

Skript Informatik B

Objektorientierte Programmierung in Java

Sommersemester 2011

- Teil 3 -

Inhalt

0 Einleitung

1 Grundlegende objektorientierte Konzepte (Fundamental Concepts)

2 Grundlagen der Software-Entwicklung

3 Wichtige objektorientierte Konzepte (Major Concepts)

... wird schrittweise erweitert

Kapitel 3:

Wichtige objektorientierte Konzepte

3.1 Hilfsmittel zur Modularisierung, Abstraktion und Hierarchie

3.2 Vererbung

3.3 Exkurs: Typing

3.4 Abstrakte Klasse

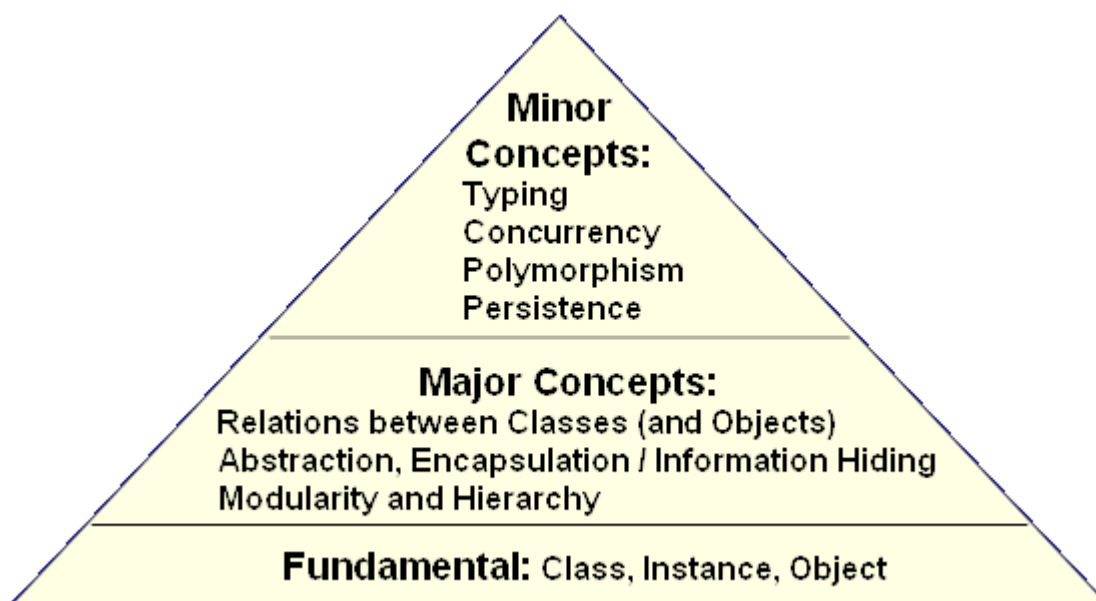
3.5 Interface

3.6 Entwicklung mit Schnittstellen

3.7 Paket

3.8 Beziehungen und ihre Modellierung

Hilfsmittel zur Modularisierung
und Abstraktion

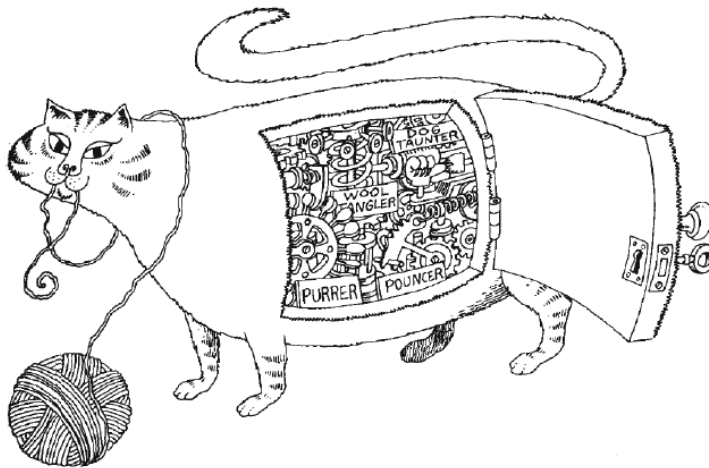


[Boo94]

3.1 Modularisierung, Abstraktion und Hierarchie

Modularisierung ist ein Vorgehen im Umgang mit großen Aufgaben und Systemen: Große Einheiten werden in kleinere unterteilt. Diese kleinere Einheiten können besser erfasst und schrittweise eine nach der anderen bewältigt werden. Für eine effiziente arbeitsteilige Problembearbeitung (z.B. im Projektteam oder in einem Netzwerk von Prozessoren) ist eine passende Modularisierung die Voraussetzung.

Modularisierung hängt eng mit Kapselung und Information Hiding zusammen. Mit Hilfe von Modulen können Interna versteckt bzw. gekapselt werden. Ein Modul ist eine sinnvoll in sich abgeschlossene Einheit.



Aus: [Boo94]

Module sind zunächst programmiersprachenunabhängig. Module und Modularisierung findet man sowohl in Programmen wie auch im Entwurf und in der Analyse, d.h. in den verschiedenen Entwicklungsphasen.

Module sind sogar noch allgemeiner zu sehen: Man findet sie auch jenseits der Welt der Softwareentwicklung.

Eine Umsetzung von Modularisierung (ebenso wie Kapselung und Information Hiding) wird in den unterschiedlichen Programmiersprachen auf unterschiedliche Weise (mit unterschiedlichen Hilfsmitteln und Konstrukten) unterstützt.

Java unterstützt eine Modularisierung z.B. durch folgende in OO Sprachen durchaus typischen Sprachkonstrukte:

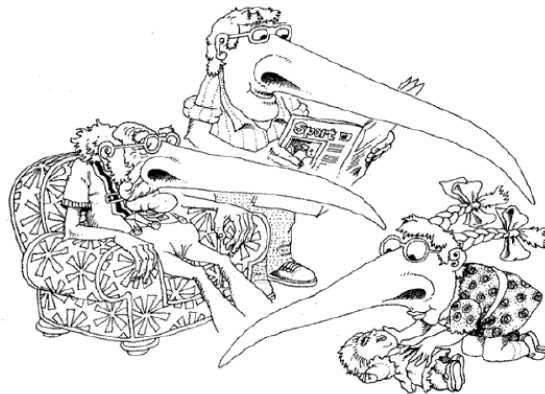
- Klassen, Objekte
 - Vererbung (als hierarchische Modularisierung)
 - Interfaces
 - Abstrakte Klassen
 - Packages (Gruppierung)
 - Sichtbarkeitsbereiche (z.B. Blockstrukturen im Code als eine Form von schwacher Modularisierung)
- } Vertrag und Schnittstelle

Bemerkungen:

Das Prinzip der Modularisierung gab es schon vor dem „objektorientierten Zeitalter“. Die Entwicklung ist vielmehr umgekehrt: Ausgehend von der Idee zur Modularisierung gelangte man zur Gestaltung mit Hilfe von Objekten.

Eine imperative Programmiersprache, wie C beispielsweise, unterstützt grundsätzlich ebenfalls eine Modularisierung (z.B. durch Funktionen und Sichtbarkeitsbereiche). Allerdings ist die Unterstützung nicht so ausgeprägt, wie in der OO Welt.

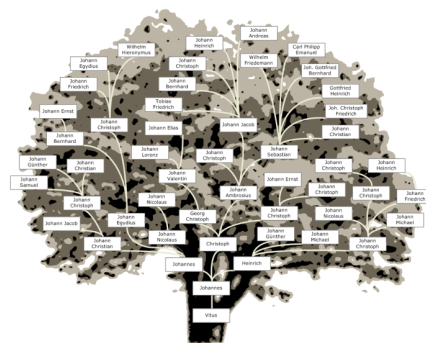
3.2 Vererbung



[Boo94]

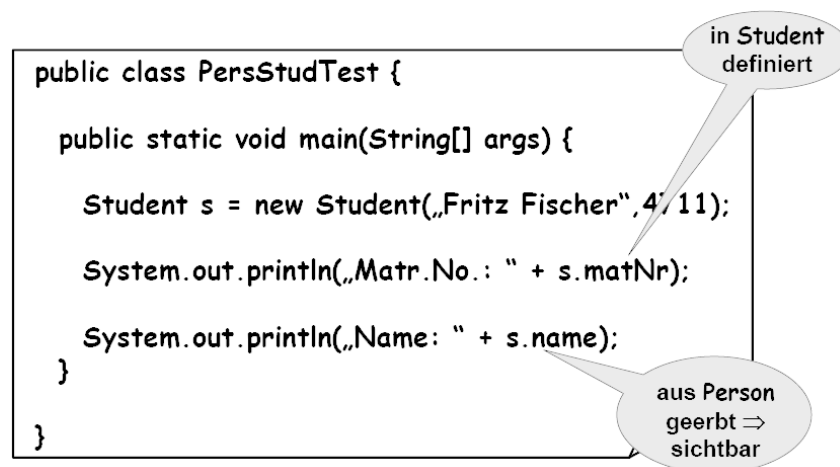
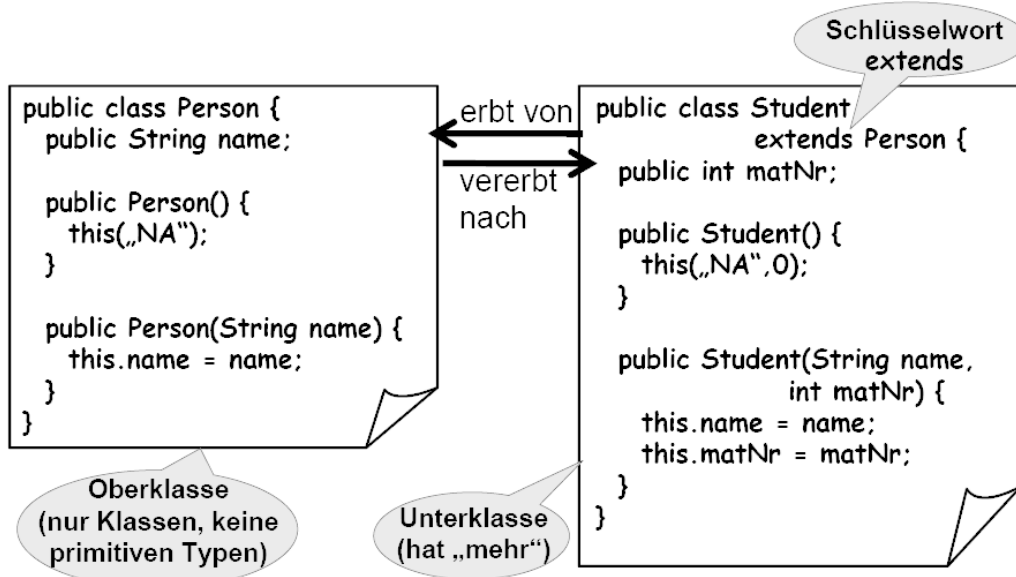
- Eltern geben ihren Kindern Eigenschaften mit.
- Eltern heißen Oberklasse oder Superklasse oder Base Class oder Elternklasse.
- Kinder heißen Unterklasse oder Subklasse oder Derived/abgeleitete Klasse oder Kindklasse.
- Vererbte Eigenschaften (von oben nach unten) sind: sichtbare Attribute (Struktur) und die Methoden (Verhalten).
- Syntax in Java:


```
class Subclass extends Superclass { ... }
```
- Die Unterklasse kann übernehmen und spezialisieren/überlagern, aber nicht eliminieren.
- Vererbung bindet Klassen dichter aneinander: Klassenhierarchien (diese können beliebig tief geschachtelt sein)



