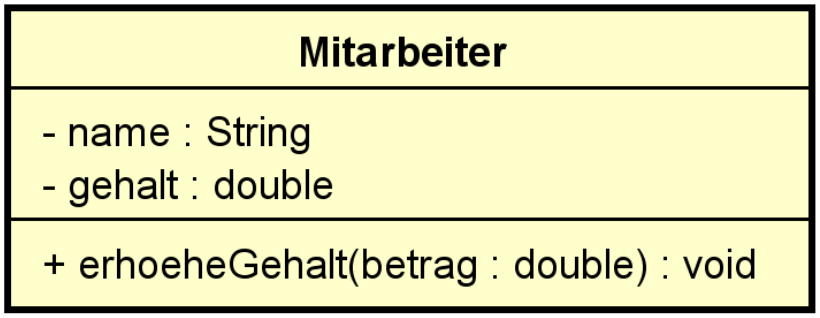
UML Klassendiagramm

Das Klassendiagramm – ein **Strukturdiagramm** – haben wir bereits kennengelernt. Wir wiederholen nochmal das Wichtigste und gehen etwas mehr in die Tiefe.

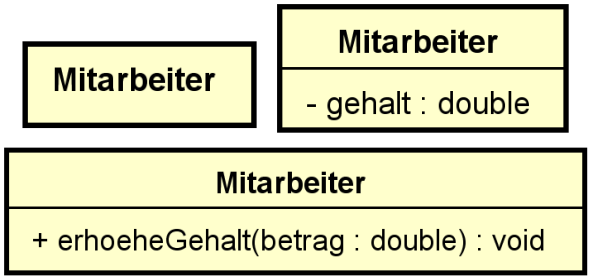
Eine Klasse wird genauso wie ein Objekt mit einem Rechteck dargestellt und besteht aus drei Abschnitten: Klassenname, Attribute und Operationen (Methoden).

Notation von Klassen



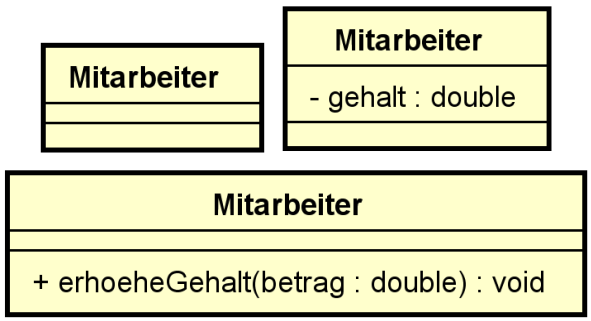
Sind die Attribute oder Methoden aktuell nicht von Interesse, kann der entsprechende Abschnitt ausgelassen werden:

Abschnitte weglassen



Um Missverständnissen vorzubeugen, kann man trotzdem Trennstriche einfügen, um einen leeren Abschnitt zu signalisieren. Dies ist jedoch nicht verpflichtend.

Abschnitte andeuten



Klassennamen

Sprechende Klassennamen sind wichtig für ein verständliches Modell. Klassen werden mit einem Substantiv im Singular (Hauptwort in Einzahl) benannt. Optional können sie durch Adjektive

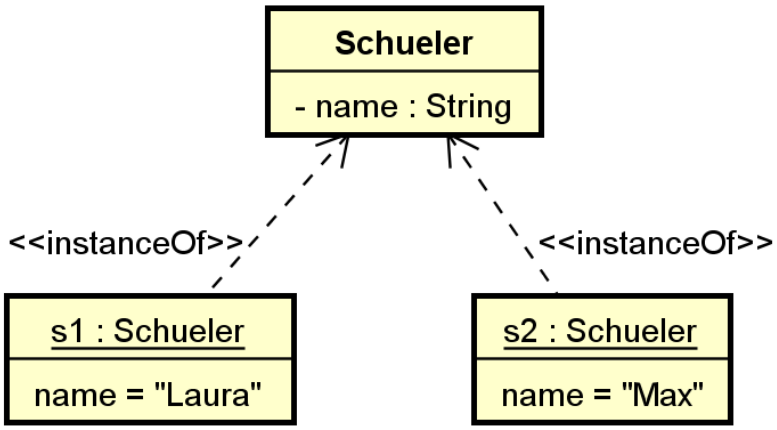
(Eigenschaftswörter) ergänzt werden. Klassennamen sollen eindeutig und sprechend sein.

Attribute

Attribute und Datentypen

Die Attribute werden im zweiten Abschnitt einer Klasse eingetragen. Sie beschreiben die **Daten**, welche alle Objekte dieser Klasse speichern können. **Attributnamen** beginnen mit einem kleinen Anfangsbuchstaben und sind ebenfalls normalerweise Substantive (Hauptwörter). Attribute haben einen bestimmten **Datentyp**. Alle Objekte derselben Klasse besitzen dieselben Attribute, speichern jedoch unterschiedliche Attribut*werte*. Der Datentyp wird nach dem Attributnamen, getrennt durch einen Doppelpunkt, angegeben. Die genauen Datentypen überlegt man sich normalerweise erst in der Entwurfsphase. Folgendes Diagramm zeigt den Zusammenhang zwischen Klasse und ihren Objekten:

Klassen und Objekte



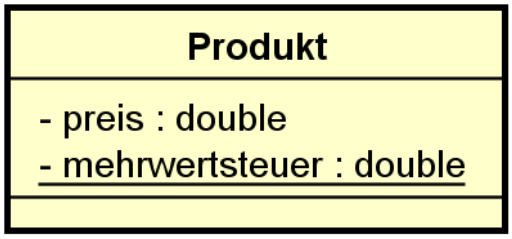
Objekte sind *Instanzen* einer Klasse. Sie speichern nun konkrete Werte für die Attribute. In diesem Fall gibt es zwei Objekte: `s1` und `s2`. Beide Objekte sind Instanzen derselben Klasse `Schueler`. `s1` speichert den String `"Laura"` für das Attribut `name`. `s2` speichert hingegen den String `"Max"`. Die `<<instanceOf>>`-Abhängigkeit signalisiert, dass diese Objekte Instanzen der Klasse `Schueler` sind. Nachdem hier ja eigentlich ein Objektdiagramm mit einem Klassendiagramm vermischt wird, verdeutlicht dieser Pfeil die Beziehung zwischen einer Klasse und dazugehörigen Objekten.

