliebig viele, und die "1" auf der Firmen-Seite, die Assoziation wird folgendermaßen gelesen: "Ein Mitarbeiter arbeitet in genau einer Firma, in einer Firma können mehrere Mitarbeiter arbeiten"

Vererbung – abstrakte Klassen und Polymorphie

In einer Vererbungshierarchie kann es eine oder mehrere Superklassen geben, die nur der Strukturierung dienen, von denen aber kein Objekt erzeugt werden soll. Diese Superklassen geben häufig Methoden vor, die von den Subklassen konkret implementiert werden müssen. Diese Methoden werden zu abstrakten Methoden gemacht und damit die Superklasse zu einer **abstrakten Klasse**.

Beispiel:



Die Klasse Mitarbeiter dient der Strukturierung, sie fasst die gemeinsamen Attribute und Methoden zusammen. Sie gibt eine Methode `berechneBruttogehalt()` vor, diese Methode kann hier nicht implementiert werden, da die Berechnung für Angestellte und freie Mitarbeiter unterschiedlich ist. Die Methode ist abstrakt und wird **kursiv geschrieben**, oder mit dem Wort <<abstract>> markiert. Die Methode `berechneBruttogehalt()` ist auch eine Methode der beiden Subklassen, dort wird sie auch konkret implementiert. Für einen Angestellten werden Gehalt und Ortszuschlag addiert, für einen freien Mitarbeiter wird der Stundensatz mit der Anzahl der Stunden multipliziert. Es ergibt Sinn die Methode in der Superklasse Mitarbeiter vorzugeben, denn so haben alle Methoden mit der gleichen Bedeutung auch die gleiche Schnittstelle und es wird sichergestellt, dass sie auch implementiert werden, andernfalls gibt es einen Compilerfehler. Eine abstrakte Methode einer Superklasse muss in jeder Blatt-Subklasse des Vererbungsbaums implementiert werden. Von einer abstrakten Klasse kann kein Objekt erzeugt werden.

Das griechische Wort Polymorphie bedeutet "Vielgestaltigkeit". In der Objektorientierung hat die Polymorphie in Verbindung mit einer Vererbungshierarchie eine große Bedeutung. Sie bedeutet, dass verschiedene Objekte bei Aufruf derselben Operation unterschiedliches Verhalten an den Tag legen können. Welche Methode beim Aufruf einer Operation aufgerufen wird, hängt davon ab, welche Klassenzugehörigkeit das betreffende Objekt hat. Der Aufruf der Operation erfolgt damit polymorph. Es wird erst zur Laufzeit, in Abhängigkeit vom konkreten Objekt entschieden, welche Methode aufgerufen wird. Dies wird mit dem Begriff "späte Bindung", "late binding“ bezeichnet.

Aus Sicht des Methodenaufrufs betrachtet: ein Methodenaufruf, kann an Objekte unterschiedlicher Klassen gesendet werden und diese Objekte interpretieren die Botschaft je nach Klasse unterschiedlich. Das geht dann, wenn in unterschiedlichen Klassen dieselben Methodennamen für gleichartiges Verhalten verwendet werden. Die Methoden der Subklassen überschreiben die Methoden der Superklassen. Die Methoden haben exakt gleiche Schnittstellen. Für ein Objekt wird eine Methode aufgerufen und es ist zur Compilezeit noch nicht klar, zu welcher exakten Klasse das Objekt gehört. Erst zur Laufzeit ist klar zu welcher es gehört und exakt welche Implementierung der Methode ausgeführt wird. Dieser Mechanismus heißt auch "late binding" oder "späte Bindung".

Im Beispiel: für `einMitarbeiter`-Objekt wird die Methode `berechneBruttogehalt()` aufgerufen, ist es ein Angestellter, wird die Implementierung der Klasse `Angestellter` ausgeführt, ist es ein freier Mitarbeiter die Implementierung dieser Klasse.