- Klasse von der alle Klassen in Java automatisch direkt oder indirekt erben:
`java.lang.Object`

Beispiel für eine Vererbung in Java:

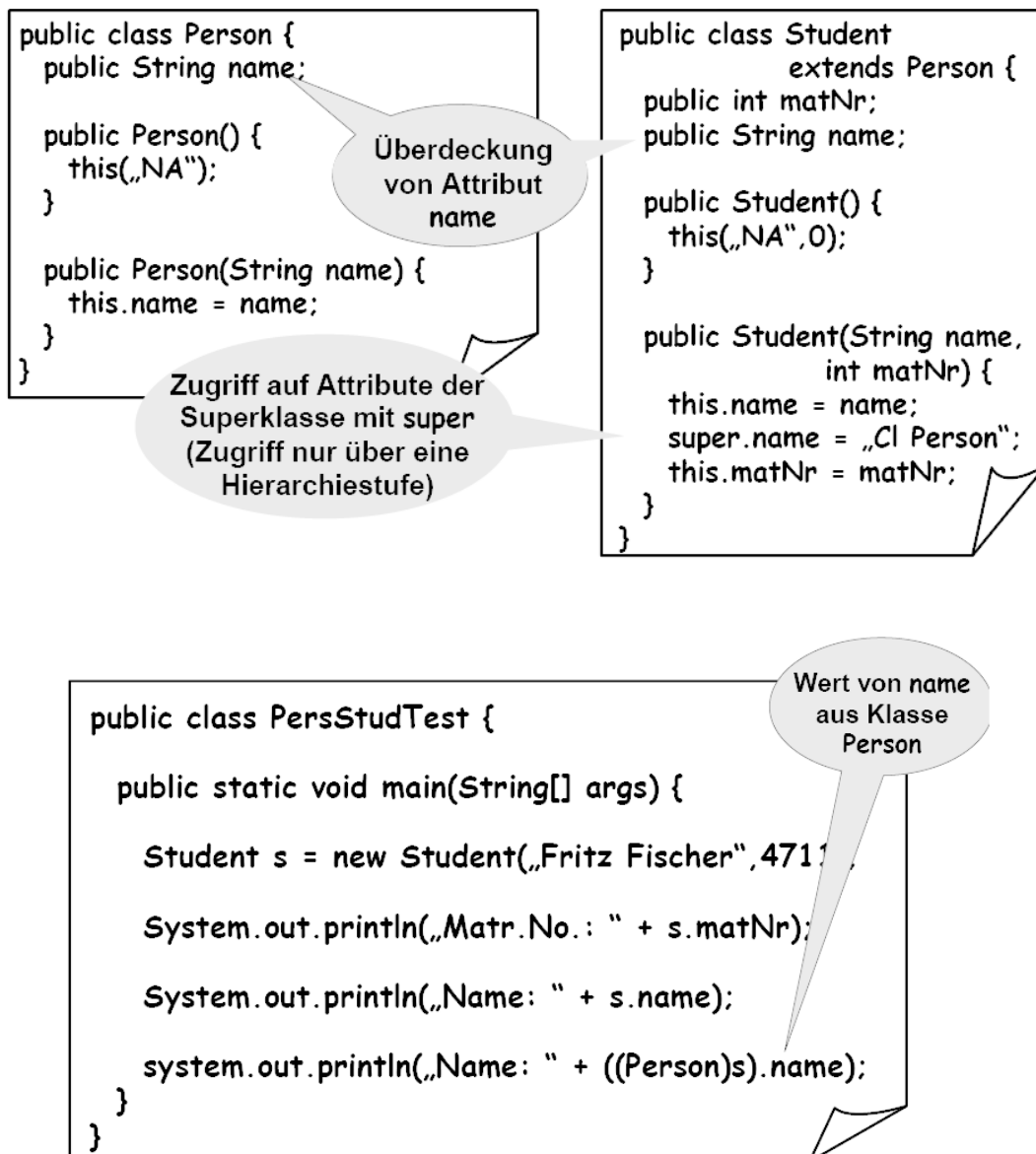


Bemerkung:

Im oberen Beispiel wird in Klasse **PersStudTest** direkt auf die Attribute, die **Student** von der Superklasse erbt, zugegriffen. Auch hier gilt: Ein direkter Zugriff auf Attribute sollte vermieden werden! Das Beispiel soll daher kein Vorbild sein, sondern dient nur der Demonstration der Zusammenhänge.

Durch Vererbung können Klassenelemente (Attribute oder Methoden) überdeckt werden. Ein expliziter Zugriff auf die Elemente der eigenen Superklasse ist durch das Schlüsselwort **super** möglich. Auf diese Weise kann aber nur explizit auf Elemente der direkten, eigenen Superklasse zugegriffen werden. Eine Verkettung von **super.super....** ist in Java nicht möglich.

In anderen OO Sprachen gilt meist Ähnliches.



Jenseits von **super** kann teils auch durch explizite Typanpassung das Element einer bestimmten Klasse in der Vererbungshierarchie ausgewählt werden.

Während **super** nur in der eigenen Klassenhierarchie funktioniert, können Typanpassungen auch von außen angewendet werden. Die Instanz von **PersStudTest** ist beispielsweise nicht in der Person-Student-Klassenhierarchie, kann aber durch die Typanpassung **(Person)s** von außen (d.h. außerhalb dieser Klassenhierarchie) festlegen, dass auf das Attribut **name** aus der Klasse **Person** zugegriffen werden soll.

Zur Typanpassung und ihren Regeln werden wir später noch etwas mehr sehen.

Beispiel einer sinnvollen Vererbung in Java:

Jede Klasse ist für ihre eigenen Attribute zuständig. Klasse **Person** hat das Attribut **name** und sorgt mit Methoden für dessen „Verwaltung“. Klasse **Student** ist für die spezifischen studentischen Attribute zuständig und sorgt so z.B. für die Initialisierung und „Verwaltung“ der Matrikelnummer. Jenseits der Attribute werden auch Methoden in die Unterklasse vererbt.

```

public class Person {
    private String name;

    public Person() { this("NA"); }
    public Person(String name) {
        this.setName(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

Konvention:
Attribut ist privat
mit
getter/setter
Methoden

```

public class Student
    extends Person {
    private int matNr;
    ...
    public Student(String name,
        int matNr) {
        this.setName(name);
        this.setMatNr(matNr);
    }
    ...
    public void setMatNr(int matNr) {
        this.matNr = matNr;
    }
    public int getMatNr() {
        return matNr;
    }
}

```

Student erbt
Methode getName
aus Oberklasse
Person

Nutzung:

```

Student s = new Student("Fritz Fischer", 4711);
System.out.println("Name: " + s.getName());

```

Überschreiben von Methoden:

```

public class Person {
    ...
    public String getName() {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
    ...
}

```

```

public class Student
    extends Person {
    ...
    public String getName() {
        return ("Name:" +
            super.getName());
    }
    public void setName(String name) {
        super.setName(name);
    }
}

```

Spezialisierung von Methodencode:
Student erbt Methode getName aus Oberklasse
Person, überschreibt diese dann aber durch eine
eigene („bessere“) mit gleicher Signatur +
Rückgabetyt (bzw. Untertyp in der Unterklasse)

Expliziter Zugriff
auf Methoden der
Oberklasse mit
super

Ist eine Methode überschrieben, wird bei ihrer Verwendung (z.B. bei `s.getName()`) die Methode aufgerufen, die in der Instanz (hier `s`) direkt definiert wurde. Nur wenn dort keine definiert ist, wird die nächste in der Vererbungshierarchie (nach oben) gerufen. Hätte ein `Student` demnach keine Methode `getName()` würde die Methode aus `Person` gerufen.

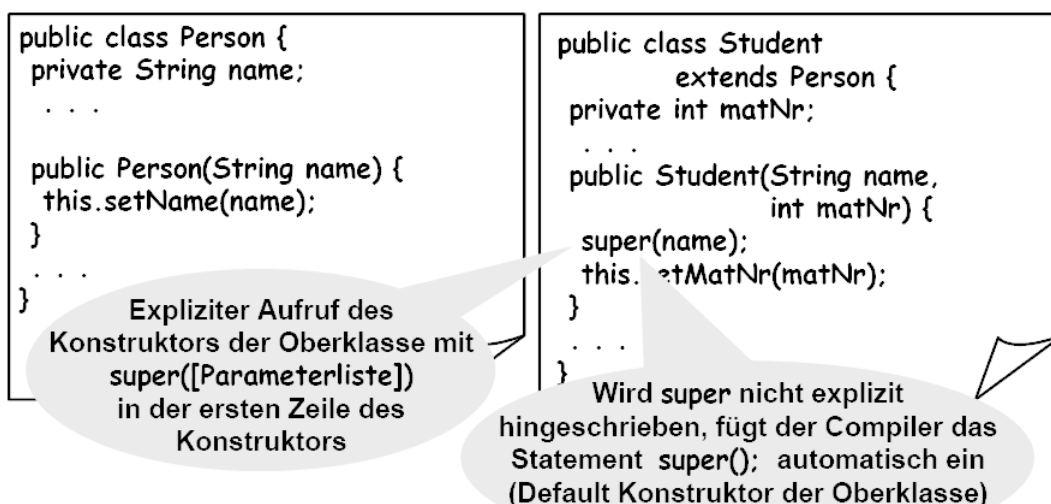
Begriffe Überladen, Überschreiben, Überlagern, Verdecken:

- Überlagern, Verdecken: Eine Klasse hat Attribute oder Methoden(signaturen), die bereits in der Superklasse enthalten sind. Diese Elemente der Klasse verdecken bzw. überlagern damit die Elemente der Superklasse.
- Überschreiben (engl. Overriding): *Überschreiben* ist eine besondere Form der *Überlagerung*. Sie betrifft insbesondere Methoden. Bei Methoden gibt es eine dynamische Bindung (die wir noch genauer betrachten werden). Durch diese dynamische Bindung „verschwinden“ die Methoden in der Superklasse, d.h. man kann auf sie nicht mehr ohne weiteres zugreifen. Sie sind also überschrieben. Auf die Attribute der Superklasse kann jedoch noch zugegriffen werden: sie werden nicht überschrieben, sondern nur durch ein gleichnamiges Attribut überlagert.
- Überladen (engl. Overloading): Eine Methode ist überladen, wenn es in der Klasse oder der Klassenhierarchie eine gleichnamige Methode gibt, die sich in den Parametern (Anzahl oder Typen) unterscheidet. Im Gegensatz zum *Überschreiben*, sind überladene Methoden aus Sicht der Programmierungsumgebung zueinander vollkommen unabhängige Methoden.

Konstruktoren in der Vererbung:

- Konstruktoren werden (trotz ihrer Ähnlichkeit zu Methoden) nicht vererbt.
- Die Initialisierung der eigenen Attribute sollte jede Klasse selbst übernehmen (Verantwortlichkeitsprinzip).
- In allen Vererbungshierarchien müssen die Attribute sinnvoll belegt werden.
- In einem Konstruktor wird immer als erstes (automatisch) der Konstruktor der Superklasse aufgerufen. Das ist in Java so, aber in der Regel auch in anderen OO Sprachen.

Hat die Superklasse in Java keinen passenden Konstruktor, wird ein Fehler gemeldet.



Schlüsselwort `super` in Java (analog zu `this`):

```
public class Student extends Person {
    private int matNr;
    ...
    public Student(String name,
                    int matNr) {
        super(name);
        this.setMatNr(matNr);
    }
    public void setName(String name) {
        super.setName(name);
    } ...
}
```

Achtung: Zwei Bedeutungen von `super`:

1. Aufruf des Konstruktors der Oberklasse
2. Referenz auf die eigene Instanz interpretiert als Instanz der Oberklasse

Finale Klassen: „Überschreiben verboten“

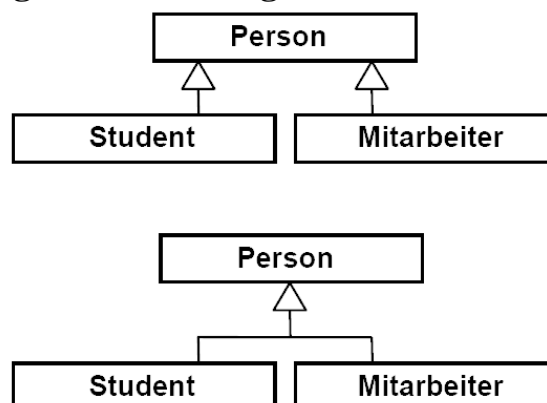
Das Schlüsselwort `final` kann in Java an Klassen, Methoden und Attributen verwendet werden:

- Finale Klassen haben keine Subklassen.