Statische Attribute

Manchmal möchte man, dass sich Objekte derselben Klasse einen Attributwert teilen. Dafür verwendet man **Klassenattribute**, auch statische Attribute genannt. Ein statisches Attribut existiert nur einmal und ist für alle Objekte gleich. Ein statisches Attribut kann auch dann

verwendet werden, wenn noch gar kein Objekt der Klasse erzeugt wurde. Sie werden im Klassendiagramm unterstrichen dargestellt.

Notation statischer Attribute



In diesem Beispiel definiert die Klasse `Produkt` die Attribute `preis` und `mehrwertsteuer`. Das Attribut `preis` ist ein *Objekt*attribut, d.h. jedes Produkt besitzt einen eigenen Preis. Im Gegensatz dazu ist `mehrwertsteuer` ein *Klassen*attribut (statisches Attribut). Alle Produkte speichern somit dieselbe Mehrwertsteuer. Wenn sich eines Tages die Mehrwertsteuer ändert, so muss dies nur einmal zentral angepasst werden. Man erspart sich daher, alle Objekte durchzugehen und die Mehrwertsteuer überall ändern zu müssen. Außerdem wird dadurch Speicherplatz gespart.

Sichtbarkeiten

In der Entwurfsphase werden außerdem auch die Sichtbarkeiten von Attributen angegeben. Hier unterscheidet man, wie in Java, zwischen **private**, **package**, **protected** und **public**.

Private Attribute sind nur innerhalb der eigenen Klasse sichtbar. *Package* Attribute sind für alle Klassen sichtbar, die sich im gleichen Paket befinden. *Protected* Attribute sind zusätzlich in allen Unterklassen sichtbar. *Public* Attribute sind für alle Klassen sichtbar.

protected vs. package

Ist protected nicht „weniger sichtbar" als package?

Nein, zumindest nicht in Java. Dabei handelt es sich um ein häufiges Missverständnis. *Protected* Attribute sind nicht nur in allen Unterklassen, sondern auch im gesamten Paket sichtbar. In Java ist es jedoch unüblich, die *package*-Sichtbarkeit (auch *default* genannt) zu verwenden. Dies ist in anderen Programmiersprachen wie C# anders.

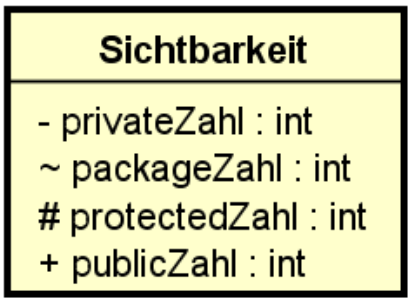
Sichtbarkeiten in Java

Folgende Tabelle gibt einen Überblick über die Sichtbarkeiten in Java:

	Eigene Klasse	Gleiches Paket	Subklassen	Alle
- private	✓			
~ package	✓	✓		
# protected	✓	✓	✓	
+ public	✓	✓	✓	✓

Im Klassendiagramm wird das jeweilige Sichtbarkeits-Symbol dem Attribut vorangestellt:

Notation von Sichtbarkeiten

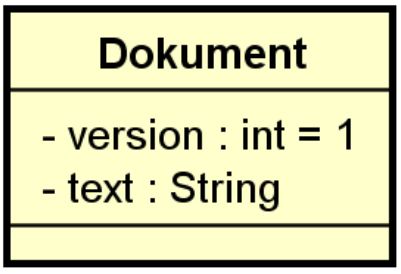


Konvention: Nur private und protected Attribute

In diesem Kurs halten wir uns an die übliche Java-Konvention, dass Attribute in Java prinzipiell immer ***private*** sind. Benötigt eine Subklasse Zugriff auf das jeweilige Attribut, sollten grundsätzlich zunächst die jeweiligen Getter und Setter verwendet werden. Soll es keine Getter und Setter geben, kann ein *protected* Attribut verwendet werden. Auf *public* Attribute ist unbedingt zu verzichten, da dies der Datenkapslung widerspricht. Auch *package* werden wir vermeiden, da dies in Java als eher unsauber gilt.

Anfangswert

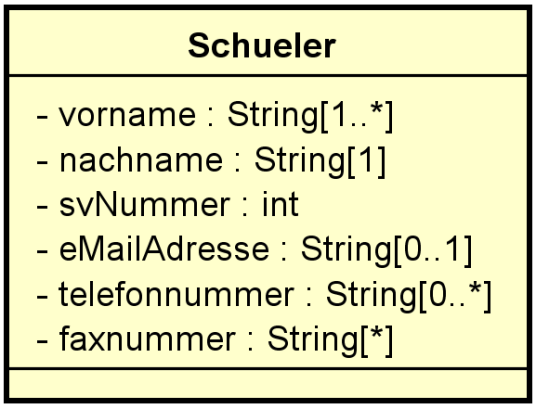
Für Attribute kann ein Anfangswert (Default-Wert) festgelegt werden:



In diesem Fall wird das Attribut `version` aller Objekte der Klasse `Dokument` standardmäßig mit dem Wert `1` initialisiert. Mit Anfangswerten sollte jedoch sparsam umgegangen werden, da die Diagramme sonst schnell unübersichtlich werden können.

Multiplizitäten für Attribute

Manchmal möchte man zu einem Attribut nicht nur einen, sondern **mehrere Werte** speichern. Dies kann über sogenannte Multiplizitäten (auch Kardinalitäten genannt) ausgedrückt werden:



Interpretation von Multiplizitäten

Die verwendete Notation `[min..max]` definiert ein **Minimum** und **Maximum** an Werten für das Attribut. Der Stern `*` steht dabei für „beliebig viele“ (inklusive 0). Das Diagramm ist daher so zu interpretieren:

- *- vorname : String[1..*]*
  Ein Schüler kann beliebig viele Vornamen haben, aber immer zumindest einen.
- *- nachname : String[1]*
  Es muss genau einen Nachnamen für jeden Schüler geben.
- *- svNummer : int*
  Es muss genau eine SV-Nummer pro Schüler gespeichert werden (dasselbe wie *[1]*).
- *- eMailAdresse : String[0..1]*
  Optional kann eine E-Mail-Adresse gespeichert werden, aber nicht mehrere.
- *- telefonnummer : String[0..*]*
  Es können optional beliebig viele Telefonnummern eines Schülers gespeichert werden.
- *- faxnummer : String[*]*
  Es können auch optional beliebig viele Fax-Nummern gespeichert werden (dasselbe wie *[0..*]*).

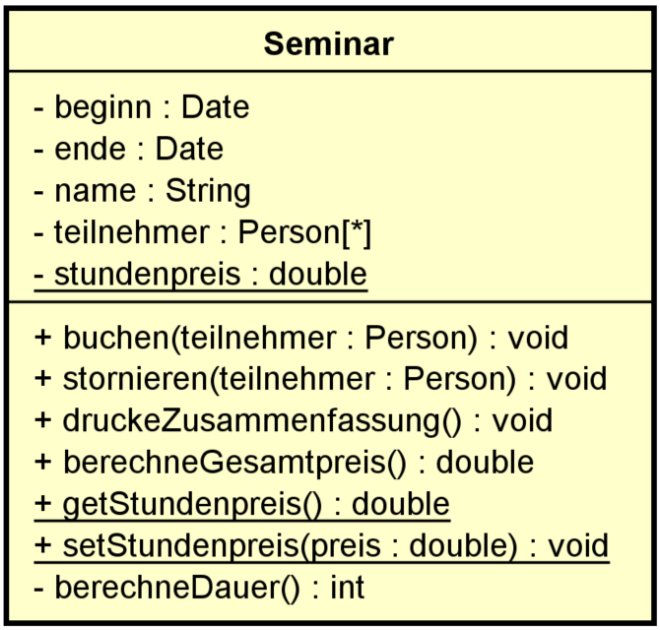
Auch Multiplizitäten machen ein UML-Klassendiagramm komplexer. Daher gibt man sie nur dann an, wenn ein Attribut mehrere Werte haben kann oder optional ist und dies für die fachlichen Anforderungen besonders wichtig ist.

Operationen

Operationen

Operationen, auch Methoden genannt, werden im dritten Abschnitt des Klassendiagramms eingetragen. Eine Operation ist eine Tätigkeit. Die Operationen bestimmen somit das **Verhalten** von Objekten. Operationen haben Zugriff auf die Attribute einer Klasse. Der **Operationsname** beginnt ebenfalls mit einem Kleinbuchstaben und soll ein Verb beinhalten. Damit wird ausgedrückt, was die Operation leistet. Ein Beispiel:

Notation von Operationen



Die Klasse Seminar besitzt sechs Operationen. Die Methoden `buchen()` und `stornieren()` erwarten jeweils einen Parameter vom Typ `Person`, um sie zur Teilnehmerliste hinzuzufügen bzw. davon zu entfernen. Es kann eine Zusammenfassung gedruckt werden. Die Methode `berechneGesamtpreis()` liefert den Gesamtpreis als `double` zurück. Nachdem alle Seminare denselben Preis pro Stunde verwenden, wird dieser in einem statischen Attribut `stundenpreis` gespeichert. Die statische Operation `getStundenpreis()` ermöglicht den Zugriff auf dieses statische Attribut. Die Operation `berechneDauer()` ist eine private Hilfsmethode, welche die Dauer des Seminars berechnet. Die Dauer wird nämlich in `druckeZusammenfassung()` und `berechneGesamtpreis()` benötigt. Daher macht es Sinn, dafür eine eigene Hilfsmethode zu erstellen.

Sichtbarkeiten von Operationen

Operationen verwenden dieselben Sichtbarkeiten wie Attribute: *public*, *protected*, *package* und *private*. Im Unterschied zu Attributen

sind die meisten Operationen jedoch ***public***. Eine Klasse möchte ja meistens ihr Verhalten nach außen sichtbar machen und Dienste anbieten. Handelt es sich jedoch nur um Hilfsmethoden, um beispielsweise Code innerhalb der Klasse wiederzuverwenden oder schöner zu strukturieren, so können sie auch *private* sein. Solche Hilfsmethoden sind meistens für andere Klassen uninteressant. Soll eine solche Hilfsmethode in Subklassen wiederverwendbar sein, kann sie auch *protected* sein. Auch bei Operationen verzichten wir in Java möglichst auf *package* Sichtbarkeit.

Statische Operationen

Was ist eigentlich der Unterschied zwischen statischen Methoden und normalen Methoden?

Statische Operationen haben **keinen Zugriff** auf ein bestimmtes **Objekt**. In Java bedeutet das für uns, dass auf **`this`** nicht zugegriffen werden kann und somit normale Objektattribute normalerweise nicht verwendet werden können. Statische Methoden ermöglichen einerseits den Zugriff auf statische Attribute, für die `this` nicht benötigt wird. Andererseits können sie verwendet werden, um Operationen durchzuführen, die *alle* Objekte betreffen. `druckeTeilnehmerliste()` könnte beispielsweise alle Personen ermitteln, die jemals ein Seminar besucht haben. Des Weiteren werden statische Methoden auch in vielen APIs für die Erzeugung von Objekten verwendet, die nicht direkt erzeugt werden können. Manchmal werden auch statische Methoden für sonstige Hilfszwecke verwendet, wie zum Beispiel Konvertierungen.