```
public final class Student extends Person {
    ...
}
```

- Finale Methoden können nicht überschrieben werden.
- Finale Attribute können nach der Initialisierung nicht mehr verändert werden (= Konstanten).
- Beispiele finaler Klassen in Java: `String`, `Math`, `System`

Die genannten, expliziten Einschränkungen mit Hilfe des Schlüsselworts `final` in Java finden sich auch in anderen OO Sprachen. Das Schlüsselwort kann hier jedoch variieren.

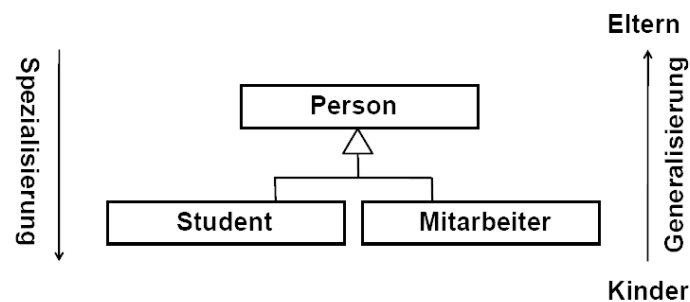
Grafische Darstellung der Vererbung in UML:

Vorgehensweise: Generalisierung und Spezialisierung

Eine Vererbung kann auf zwei Arten realisiert werden: durch Spezialisierung oder durch Generalisierung. Durch Spezialisierung wird eine Klasse in speziellere Subklassen verfeinert. In einer Generalisierung wird aus einer Menge von Klassen eine generellere Superklasse ermittelt.

Das Ergebnis kann (wie im folgenden Beispiel) dasselbe sein. Die Vorgehensweise zum Ergebnis unterscheidet sich jedoch.

In einer objektorientierten Entwicklung werden beide Vorgehen zur Gestaltung der Lösung angewendet.



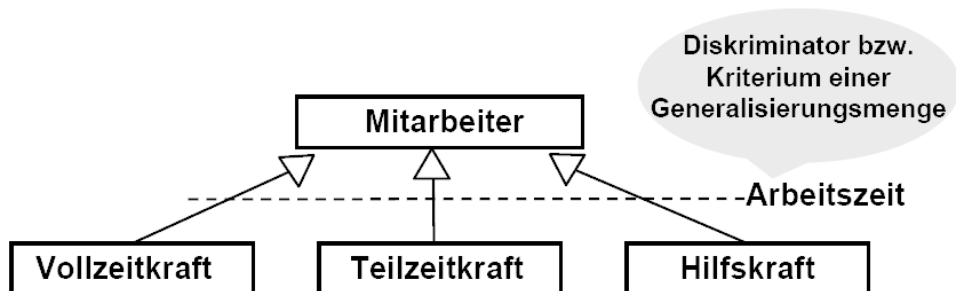
Diskriminator

Eine Vererbung kann nach verschiedenen Kriterien realisiert werden. Das Kriterium wird auch *Diskriminator* genannt.

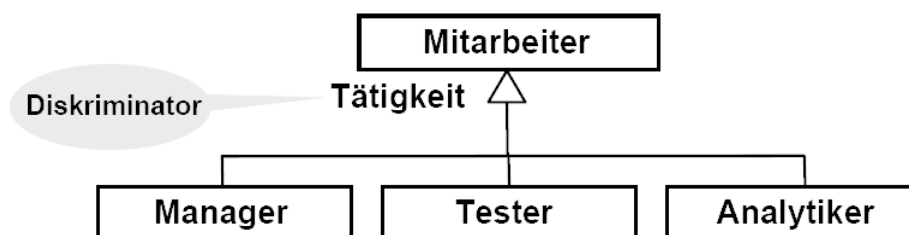
Ein Mitarbeiter kann beispielsweise nach dem Kriterium „Arbeitszeit“ verfeinert werden. Ist eine Menge von Subklassen nach dem gleichen Kriterium verfeinert, heißt dies auch homogene Spezialisierung bzw. Generalisierung.

Eine Vererbung mit Diskriminator kann in UML auf zwei Varianten dargestellt werden. Beide im Beispiel dargestellte Varianten sind homogene Generalisierungen.

Variante 1:

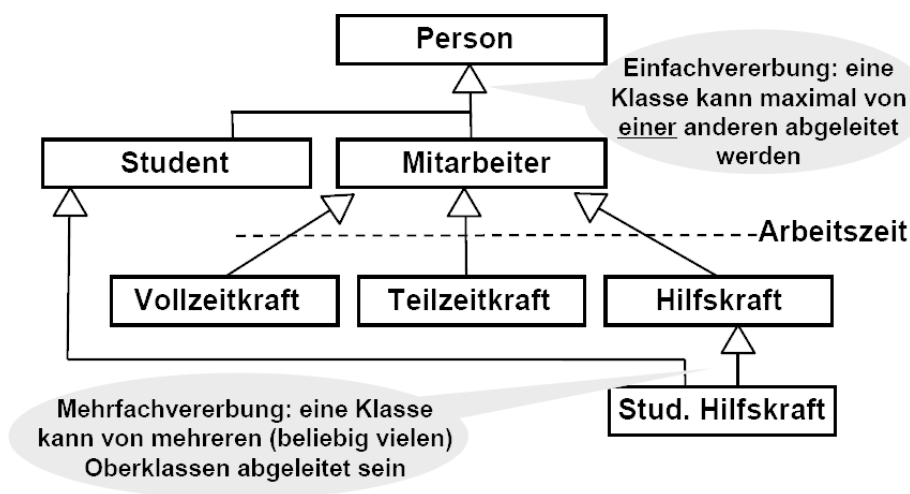


Variante 2:

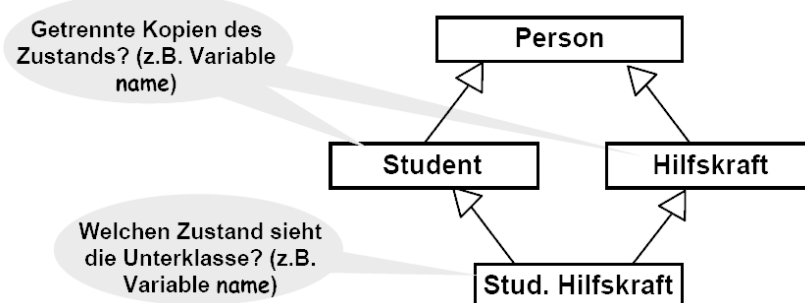


Formen der Vererbung:

- Einfachvererbung (engl. Single Inheritance) versus Mehrfachvererbung (engl. Multiple Inheritance)
 Eine Einfachvererbung erlaubt nur eine Superklasse pro Klasse. Eine Mehrfachvererbung erlaubt eine bis viele Superklassen zu einer Klasse.
 Mehrfachvererbung wird z.B. in C++ unterstützt; in den Sprachen Smalltalk und Java ist sie nicht möglich.
 Wir werden gleich sehen, dass eine Einfachvererbung weniger Probleme (aber natürlich auch weniger Möglichkeiten) mit sich bringt als eine Mehrfachvererbung.
- Abstrakte versus konkrete Vererbung (bzw. Spezifikationsvererbung versus Implementationsvererbung)
 Subklassen in einer Implementationsvererbung bekommen den Code aus den Superklassen (die Implementierung wird vererbt). In einer abstrakten Vererbung (Spezifikationsvererbung) bekommen die Subklassen nur die Spezifikationen vererbt (also vorgeschrieben). Sie müssen die Implementierung hierzu passend selbst ergänzen.
 Abstrakte Vererbung wird mittels abstrakter Klassen oder Interfaces realisiert. Sie sind gleichzeitig ein Hilfsmittel für eine restriktivere Art der Mehrfachvererbung.
 Wir werden uns diese Unterscheidung bzw. die abstrakte Vererbung weiter unten noch genauer anschauen.
- Strikte Vererbung (einfache Vererbung)
 In den Subklassen kommen weitere Eigenschaften hinzu. Es wird aber nichts überschrieben oder überlagert.



Probleme der Mehrfachvererbung: Das Diamond Inheritance Problem



Problem: Welche Variablen(werte) und Methoden(implementierungen) werden vererbt?

„Die oberste Superklasse“

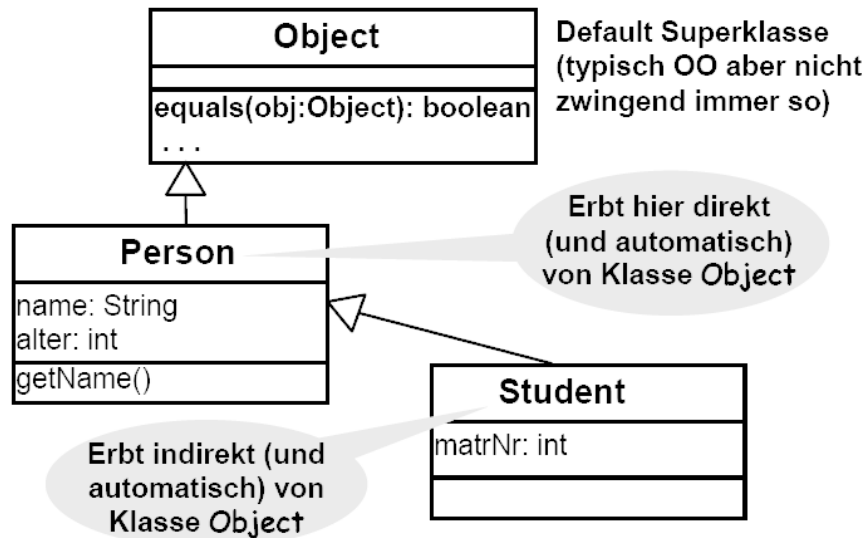
Die meisten objektorientierten Sprachen haben eine Klasse, die in der Vererbungshierarchie ganz oben steht und von der alle Klassen automatisch erben.

In Java heißt diese Klasse (wie häufig) **Object**.

Die Klasse **Object** enthält allgemeine Eigenschaften wie z.B. die Methode **equals()**.

Dadurch haben alle Klassen automatisch (durch Vererbung) eine Standardimplementierung für **equals()**.

Klassenhierarchie in Java:



Nutzen und Probleme der Vererbung:

- Gewinn aus der Vererbung: Wir erreichen eine Modularisierung, Abstraktion, Wiederverwendung und Schnittstellenbildung (mittels abstrakter Vererbung).
- Aber: Es gibt keine expliziten Grenzen mehr ("Fragile Base Class Problem"). Ändert sich die Superklasse (d.h. die zerbrechliche Base Class), so ändern sich automatisch alle erbenden Klassen. Die Beziehung ist implizit und sehr zerbrechlich. Getrennte Klassen nur mit Schnittstellen und ohne Vererbung sind häufig der bessere Entwurf.
- Alternativen der Vererbung: z.B. Delegation (Nutzung der Dienste einer anderen Klasse durch Aufruf, d.h. Delegation der Aufgabe), Schnittstellen (kommt noch), generische Ansätze (kommt noch), generative Ansätze
Generative Ansätze transformieren den Code. Getrennte Einheiten werden z.B. durch Generation zu einer Einheit verschmolzen, was zu einem ähnlichen Ergebnis wie die Vererbung führen kann: Modularisierung in der Programmgestaltung aber trotzdem Kombination der Möglichkeiten in der Ausführung.

3.3 Exkurs: Typing

Beim Typsystem einer Programmiersprache unterscheidet man zwischen:

- Streng typisierten Sprachen (z.B. Java): Die Typen werden von der Programmiererin angegeben und der Compiler / Interpreter überprüft, ob die Werte zu den angegebenen Typen passen.
- Typisierung durch Typinferenz und zur Laufzeit (z.B. Smalltalk): Die Typen müssen nicht (durchweg) angegeben werden. Anhand der Werte ermittelt die Programmierumgebung selbstständig, welcher Typ verwendet werden muss. Der Typ wird also aus den Werten und deren Weiterverarbeitung abgeleitet (Inferenz). Der Vorteil ist die einfachere Programmierung. Nachteil ist, dass der Compiler / Interpreter nicht mehr prüfen kann, ob die Werte genau wie vom Programmierer erwartet verwendet werden.

Bemerkung:

Typen sind nach der Einteilung von [Boo94] weniger wichtige Konzepte.

Dies bezieht sich auf den Punkt, dass objektorientierte Umgebungen nicht zwingend streng typisiert sein müssen.

Wir behandeln Typen schon in diesem Kapitel, da wir sie später noch als Grundlage brauchen.

Relevante Prinzipien im Rahmen der Typisierung:

- Type Casting / Typanpassungen
- Substitutionsprinzip
- Generische Typen (folgt später)