Parameter

Operationen können beliebig viele Parameter verschiedener Datentypen übergeben werden. Diese stehen innerhalb der Operation zur Verarbeitung zu Verfügung. Parameter können genutzt werden, um Informationen zur Ausführung des Verhaltens auszutauschen. In Java werden Parameter immer *„by value“* übergeben. Das bedeutet, dass sich Änderungen an den Parametern nicht auf die Variablen des Aufrufs auswirken. Bei Objekten ist dies jedoch anders zu bewerten: Wird als Parameter ein Objekt übergeben und der Zustand geändert, so sind diese Änderungen auch nach dem Methodenaufruf außerhalb sichtbar. Bei Objekten wird nämlich die Referenz kopiert, sodass Änderungen im referenzierten Bereich sehr wohl sichtbar bleiben.

```java
public static void main(String[] args) {
	int x = 1;
	Person peter = new Person();
	peter.setAlter(18);
	Person maria = new Person();
	maria.setAlter(30);
	modify(x, peter, maria);
	System.out.println(String.format("%d %d %d", x,
		peter.getAlter(), maria.getAlter()));
	// Ausgabe: 1 35 30
}

public static void modify(int a, Person p1, Person p2) {
	a = 10;
	p1.setAlter(35);
	p2 = new Person();
	p2.setAlter(40);
}
```

In diesem Beispiel sieht man, dass die Variable `x` nach dem Aufruf von `modify` noch immer den Wert `1` speichert. Der Wert von `x` wurde in den Parameter `a` *kopiert*. Deshalb ist die Änderung auf `10` in der `main`-Methode nicht sichtbar, da nur die lokale Kopie von `x`, nämlich `a`, bearbeitet wurde. Anders ist dies beim Objekt `peter`: Hier wurde `setAlter` aufgerufen, wodurch sich der Zustand des Objektes geändert hat. Diese Änderung ist in der `main` Methode sichtbar, da beide Methoden die gleiche Referenz verwenden. Der Zustand von `maria` ändert sich jedoch nicht: `p2 = new Person()` erstellt ein neues Objekt. Diese Änderung ist in der `main`-Methode nicht sichtbar, da sie mit der alten Objektreferenz weiterarbeitet.

Rückgabewert

Operationen können einen Rückgabewert haben. Dies kann entweder ein primitiver Datentyp oder ein Objekt sein. Wird nichts zurückgeliefert, wird dies in der UML durch `void` signalisiert (auch wenn manche Programmiersprachen gar keinen Rückgabetyp angeben).

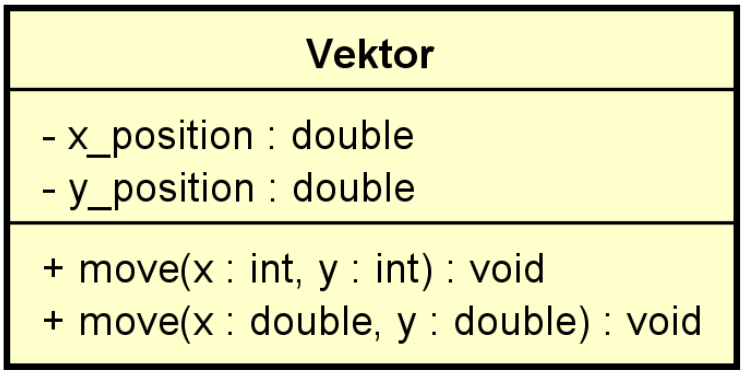
Signatur

Der **Name** einer Methode bildet zusammen mit der **Anzahl an Parametern** und ihren **Datentypen** sowie deren **Reihenfolge** die Methodensignatur. Anhand dieser bestimmt Java bei einem Methodenaufruf, welche Methode tatsächlich aufgerufen wird.

Überladen von Methoden

Unterscheiden sich Methoden innerhalb derselben Klasse nur anhand ihrer **Parameterliste**, spricht man von Überladen. Bei gleichen

Methodennamen bestimmt Java daher anhand der übergebenen Parameter und Datentypen, welche Methode genau aufgerufen wird.

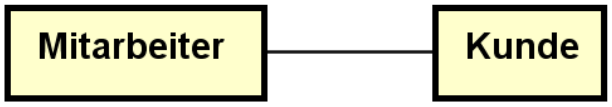


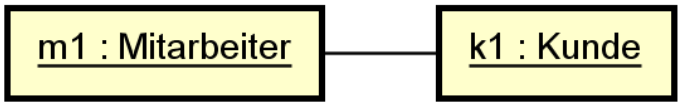
Die Klasse Vektor definiert zwei Methoden, die sich nur anhand der Parameterliste unterscheiden. Welche `move`-Methode aufgerufen wird, ergibt sich anhand der übergebenen Parameter. Wird die Methode mit `move(1, 1)` aufgerufen, so wird die erste Methode mit den `int`-Parametern ausgewählt. `move(1.0, 2.0)` ruft hingegen die `double`-Variante auf. Die Aufrufe `move(1, 2.0)` und `move(1.0, 2)` rufen ebenfalls die zweite Methode mit den `double`-Parametern auf. Das liegt daran, dass Java in solchen Fällen die präzisere Darstellung bevorzugt und bei kompatiblen primitiven Datentypen eine implizite Typumwandlung des zweiten Parameters durchführt („Promotion“ genannt) – hier von `int` auf `double`. So wird auch beim Aufruf `move(1, 2.0f)` der `float`-Wert auf `double` gecastet, sodass ebenfalls die zweite Methode verwendet wird. Es wird also niemals eine ungenauere Darstellung bevorzugt, wenn eine genauere Darstellung ohne Informationsverlust verfügbar ist.