- Polymorphismus (folgt später)

3.3.1 Typanpassungen: Type Casting / Type Casts

- Typanpassung bzw. Typkonvertierung = Typecast (engl.) = “casten” (umgangssprachlich)
- Eine Typanpassung wird zur Compilezeit ausgewertet (statisch).
- Casts bei primitiven Datentypen bedeutet in der Regel: Einschränkung des Wertebereichs.
- Bei Referenzen werden Casts in OO Sprachen meist wegen des Substitutionsprinzips eingesetzt (im Rahmen der Vererbung): statisch wo möglich, sonst dynamisch.

Ein statischer Type Cast (d.h. zur Übersetzungszeit) behindert die Laufzeit nicht.

Deshalb wird generell versucht, alle Casts zur Übersetzungszeit umzusetzen.

Manche Casts können aber erst zur Laufzeit durchgeführt werden, weil die benötigten Informationen erst zur Laufzeit bekannt sind.

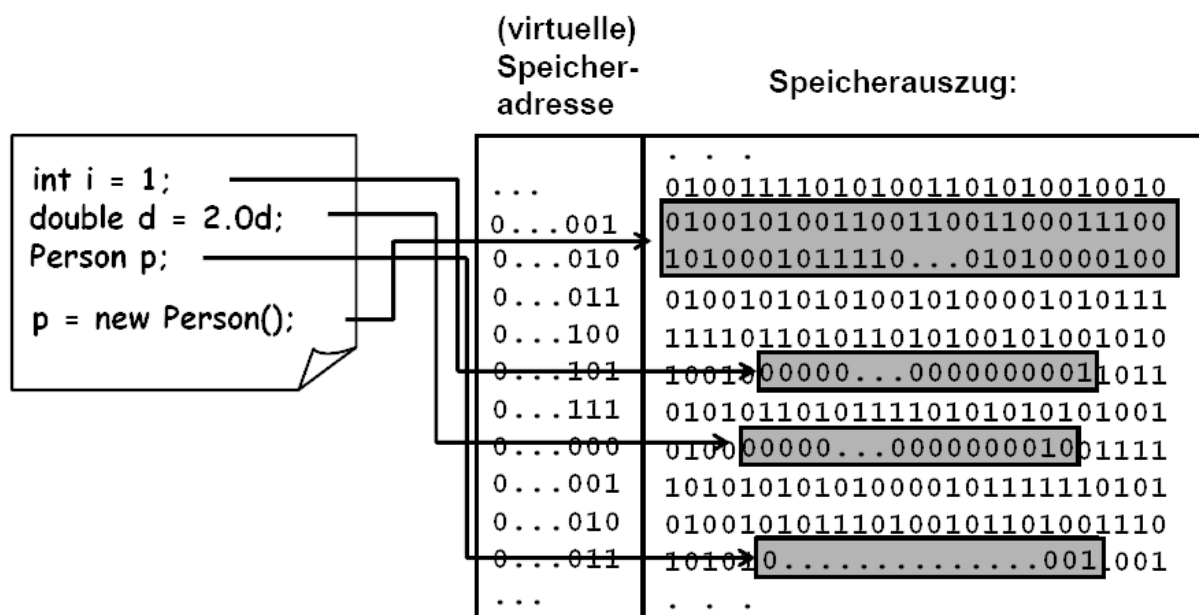
Werden Variablen mit einem bestimmten Datentyp deklariert, wird im Speicher ein entsprechend großer Speicherplatz reserviert und der Wert der Variablen (binär) dort abgelegt. Im Bild wird, z.B. eine Variable `i` vom Typ `int` angelegt und erhält den Wert 1.

An irgendeinem freien Speicherplatz werden dadurch 32 Bits „reserviert“ und mit diesen 32 Bits wird der Wert dargestellt. Der vom System (Virtuelle Maschine und Betriebssystem) nach einer bestimmten Strategie gewählte freie Speicherplatz hat eine Speicheradresse.

Bei Referenzen wird ein Speicherplatz für die Referenzvariable benötigt (zur Deklarationszeit) und gleichzeitig ein Speicherbereich, an dem die Instanz zum Zeitpunkt der Instanziierung während der Laufzeit abgelegt wird. Während die Referenzvariable häufig im Stack angelegt wird, wird der Speicher für Instanzen öfters im Heap bereitgestellt. Oft wissen

wir erst zur Laufzeit, ob, welche und wie viele Instanzen erzeugt werden, wie groß diese sind und wie lange deren Lebensdauer jeweils ist (z.B. abhängig von Benutzereingaben oder sonstigen statisch nicht ermittelbaren Informationen). Programmierumgebungen versuchen in der Regel die Laufzeit zu optimieren, d.h. sie reservieren schon zur Übersetzungszeit bereits den zur Übersetzungszeit statisch ermittelbaren Speicherbedarf. Was erst zur Laufzeit bekannt ist (z.B. bei dynamischen Listen von Objekten) kann in der Regel auch erst während der Ausführung angelegt werden.

Im Beispiel in der folgenden Abbildung liegt eine durch **new** erzeugte Instanz im Speicher an der Adresse 0...001. Der Inhalt der Referenzvariablen **p** muss damit den Inhalt 0...001 haben. Die Referenzvariable selbst liegt an der Speicheradresse 0...011.



Wenn wir den Rechner mit seinem Speicher betrachten, können wir darin zunächst keine Struktur erkennen. Die Typen sind nicht sichtbar. Der Speicher weiß nichts über Typen. Der Speicher muss also anhand der Typvereinbarungen erst interpretiert werden.

Der Speicher wird hinsichtlich der darin abgelegten Typen interpretiert von:

- dem Compiler,
- der Laufzeitumgebung einer Programmiersprache (in Java: JVM) und/oder
- dem Menschen.

Eine „Interpretation“ heißt dabei, dass ein bestimmter Speicherausschnitt z.B. mit einer **int**-„Brille“ betrachtet wird. Der Inhalt an dieser Speicherstelle wird also als Typformat **int** ausgewertet.

Wurde im Speicher ein bestimmter Bereich für einen bestimmten Typen angelegt (z.B. 32 Bit bei einer **int**-Deklaration), können wir im Anschluss diesen Speicherbereich auch uminterpretieren und den in diesem Speicherbereich abgelegten Inhalt z.B. als Typ **float** betrachten.

Das machen wir mit einem Type Cast. In Java ist ein im Beispiel aufgeführter Type Cast: **(float) i** oder implizit durch entsprechende Zuweisung **float f = i;** (bei einer Variable **i**, die vom Typ **int** deklariert wurde).

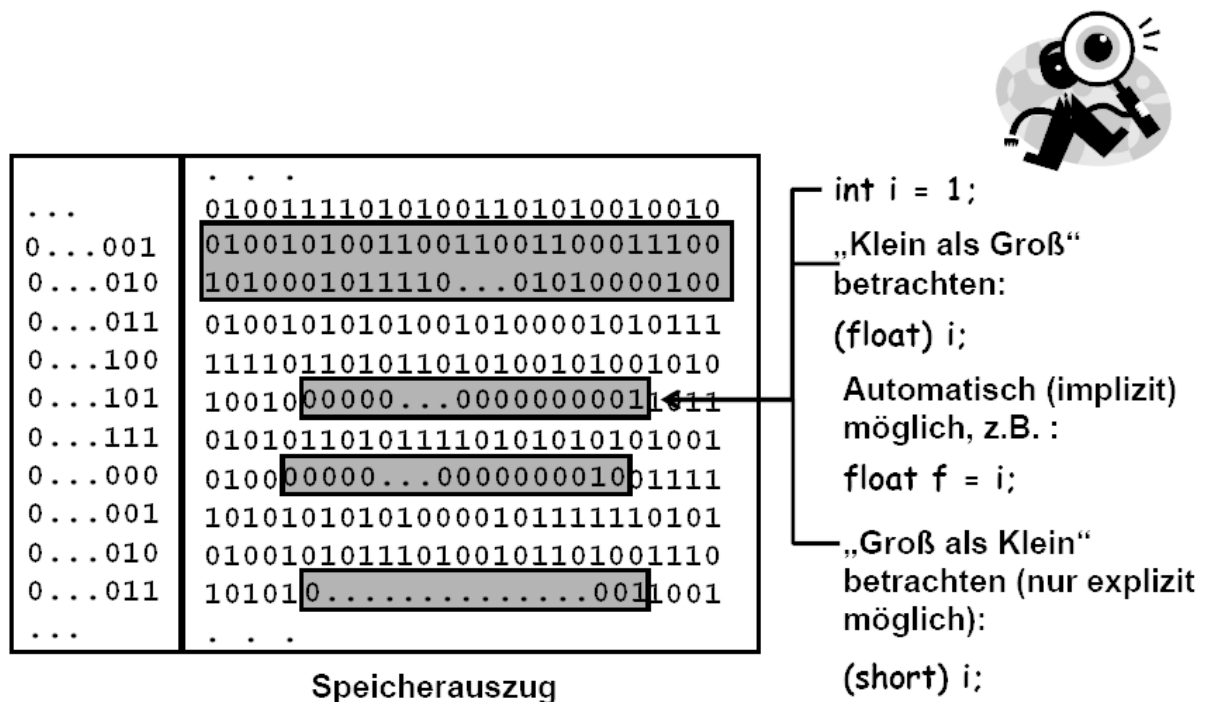
Einen **int**-Speicherbereich als **float** zu interpretieren ist kein Problem und kann das System z.B. im Rahmen einer Zuweisung implizit, d.h. automatisch durchführen.

Die umgekehrte Richtung, d.h. die Interpretation des **int**-Speicherbereichs als vom Typ **short** ist aus Sicherheitsgründen in aller Regel nicht automatisch vom System durchführbar. Der Typ **short** ist kleiner als **int**. Dadurch geht gegebenenfalls ein Stück der Information im größeren Speicherbereich verloren: Wir betrachten nur noch einen Ausschnitt des 32 Bit **int**-Speicherbereichs. Die Programmiererin muß daher genau wissen, ob eine solche Typanpassung den Wert im Speicher nicht zerstört.

Wir können also drei Interpretationsrichtungen unterscheiden:

- ein Speicherbereich wird mit einem größeren Typ interpretiert („Klein als Groß“ = *Widening*),
- ein Speicherbereich wird mit einem kleineren Typ interpretiert („Groß als Klein“ = *Narrowing*),
- ein Speicherbereich wird mit einem anderen Typ interpretiert, der gleich groß ist (d.h. gleiche Bitanzahl hat).

In der Abbildung wird die verschiedene Betrachtung eines `int†`-Speicherbereichs dargestellt.



Widening Conversion: Betrachtung „Klein als Groß“

Nur diese Anpassungen sind in der Regel automatisch vom System durchführbar.

Beispiele in Java:

Gegeben: short sh = 1; byte by = 2; long lo = 3; int in = 4;

- **sh + by** Ergebnis ist vom Typ **int**
(automatischer Cast von **sh** und **by** nach **int** für die interne Berechnung)
- **sh + lo** Ergebnis ist vom Typ **long**
(automatischer Cast der Operanden nach **long** für die interne Berechnung)
genauso für: **by + lo** oder **in + lo**

- (short)(sh + by) ist im Ergebnis wegen explizitem Typcast vom Typ **short**

Folgende Typanpassungen kann Java automatisch durchführen: [JavaInsel09]

Von Typ	In Typ
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Narrowing Conversion (=Typ-Einengung, Down-Cast, Unsafe Casting): Betrachtung „Groß als Klein“

Diese Typanpassungen engen den Typ ein.

Achtung: Durch „Narrowing“ kann sich Informationsverlust ergeben.

Die oberen Bits werden einfach abgeschnitten und es findet keine Anpassung des Vorzeichens statt ⇒ aus Sicherheitsgründen in Java nur explizit möglich.

Beispiele für besondere Effekte bei Casts (mit zugehöriger Bitdarstellung):