Assoziationen

Assoziationen und Objektbeziehungen

Objekte können zueinander Beziehungen haben. Assoziationen fassen diese Beziehungen auf Klassenebene zusammen. Eine **Objektbeziehung** zwischen Objekten ist somit eine **Instanz** einer **Assoziation** – genau so wie ein Attributwert eine Instanz eines Attributs ist und ein Objekt eine Instanz einer Klasse ist.

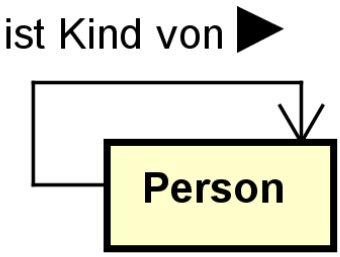


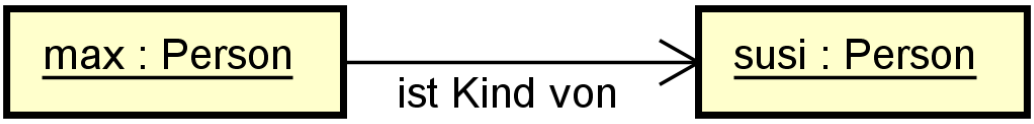


Die Klassen `Mitarbeiter` und `Kunde` besitzen eine Assoziation zueinander. Die Assoziation besitzt keinen Pfeil, weil noch keine Richtung festgelegt wurde. Die zwei Objekte `m1` und `k1` haben nun eine Objektbeziehung zueinander aufgebaut. Diese Objektbeziehung ist somit ein konkretes Exemplar (Instanz) der Assoziation zwischen den Klassen.

Reflexive Assoziationen

Objekte der gleichen Klasse können ebenfalls Beziehungen miteinander haben. Das bedeutet für die Klasse, dass sie eine **Assoziation mit sich selbst** haben musss. Dann sprechen wir von einer reflexiven Assoziation.



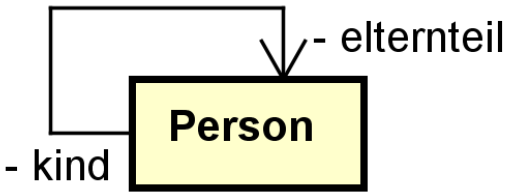


Assoziationsnamen

In diesem Beispiel sieht man, dass Assoziationen auch einen *Namen* besitzen können. Diese verleihen der Assoziation zusätzliche Ausdruckskraft. Das ist vor allem bei reflexiven Assoziationen wichtig, da sie sonst unverständlich sind. Das Dreieck neben dem Assoziationsnamen gibt die Leserichtung an.

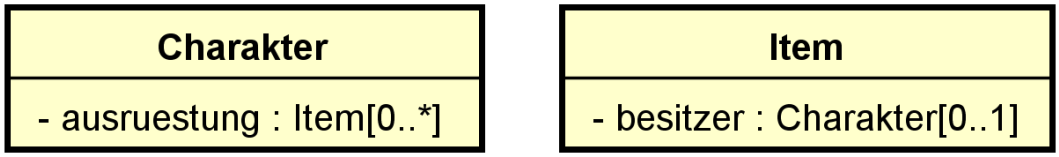
Eine weitere Möglichkeit zur Verbesserung der Verständlichkeit einer Assoziation sind Rollen. Rollen werden an den Enden einer Assoziation eingetragen:

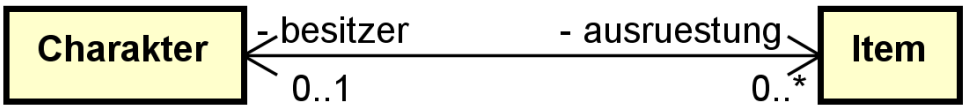
Rollen



In diesem Beispiel wurde die reflexive Assoziation mithilfe von *Rollen* verständlicher gestaltet. **Reflexive Assoziationen müssen immer entweder Rollen oder Assoziationsnamen besitzen**, da sie sonst unverständlich sind. Man sieht außerdem, dass Rollen ebenfalls Sichtbarkeiten besitzen können. Das liegt daran, dass Assoziationen auch als Attribute dargestellt werden können – und Attribute besitzen bekanntlich Sichtbarkeiten. Folgende Darstellungsarten sind daher äquivalent:

Assoziationen und Attribute



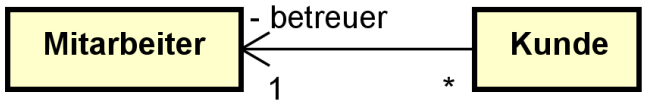


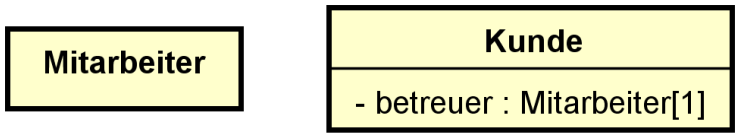
Assoziationen können daher auch **als Attribute** dargestellt werden. In der Praxis ist jedoch meist die **Darstellung als Linie zu bevorzugen**, da sie normalerweise übersichtlicher ist (ausgenommen z.B. API-Beschreibungen). Nimmt eine Klasse eine bestimmte *Rolle* bei einer Assoziation ein, so kann diese Rolle auch als Attributname bei der **gegenüberliegenden** Klasse aufgenommen werden. In diesem Beispiel nimmt `Charakter` die Rolle `besitzer` ein. Diese Rolle kann auch als Attribut `besitzer` in der Klasse `Item` dargestellt werden. Attribute können mehrere Werte besitzen. Dies gilt somit auch für Assoziationen. In diesem Beispiel kann ein `Charakter` beliebig viele `Item`s besitzen ([0..*]). Jedes `Item` ist entweder keinem oder genau einem `Charakter` – dem Besitzer – zugewiesen ([0..1]). Liegt ein

`Item` beispielsweise noch am Boden, so hat es noch keinen Besitzer (`besitzer=null`). Sobald ein `Charakter` das `Item` aufhebt, wird es der `ausruestung` hinzugefügt (z.B. eine Collection) und das Attribut `besitzer` wird auf den jeweiligen Charakter gesetzt. Bisher haben wir immer beides – die Assoziation und die Attribute – in einem Klassendiagramm angegeben.

Navigierbarkeit

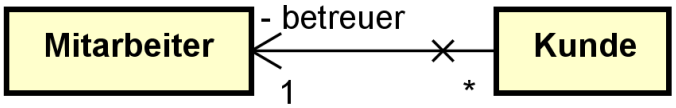
Die Attribute hängen auch eng mit der Navigierbarkeit (den Pfeilspitzen) einer Assoziation zusammen. Folgende Darstellungsarten sind daher gleichwertig:





In beiden Fällen besitzt ein Objekt der Klasse `Kunde` immer genau einen Betreuer, also ein Objekt der Klasse `Mitarbeiter`. Hat man ein `Kunde`n-Objekt gegeben, kann automatisch der zugehörige Betreuer über das Attribut `betreuer` ermittelt werden. Dies wird über die Pfeilrichtung verdeutlicht. Umgekehrt ist das nicht möglich: Hat man ein Objekt der Klasse `Mitarbeiter` gegeben, können nicht ohne Weiteres alle betreuten Kunden ermittelt werden. Möchte man diesen Sachverhalt verdeutlichen, kann das Ende der Assoziation mit einem X markiert werden:

Nicht navigierbare Assoziationsenden



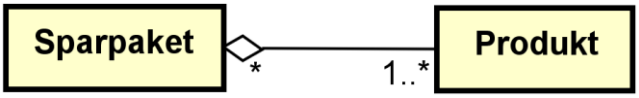
Dies wird in der Praxis jedoch selten gemacht. Streng genommen ist ein Ende ohne „X“ oder Pfeil jedoch unspezifiziert.

Eine Assoziation nennt man

- Unidirektional, wenn sie in eine Richtung navigiert werden kann (1 Pfeilspitze)
- Bidirektional, wenn sie in beide Richtungen navigiert werden kann (2 Pfeilspitzen)
- Unspezifiziert, wenn keine Richtungen angegeben sind (keine Pfeilspitzen)

Aggregation

Eine spezielle Art der Assoziation ist die Aggregation. Sie wird verwendet, wenn eine **Teil-Ganzes-Beziehung** besteht. Es existiert also eine *„ist Teil von“*- oder *„besteht aus“*-Beziehung:



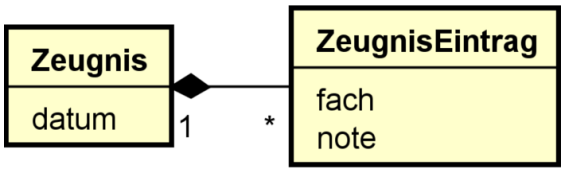
Ein `Sparpaket` besteht aus mehreren `Produkten`. Somit ist das `Sparpaket` „das Ganze“, auch „Aggregat“ genannt. Der Diamant einer Aggregation befindet sich immer auf der Seite des Ganzen. Ein `Sparpaket` muss zumindest ein `Produkt` beinhalten. Ein `Produkt` kann in mehreren `Sparpaket`en angeboten werden.

Komposition

Eine **stärkere Form der Aggregation** ist die Komposition. Auch hier existiert eine Teil-Ganzes-Beziehung mit folgenden zusätzlichen Eigenschaften:

- Jedes Teil kann nur *genau einem Ganzen* zugeordnet sein. Die Multiplizität des Ganzen ist somit immer 1. Deswegen nennt man Kompositionen auch „unshared Relationships“.
- Das Ganze ist für das *Erzeugen und Löschen der Teile* verantwortlich. Die Klasse des Ganzen sollte daher auch die Teile selbst instanziieren und z.B. in Java in einer Collection speichern.
- Wird das Ganze *gelöscht*, so werden auch automatisch die Teile *mitgelöscht*, da sie immer einem Ganzen zugeordnet sein müssen. Die Änderung der Zuordnung zu einem anderen Ganzen ist jedoch erlaubt.

Notation von Kompositionen



In diesem Beispiel beinhaltet ein `Zeugnis` mehrere Zeugniseinträge. Ein `ZeugnisEintrag` muss sich immer in genau einem `Zeugnis` befinden. Wird das `Zeugnis` gelöscht, so werden alle Zeugniseinträge darin ebenfalls gelöscht. Man sieht, dass bei einer Komposition der Diamant ausgefüllt wird. Bei Aggregationen und Kompositionen wird die Navigierbarkeit meistens so modelliert, dass vom Ganzen zu den Teilen navigiert werden kann.