```
int m = 123456789;      // 000001101011011100110100010101
short sm = (short) m;   // 1100110100010101
System.out.println(sm); // -13035
int n = -123456;        // 11111111111111000011101110000000
short sn = (short) n;    // 0001110111000000
System.out.println(sn);  // 7616
```

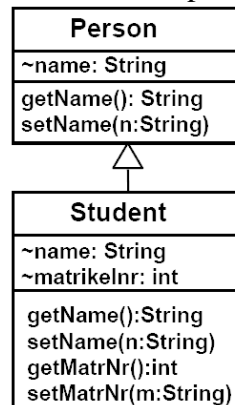
Am Beispiel ist gut zu sehen, dass ein Narrowing zu gegebenenfalls unerwarteten Werten führt.

Konversion zwischen Typen gleicher Bitanzahl:

- In der Regel auch nur explizit möglich und nicht trivial (wegen Unterschieden im Vorzeichen).
- Beispiel: **short**, **char** (beide sind 16 Bit Typen)
`char ch = (char)64000; // Binär 1111010000000000`
 Höchstes Bit ist „1“. Dieses Bit signalisiert, dass wir eine negative Zahl haben, wenn wir mit der Brille „**short**“ interpretieren (ähnlicher Effekt im umgekehrten Fall).

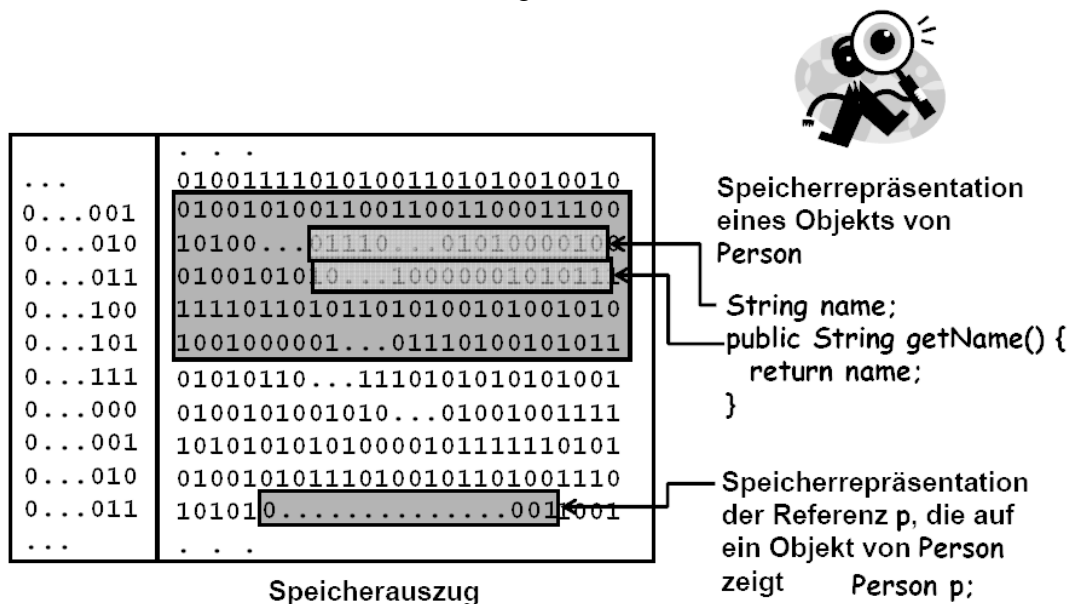
Referenzen und Instanzen im Speicher:

Wir betrachten folgende Programmstruktur als Beispiel im Speicher:



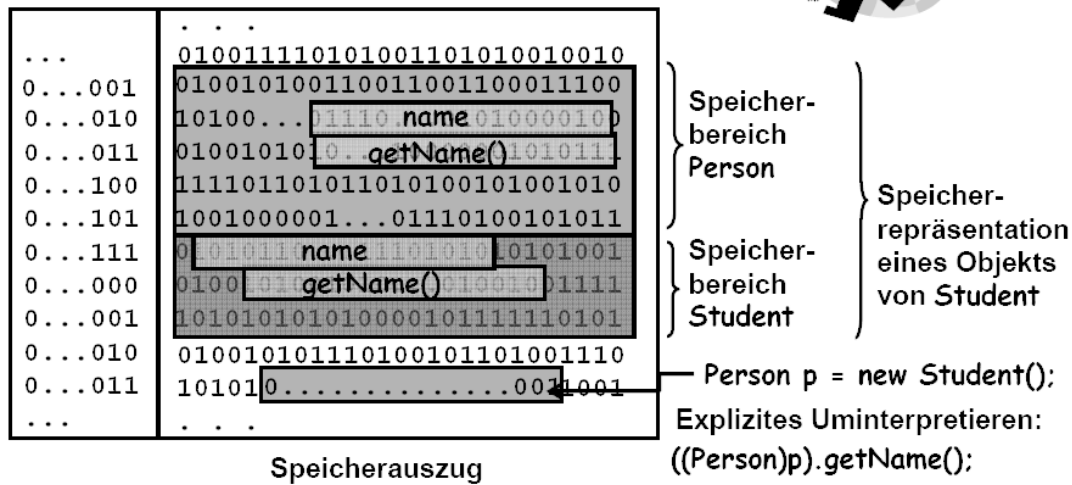
Bemerkung: Der Entwurf dient nur zur Darstellung von Zusammenhängen. Ein guter Entwurf würde einiges daran anders machen (z.B. kein zusätzliches Attribut **name** in **Student**).

In der folgenden Abbildung wird der Speicher vereinfacht dargestellt. Einige Klassen- bzw. Instanzelemente lassen wir vereinfachend weg.



Wird eine Instanz von **Student** erzeugt, so kann man sich die Instanz im Speicher(modell) zusammengesetzt aus einem Speicherbereich für **Person** und dem Speicherbereich für **Student** vorstellen (und auch dem Speicherbereich für **Object** bzw. von weiteren Klassen in der Klassenhierarchie).

Achtung: Wird eine Instanz von **Student** erzeugt, wird nicht automatisch eine zweite Instanz für **Person** erzeugt. Die Instanz **Student** umfasst vielmehr die Speicherbereiche, die für ihre Superklassenelemente notwendig sind.



Bei der Zuweisung einer Instanz von **Student** in eine Variable vom Typ **Person** kann man nicht wirklich von *Narrowing* sprechen. Tatsächlich werden ja nur Referenzen (gleiche Größe und gleiche Interpretation) kopiert.

Wenn wir das Prinzip dennoch übertragen wollen, sieht hier das Ganze so aus:

Eine Instanz vom Typ **Student** hat den Speicherbereich von **Student** und **Person**.

Wenn wir die **Student**-Instanz nun mit der „Brille“ **Person** betrachten, ihn also (z.B. durch die Zuweisung in eine Variable vom Typ **Person**) uminterpretieren, so engen wir den Speicherbereich tatsächlich ein. Methoden und Attribute, die nur in **Student** vorkommen, können dann nicht mehr benutzt werden. Darüberhinaus ist diese Typverkleinerung allerdings gar nicht unsicher und wird im Gegenteil sogar häufig und als guter Programmierstil verwendet. Die Uminterpretation einer **Person** als **Student** (z.B. durch den Type Cast **(Student) p**) ist dagegen unsicher und sollte nicht gemacht werden. Ein kleinerer Speicherbereich mit weniger Inhalten wird als größerer Speicherbereich interpretiert. Werden für diese **Person** z.B. Methoden aus **Student** aufgerufen, merkt dies der Compiler nicht, da er sich auf die Typangaben des Entwicklers verlässt. Erst zur Laufzeit kann festgestellt werden, dass der Entwickler „geschummelt“ hat und der **Student** gar keiner ist.

Experimente am Beispiel:

Person p = new Person();

p.getMatrNr();

⇒ Compilerfehler: Statische Typprüfung

Fehler, weil Typ **Person** keine Methode **getMatrNr()** hat.

Person p = new Student();

p.getMatrNr();

⇒ Compilerfehler: Statische Typprüfung

Fehler, weil Typ **Person** keine Methode **getMatrNr()** hat.

Dass **p** zur Laufzeit eine Referenz auf eine Instanz von **Student** ist, könnte der Compiler theoretisch nur in Ausnahmefällen ermitteln.