Vererbung

Generalisierung und Spezialisierung

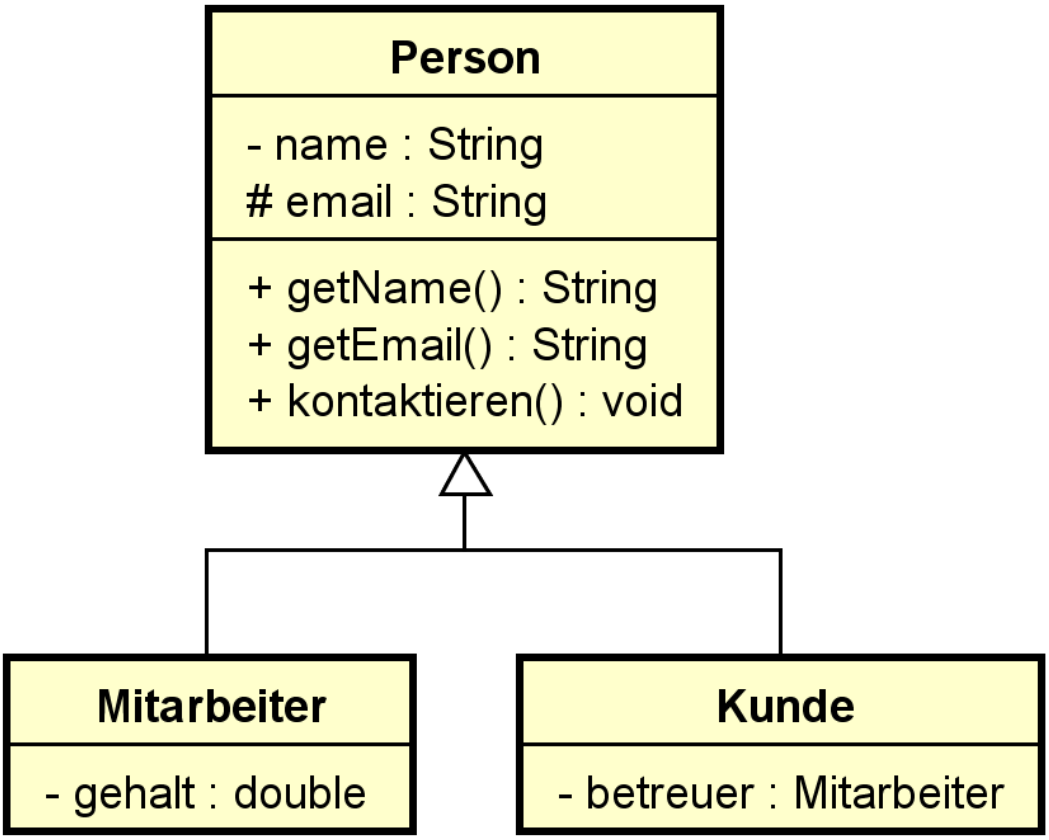
Klassen können voneinander erben. Man spricht dabei auch von Generalisierung und Spezialisierung. Bei der **Generalisierung** fasst man speziellere Klassen (Subklassen) zu einer allgemeineren Klasse (Superklasse) zusammen. Die **Spezialisierung** betrachtet diesen Prozess umgekehrt: Ausgehend von der allgemeinen Klasse (Superklasse) bildet man die spezielleren Klassen (Subklassen). Erbt eine Klasse von einer anderen Klasse, so erbt sie von ihrer Superklasse:

- Attribute
- Operationen
- Assoziationen

Vererbung privater Attribute und Operationen

Werden private Attribute eigentlich auch vererbt?

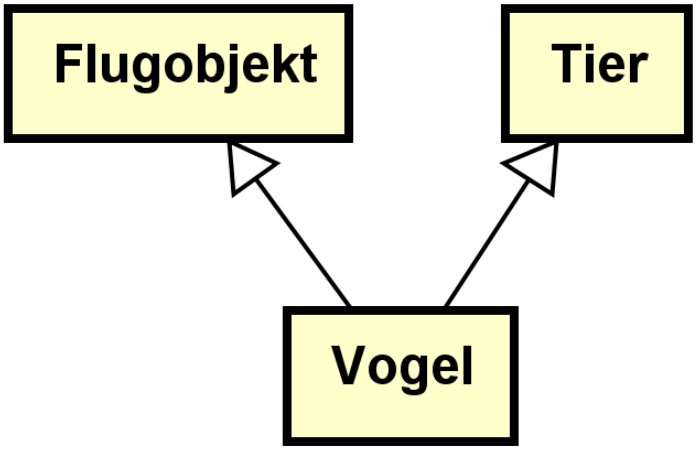
Private Attribute und Operationen werden zwar auch **vererbt**, sind aber in der Subklasse **nicht direkt sichtbar**. Sie können daher nur über andere Operationen, die sichtbar sind, verwendet werden – sofern vorhanden.



In diesem Beispiel erben die Klassen `Mitarbeiter` und `Kunde` von der Klasse `Person`. Sie haben auf alle Operationen sowie das Attribut `email` Zugriff. Das Attribut `name` ist jedoch nicht sichtbar – lediglich ein lesender Zugriff über `getName()` ist möglich. Man sieht, dass sich der Vererbungspfeil vom navigierbaren Assoziationspfeil unterscheidet, indem er ein Dreieck bildet.

Mehrfachvererbung

Manche Programmiersprachen erlauben das Erben von mehreren Klassen, weshalb es in der UML grundsätzlich erlaubt ist, von mehreren Klassen zu erben. In der Praxis ist dies jedoch zu **vermeiden**, vor allem wenn man Sprachen ohne Mehrfachvererbung wie Java verwendet. In Java gilt die Einfachvererbung, d.h. Klassen können nur von einer einzigen konkreten oder abstrakten Klasse erben.



In diesem Beispiel erbt die Klasse `Vogel` von `Flugobjekt` und `Tier`. Wir **vermeiden** solche Fälle der Mehrfachvererbung weitgehend. Stattdessen können in Java **Interfaces** verwendet werden (wird später behandelt).

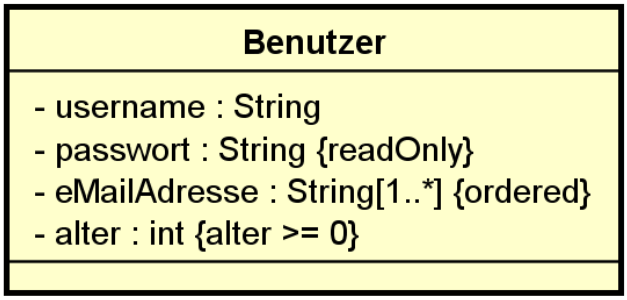
Warum ist Mehrfachvererbung eigentlich so „böse“?

Mehrfachvererbung macht das Design eines Systems **schwer zu verstehen**. Man spricht von sogenannter „Spaghetti-Vererbung“. Es ist nicht mehr klar ersichtlich, welche Klasse nun welche Attribute und Methoden überhaupt hat. Außerdem kann es schnell zu **Namenskonflikten** kommen, wenn Klassen dieselben Methoden- und Attributnamen vergeben. Weiters kann das **Diamond-Problem** auftreten: Erbt eine Klasse `A` von den Klassen `B` und `C`, die beide von Klasse `D` erben, bleibt die Frage: Welche Implementierung von `D` soll für `A` gelten? Die von `B` oder die von `C`? Dieses Problem existiert bei Interfaces nicht.