```
Person p = new Student();  
p.getName();
```

- ⇒ Kein Compilerfehler: Der Compiler findet in der statischen Typprüfung eine Methode `getName()` in `Person` und meldet daher keinen Fehler.
- Kein Laufzeitfehler: Zur Laufzeit wird die erste Methode `getName()` in der Vererbungshierarchie gewählt (von unten nach oben ausgehend von der Instanz, auf die `p` zur Laufzeit zeigt).

```
Person p = new Student();  
((Student)p).getMatrNr();
```

- ⇒ Kein Compilerfehler: Durch den expliziten Type Cast findet der Compiler in der statischen Typprüfung die Methode `getMatrNr()` in `Student`. Der Entwickler gibt dem Compiler durch den Type Cast „Nachhilfe“. Er teilt ihm explizit mit, dass in `p` eine Referenz auf eine Instanz von `Student` liegen wird.
- Kein Laufzeitfehler: Eine Instanz von `Student` hat tatsächlich die Methode.
- Bemerkung: Sollte sich zur Laufzeit herausstellen, dass in `p` eine Instanz liegt, die nicht zu `Student` angepasst werden kann, würden wir einen Laufzeitfehler erhalten.

```
Student s = new Person();  
((Person)s).getMatrNr();
```

- ⇒ Compilerfehler bereits in der ersten Zeile: Eine Referenzvariable auf eine Subklasse kann keine Referenz auf eine Superklasse aufnehmen. Dies ist die Grundregel im Substitutionsprinzip (mehr dazu folgt weiter unten).

```
Person p = new Person();  
((Student)p).getMatrNr();
```

- ⇒ Kein Compilerfehler: Durch den expliziten Type Cast findet der Compiler in der statischen Typprüfung die Methode `getMatrNr()` in `Student`.
- Laufzeitfehler: Zur Laufzeit stellt sich heraus, dass der Entwickler den Compiler getäuscht hat. In der Instanz `p` gibt es die Methode `getMatrNr()` nicht, da sie tatsächlich eine `Person`-Instanz ist.

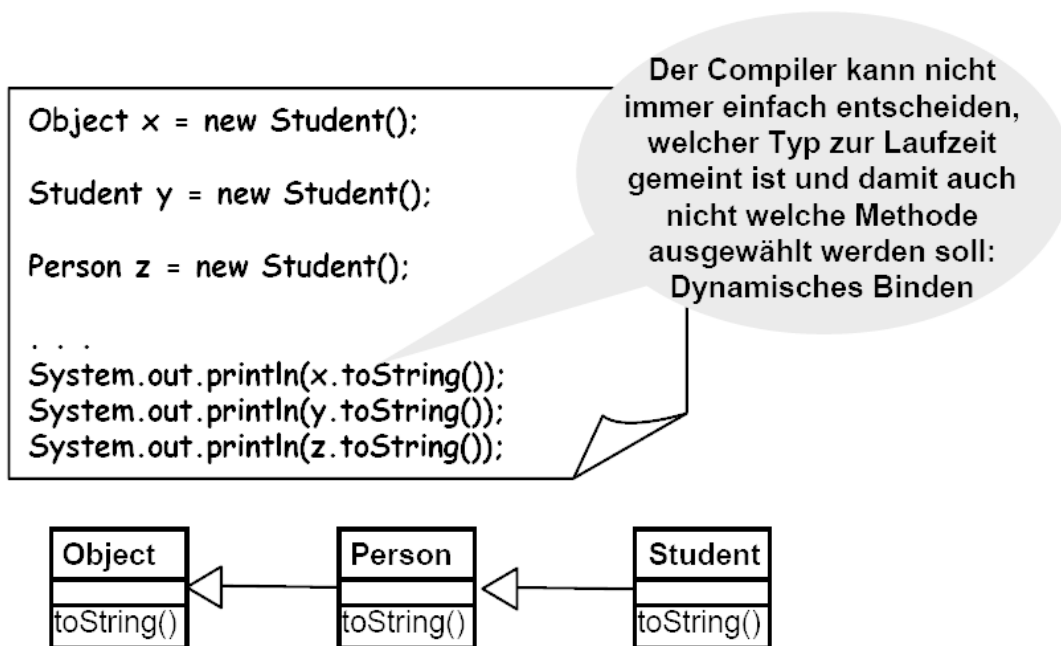
```
Book b = new Book();  
((Student)b).getMatrNr();
```

- ⇒ Compilerfehler: Ein Type Cast verhindert der Compiler, wo die Typen nicht in einer passenden Vererbungshierarchie stehen.