Erweiterte Kompetenzen

Zusicherungen

Eine weitere Möglichkeit zur genaueren Spezifizierung von Attributen sind Zusicherungen, auch Einschränkungen genannt. Mit ihnen können für Attribute bestimmte Eigenschaften festgelegt werden:



Das Attribut `passwort` darf aufgrund der `{readOnly}`-Zusicherung nicht geändert werden, nachdem es initialisiert wurde. Objekte der Klasse `Benutzer` besitzen zumindest eine E-Mail-Adresse. Besitzt ein Benutzer mehrere E-Mail-Adressen, sind diese geordnet. Die Reihenfolge der gespeicherten E-Mail-Adressen ist also wichtig: Es gibt eine erste (primäre) E-Mail-Adresse, eine sekundäre Adresse etc. Das Attribut `alter` muss zu jedem Zeitpunkt größer oder gleich 0 sein.

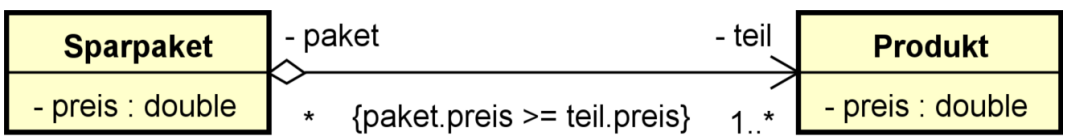
Abgeleitete Attribute

Wenn sich ein Attribut aus anderen Attributen **automatisch berechnen** lässt, spricht man von einem abgeleiteten Attribut:

Das Ende einer Vorstellung eines Kinofilms berechnet sich beispielsweise aus dem Beginn und der Dauer – abgeleitete Attribute werden daher mit einem Schrägstrich nach der Sichtbarkeit dargestellt. Abgeleitete Attribute können somit jederzeit automatisch berechnet werden. Sie dürfen nicht geändert werden.

Für Assoziationen können wie für Attribute Zusicherungen gelten:

Zusicherungen für Assoziationen



In diesem Beispiel muss der Preis für ein `Sparpaket` größer sein als der Einzelpreis seiner Teile. Es darf somit keine Objektbeziehung zwischen einem `Sparpaket` und einem `Produkt` existieren, welche die Zusicherung nicht erfüllt ist. Zusicherungen werden wie gewohnt in Mengenklammern geschrieben. Auf Attribute kann über `rolle.attributname` zugegriffen werden. Es gibt weitere spezielle Einschränkungen wie `xor`, `subsets` und `ordered`.

Weitere Konzepte

Weitere Konzepte für Assoziationen, die den Umfang dieses Kurses sprengen würden, sind mehrgliedrige Assoziationen, Assoziationsklassen, Qualifizierungen und abgeleitete Assoziationen.

Habe ich es verstanden?

Attribute:

- Ich kann UML Klassendiagramme lesen, erstellen und erklären
- Ich kann Klassen inkl. Name und Attribute sinnvoll identifizieren und grafisch darstellen
- Ich kann Datentypen von Attributen im Klassendiagramm einzeichnen
- Ich kann den Zusammenhang zwischen Klassen und Objekten erklären und diesen grafisch darstellen
- Ich kann statische Attribute modellieren und diese erklären
- Ich kann alle vier Sichtbarkeiten für Attribute einzeichnen und erklären, was diese bedeuten und welche wann zu bevorzugen sind
- Ich kann Anfangswerte einzeichnen und deren Bedeutung erklären
- Ich kann Multiplizitäten für Attribute einzeichnen und deren Bedeutung erklären
- (*Erweitert*) Ich kann Zusicherungen einzeichnen und deren Bedeutung erklären
- (*Erweitert*) Ich kann abgeleitete Attribute einzeichnen und deren Bedeutung erklären

Operationen:

- Ich kann Operationen inkl. Parameter und Rückgabewerte von UML Klassendiagrammen lesen, erklären, sinnvoll benennen, identifizieren und einzeichnen

- Ich kann statische Operationen modellieren und diese erklären
- Ich kann alle vier Sichtbarkeiten für Operationen einzeichnen und erklären, was diese bedeuten und welche wann zu bevorzugen sind
- Ich kann erklären, was bei der Übergabe von Parametern mit „by value“ gemeint ist und worin der Unterschied zwischen primitiven Datentypen und Referenzdatentypen besteht
- Ich kann das Überladen von Methoden erklären

Assoziationen:

- Ich kann Assoziationen von UML Klassendiagrammen lesen, erklären und sinnvolle Assoziationen identifizieren und zeichnen
- Ich kann den Unterschied von und den Zusammenhang zwischen Assoziationen und Objektbeziehungen erklären
- Ich kann erklären, was reflexive Beziehungen sind und diese zeichnen
- Ich kann Assoziationen benennen und sinnvolle Rollennamen vergeben
- Ich kann Assoziationen als Attribute darstellen und umgekehrt
- Ich kann die Navigierbarkeit sowie Multiplizitäten bei Assoziationen erklären und einzeichnen
- Ich kann den Unterschied zwischen Aggregation und Komposition erklären und diese einsetzen
- (*Erweitert*) Ich kann Zusicherungen für Assoziationen einsetzen

Vererbung:

- Ich kann Vererbung in Klassendiagrammen sinnvoll einsetzen und das Konzept erklären
- Ich kenne den Unterschied zwischen Generalisierung und Spezialisierung
- Ich kann erklären, dass Attribute, Operationen und Assoziationen vererbt werden, jedoch nicht immer sichtbar sein müssen
- Ich kann Mehrfachvererbung inkl. deren Nachteile erklären