```
Book b = new Student();  
((Student)b).getMatrNr();
```

- ⇒ Compilerfehler: Bereits in der ersten Zeile verhindert der Compiler den Versuch, einen impliziten Type Cast mit inkompatiblen Typen durchzuführen.

3.3.2 Dynamisches Binden



Der Compiler prüft statisch nur die Typen (d.h. im Beispiel, ob der Typ von **z** oder von **x** überhaupt eine **toString()**-Methode aufweist).

Die JVM entscheidet zur Laufzeit, welche Version einer überlagerten Methode tatsächlich gerufen wird. Diese dynamische Entscheidung wird dynamisches Binden genannt.

Zur Laufzeit (also dynamisch) wird die Methodensignatur an eine tatsächliche Methode in der Klassenhierarchie gebunden.

Wie findet der Java-Interpreter die richtige Methode?

- Zunächst wird in der Klasse des beauftragten Objektes geschaut und falls gefunden, wird die Methode ausgeführt.
- Wird die Methode nicht gefunden, wird in der nächst „höheren“ Klasse geschaut, usw.
- Spätestens in der obersten Klasse (**Object**) wird die Methode gefunden und ausgeführt.
- Der Compiler muss dazu Code generieren, der in Verbindung mit der JVM zur Laufzeit entscheiden kann, welche Methode gewählt wird.

Kann es passieren, dass es in der ganzen Klassenhierarchie und auch in **Object** nicht gefunden wird?

Der Compiler schaut in der statischen Typprüfung darauf, dass das prinzipiell nicht passieren kann.

Tricks gegen dynamisches Binden:

- In manchen Sprachen erlaubt die Programmiersprache, selbst explizit Einfluss auf die dynamische Bindung zu nehmen. Beispielsweise kann in C++ die Programmiererin mit dem Schlüsselwort **virtual** die dynamische Bindung selbst beeinflussen. Diese zusätzliche Kontrolle macht die Programmierung schwieriger, erlaubt aber gleichzeitig, ein Programm effizienter zu gestalten. Was der Entwickler weiß, kann der Compiler / Interpreter nicht immer ebenfalls wissen oder ermitteln und berücksichtigen.

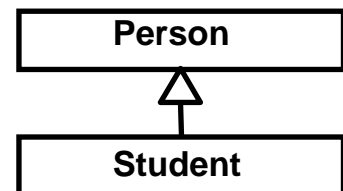
- Einsatz von Modifikatoren, die eine Überlagerung von Methoden unterbinden (und damit eine statische Bindung erzwingen)
 - **private**: in abgeleiteten Klassen nicht sichtbar, damit nicht überlagerbar
 - **final**: explizit vom Programmierer als nicht überlagerbar festgelegt
- Generell gilt in Java: Nur sichtbare Methoden werden dynamisch gebunden (alles andere wie z.B. Attribute, Klassenoperationen usw. werden statisch gebunden).
- Achtung: Type Casts helfen nicht, dynamisches Binden zu verhindern!

Man merke sich drei Grundregeln:

1. Sei B Subklasse von A bzw. A Superklasse von B,
dann geht: A a = new B();
dann geht nicht: B b = new A();
Das besagt das Substitutionsprinzip (dazu folgt unten noch mehr).
2. Der Compiler überprüft immer statisch auf der Basis der im Code angegebenen Typdeklarationen.
3. Eine dynamische Bindung findet zur Laufzeit ausschließlich für sichtbare Methoden statt!

3.3.3 Substitutionsprinzip

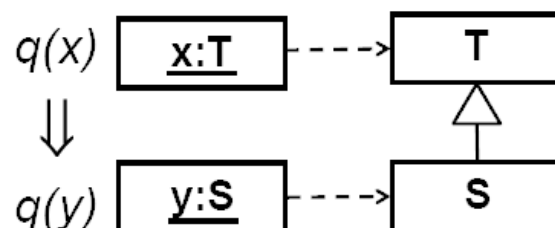
- Substitution:
 - Man kann Objekte vom Typ **Student** so verwenden, als wären es Objekte vom Typ **Person**.
 - Überall, wo ein Objekt vom Typ **Person** gefordert wird, kann auch ein Objekt vom Typ **Student** übergeben werden.



- Idee dahinter: Die Unterklasse ist nur spezieller (hat alle Eigenschaften von Person und Zusätzliches).

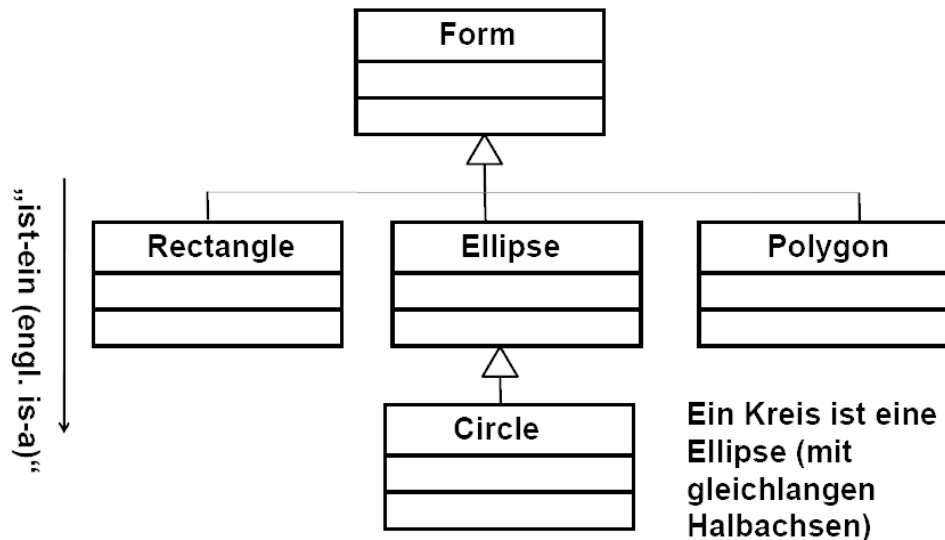
Ursprung des Substitutionsprinzips (Ersetzbarkeitsprinzip) nach Barbara Liskov und Jeannette Wing, 1993:

“Sei $q(x)$ eine beweisbare Eigenschaft von Objekten x des Typs T . Dann soll $q(y)$ für Objekte y des Typs S wahr sein, wobei S ein Untertyp von T ist.”

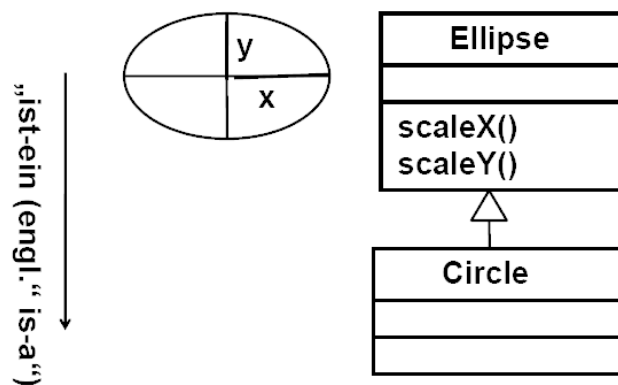


Das Substitutionsprinzip gibt an, wann ein Datentyp als Unterklasse eines anderen Typs modelliert werden soll.

Beispiel zur Anwendung des Substitutionsprinzips für einen Editor mit grafischen Elementen:



Klasse **Circle** erbt hier die Methoden der Klasse **Ellipse**:



Was sagt das Liskovsche Substitutionsprinzip dazu?

Nach dem Liskovschen Substitutionsprinzip ist das angegebene Design nicht gut. Die Superklasse hat Eigenschaften, die die Subklasse nicht hat bzw. nicht haben soll.

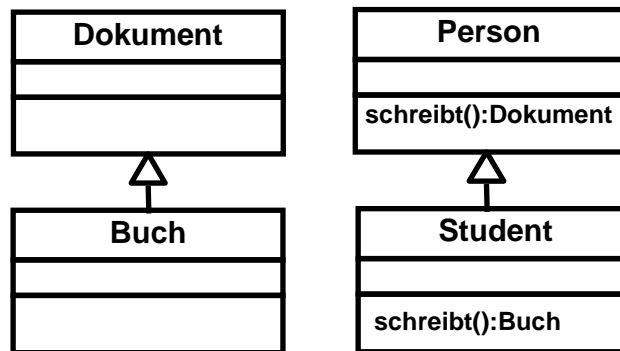
Das Liskovsche Substitutionsprinzip ist ein geeignetes Hilfsmittel, um gute Systeme zu gestalten und Systeme zu bewerten.

Dynamisches Binden unterstützt die Umsetzung des Substitutionsprinzips.

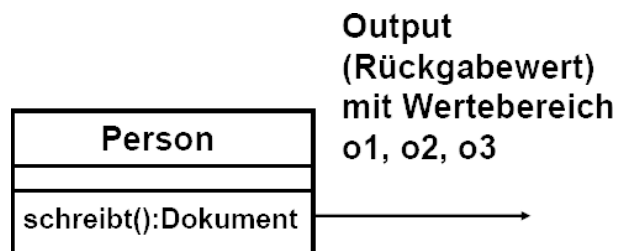
Kovarianz, Kontravarianz und Invarianz

Im Rahmen des Substitutionsprinzips stoßen wir auf die Konzepte *Kovarianz*, *Kontravarianz* und *Invarianz*. Zur Erläuterung der Konzepte betrachten wir ein Beispiel.

Beispiel für Kovarianz des Rückgabewerts: „Typhierarchie des Rückgabetyps geht mit der Vererbungshierarchie“ (co = „mit“)



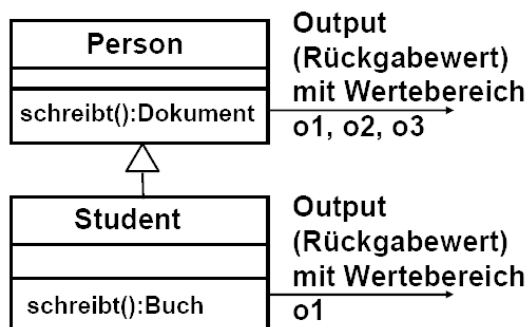
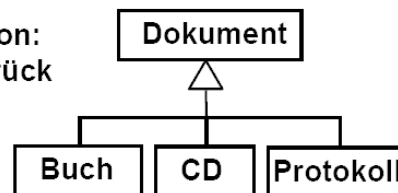
Eine Person liefert damit in der Methode `schreibt()` folgende Daten zurück (die Klasse ist in der folgenden Abbildung in UML-Darstellung; die Rückgabe ist nicht UML-konform):



Die Rückgabewerte haben beispielsweise folgenden Typaufbau:

Beweisbare Eigenschaft von Person:
Sie liefert irgendein Dokument zurück

z.B. o1 = Buch,
o2 = CD,
o3 = Protokoll



Beweisbare Eigenschaft von **Person**: Sie liefert irgendein Dokument zurück

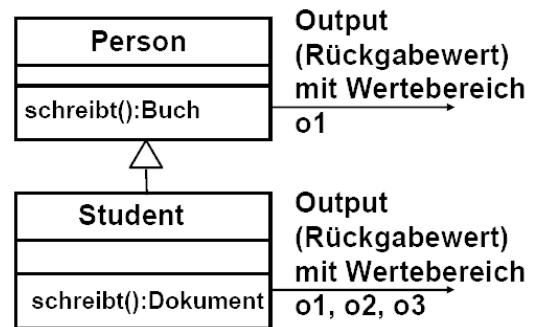
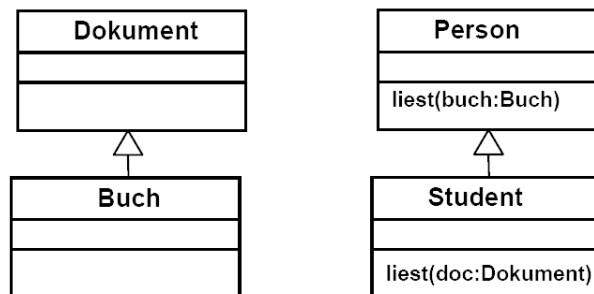
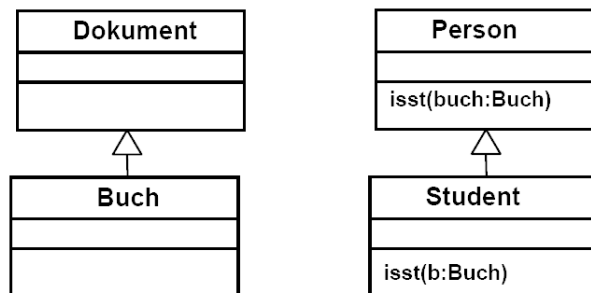
⇒ Die Subklasse **Student** muss nach dem Ersetzbarkeitsprinzip ebenfalls irgendein Dokument zurückliefern.

Das ist hier erfüllt, denn ein Buch ist durch die Vererbungshierarchie gleichzeitig auch ein Dokument.

Beispiel für Kontravarianz im Rückgabebetyp einer Methode:

Beweisbare Eigenschaft von **Person**: Sie liefert irgendein Buch zurück

⇒ Die Subklasse **Student** muss nach dem Ersetzbarkeitsprinzip ebenfalls irgendein Buch (und nicht mehr!) zurückliefern.
Das Substitutionsprinzip ist nicht erfüllt, denn eine CD ist z.B. kein Buch. Ein **Student** könnte aber eine CD als Rückgabewert haben.

**Beispiel für eine Kontravarianz des Methodenparameters: Typhierarchie des Methodenparameters geht entgegen der Vererbungshierarchie****Beispiel für eine Invarianz im Methodenparameter: Typhierarchie unverändert**

Welche Varianzen müssen gelten, damit das Liskovsche Substitutionsprinzip erfüllt ist?
Wir haben oben schon gesehen, dass der Rückgabebetyp entweder invariant oder mindestens kovariant sein muss. Methodenparameter müssen invariant oder mindestens kontravariant sein.

Varianzen in Zusammenfassung:

- Kovarianz: Die Typhierarchie geht mit der Vererbungshierarchie (co = „mit“).
- Kontravarianz: Die Typhierarchie ist entgegengesetzt zur Vererbungshierarchie.
- Invarianz: Die Typen dürfen nicht geändert werden.

Wir haben Varianzen für Rückgabetypen (Ergebnistypen) und Methodenparametertypen (Argumenttypen) gesehen.

Darüberhinaus können die Varianzen auch für sonstige Signaturerweiterungen (z.B. Exceptiontypen) und generische Klassenparameter oder Arrays betrachtet werden.

In Java gilt aktuell und seit Java 5:

- Kovarianz im Rückgabetyp und in den Ausnahmen (Exceptions),
- Invarianz in den Methodenparametern.

Java unterstützt aktuell keine Kontravarianz in den Methodenparametern obwohl das nach dem Substitutionsprinzip sinnvoll wäre. Kontravariante Parametertypen innerhalb einer Vererbungshierarchie führen in Java zu verschiedenen Methodensignaturen.

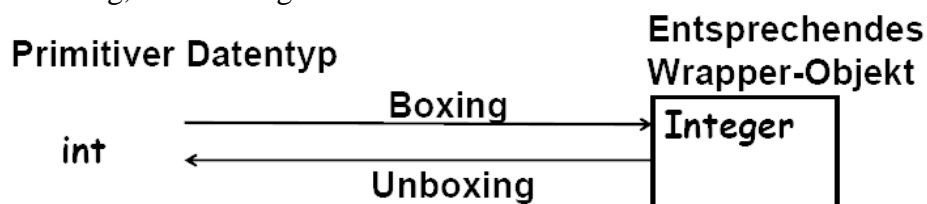
In der OO Modellierung ist eine Kovarianz in den Eingabeparametern oft wünschenswert und wird daher in Programmiersprachen (trotz resultierender größerer Typunsicherheit) teils unterstützt.

3.3.4 Autoboxing

- Problem in Java: Datentypen existieren in zwei Formen (primitiv und als Objekt)
Beispiel: `int` versus `Integer`
- Umwandlung primitiver Datentypen in das entsprechende Objekt durch eine explizite Methode (oder Konstruktor):

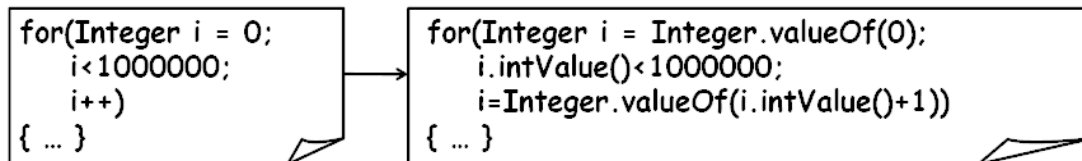
```
Integer i = Integer.valueOf(10);    // Boxing
// new Integer(10);                // Boxing
int j = i.intValue();               // Unboxing
```

- Umwandlung ab Java5 mit dem Prinzip Autoboxing:
Direkte, automatische Umwandlung eines primitiven Datentyps in das entsprechende Objekt und umgekehrt.
- Boxing, Unboxing, Autoboxing:



- mit Autoboxing: `Integer i = 10;`
`int j = i;`
- Autoboxing unterstützt das einfachere Arbeiten mit primitiven Datentypen.
(Beispiel: Java wandelt einen primitiven Datentypen selbst in das jeweilige Wrapper-Objekt um, wenn er in eine Collection eingefügt werden soll.)
- Compilation eines Programms, das eine Autoboxing-Anweisung enthält: Im Bytecode werden weiterhin die entsprechenden Umwandlungsfunktionen verwendet (z.B. `intValue` bzw. `valueOf`).

- Probleme beim Autoboxing: der Fall **Integer**
 1. Geschwindigkeitsnachteil bei verkürzter Schreibweise
 2. Inkonsistenz bzw. Abhängigkeit vom Zahlenbereich

Zu 1: Geschwindigkeitsproblematik beim Autoboxing:

Innerhalb der Schleife immer wieder: **Unboxing eines Integer-Objektes + Boxing einer int-Variable**

Zu 2: Beobachtung: Integer-Objekte im Wertebereich [-128;127] werden besonders oft beim Autoboxing verwendet

Geschwindigkeitsoptimierung für die Umwandlung innerhalb dieses Wertebereichs:
Die beim Boxing entstehenden **Integer**-Objekte dieses Wertebereichs werden in einem Pool vordefiniert zur Verfügung gestellt.

⇒ Jedes **Integer**-Objekt innerhalb dieses Intervalls existiert beim Autoboxing nur einmal.

Problem, das sich durch diese Geschwindigkeitsoptimierung ergibt:
Willkürliche Abhängigkeit vom Zahlenbereich.

Die Auswirkungen dieser willkürlichen Abhängigkeit zeigen folgende Beispiele:

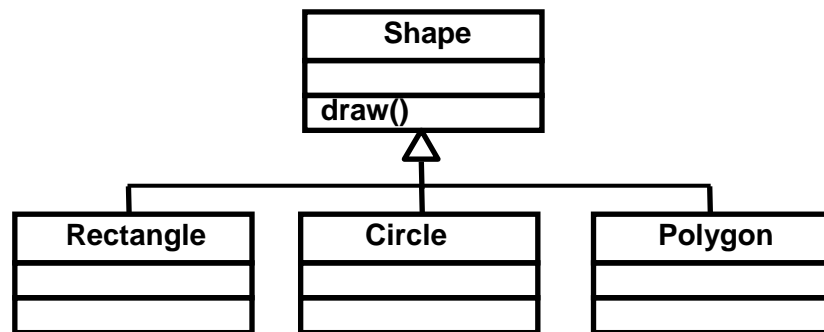
- `new Integer(42) == new Integer(42)`
Liefert **false**, da neue Objekte erzeugt werden.
- `Integer i = 4711; Integer j = 4711; i==j`
Liefert **false**, da neue Objekte erzeugt werden.
- `Integer i = 42; Integer j = 42; i==j`
Liefert **true**, da die Objekte aus dem Pool geholt werden.
- `Integer i = 42; i == new Integer(42)`
Liefert **false**, da ein Objekt aus dem Pool stammt und eines neu erzeugt wurde.

Bemerkung:

Die Bereitstellung einer Vielzahl kleiner Objekte für eine gemeinsame Nutzung (wie z.B. bei den Integer-Zahlen im Wertebereich [-128;127]) ist ein Entwurfsmuster (Muster mit Namen *Flyweight* oder *Object Pool*).

3.4 Abstrakte Klasse

Beispiel: Entwurf eines grafischen Editors:



- Wir wissen, dass sich alle unsere Formen (engl. Shape) zeichnen können sollen (**draw()**).
- Jede Form zeichnet sich aber anders.
- Wir wollen eigentlich gar keine Form (**Shape**) haben ...

Wie gehen wir vor?

Sollen wir die Klasse **Shape** wieder entfernen und **draw()** in alle Subklassen einsetzen?

Wenn sich der Aufruf **draw()** aber für alle Formen gleichzeitig z.B. im Namen ändern soll, dann müssen wir die Methode in allen unseren Formen anpassen. Die redundante Methodenimplementierung ist zudem fehleranfällig, z.B. im Fall inkonsistenter Änderungen.

Wir nutzen **Shape** daher als ein “künstliches Konstrukt” (z.B. als “künstliche Klasse”) um den *Vertrag* auszudrücken.

OO Konzepte zur Realisierung solcher Verträge:

- Abstrakte Klassen
- Schnittstellen (Interfaces)

Beides wird von Java direkt unterstützt.

Die meisten objektorientierten Programmiersprachen unterstützen Verträge durch mindestens eines dieser beiden Konzepte (und gegebenenfalls durch noch weitere).

Dieser Abschnitt behandelt zunächst nur die abstrakten Klassen. Die Alternative (Interface) ist Thema des nachfolgenden Abschnitts.

Eine abstrakte Klasse **Shape** ist (im Gegensatz zu einer konkreten Klasse) grundsätzlich eine “normale” Klasse - es sind aber keine Instanzen der Klasse **Shape** möglich.

Nutzen: Abstrakte Klassen machen Vorgaben für Unterklassen in einer Vererbungshierarchie.

- Vorgabe von Implementierung
- Vorgabe vom Vertrag bzw. der Schnittstelle (Methodensignaturen ohne Festlegung einer Implementierung)

In Java werden Klassen mittels des Modifikators **abstract** zu abstrakten Klassen.

Bemerkung: In C++ ist eine Klasse abstrakt, wenn mindestens eine ihrer Methoden mit dem Schlüsselwort **virtual** deklariert wurde.

Beispiel für eine abstrakte Klasse **Shape** in Java:

```
public abstract class Shape {
    int positionX;
    int positionY;

    public Shape() {
        System.out.println("I'm the Constructor of Shape");
    }

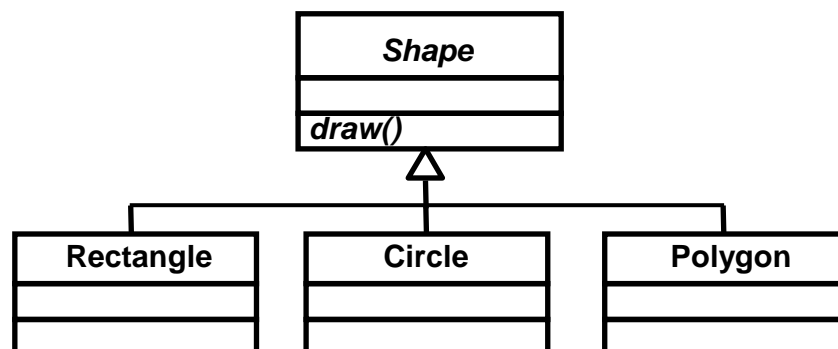
    public abstract void draw();
    public int getPositionX() {
        return (this.positionX);
    }
}
```

abstract ⇒
Shape f = new Shape();
geht nicht

Konstruktor macht
nur im Rahmen der
Vererbungshierarchie Sinn

Abstrakte Methode
⇒ Klasse muss abstrakt sein
(Compiler prüft das)
⇒ darf keinen Rumpf haben
⇒ muss in Unterklassen
(evtl. schrittweise über
mehrere Vererbungsstufen)
implementiert werden

Darstellung abstrakter Klassen im UML Klassendiagramm

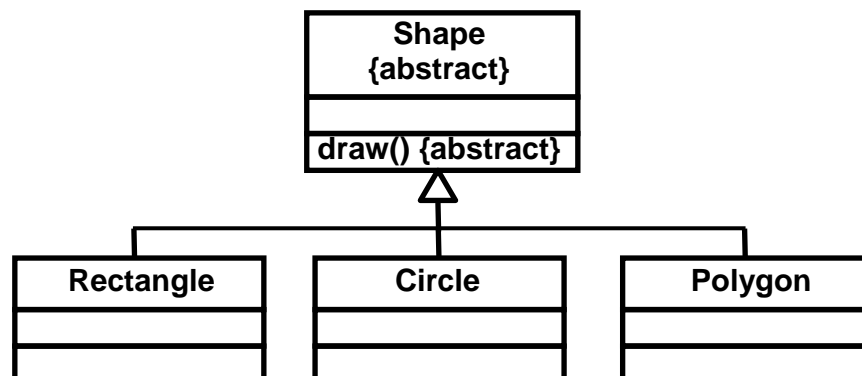


Abstrakte Klassen und Methoden werden in UML kursiv dargestellt.

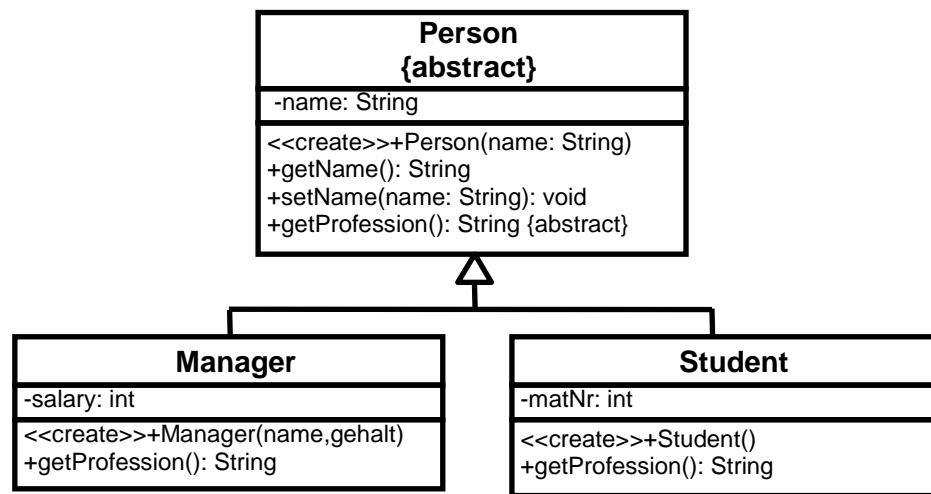
Alternative Darstellung: Kennzeichnung mit **{abstract}**.

Die alternative Darstellung wird häufig verwendet, da eine kursive Schreibweise schnell übersehen wird.

Alternative Darstellung im Beispiel:



Beispiel:



Bewertung

- Abstrakte Klassen definieren eine vertragliche Schnittstelle, die die Unterklassen implementieren müssen.
- Nachteil: Vermischung von Implementierung und Vertrag.
- Abhilfe: Schnittstelle (Interface) als eigenes Konstrukt für eine noch klarere Trennung zwischen Vertrag und Implementierung ...

3.5 Interface

Unabhängig zu allen Programmiersprachen und der Programmebene an sich, gibt es Schnittstellen. Schnittstellen sind eng verbunden mit dem Konzept der Module. Ein gutes Modul hat eine klare und explizite Schnittstelle und wird von außen ausschließlich über diese Schnittstelle angesprochen bzw. benutzt.

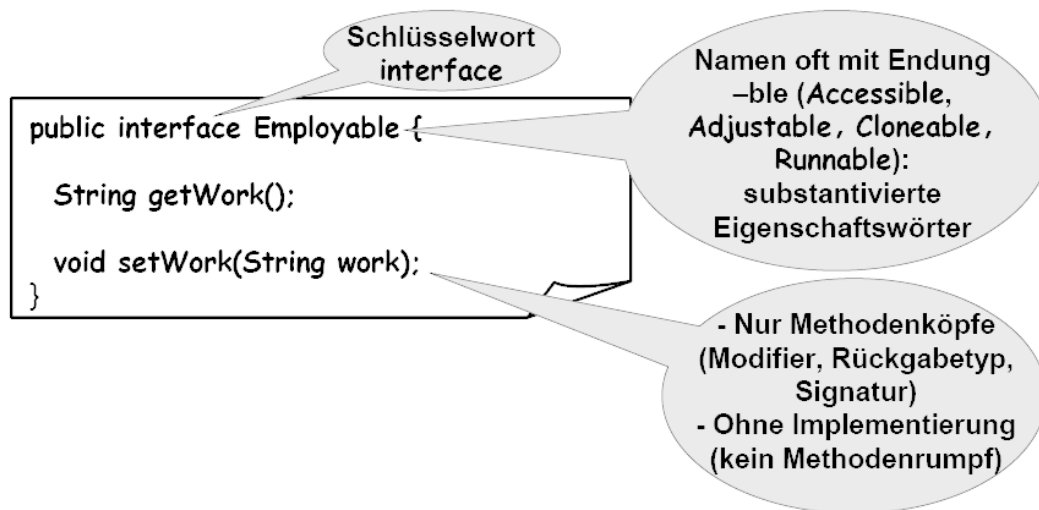
Programmiersprachen unterstützen das Konzept der Schnittstelle wiederum sehr unterschiedlich.

In Java kann eine explizite Schnittstellenbeschreibung mit Hilfe von abstrakten Klassen oder mit dem Konzept **Interface** umgesetzt werden.

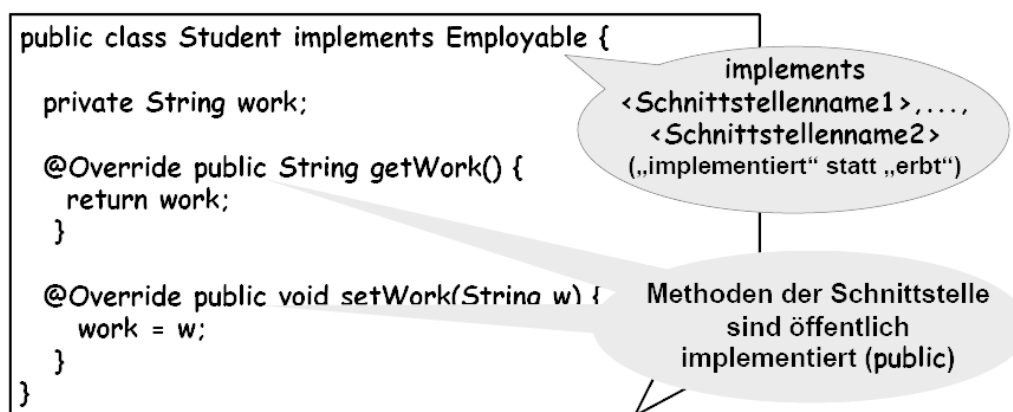
Eigenschaften eines Interface aus Sicht der Programmierung: Ein Interface

- zeigt eine bestimmte Fähigkeit an.
- hat keine Konstruktoren.
- ist immer **public** und auch alle enthaltenen Methoden und Konstanten sind automatisch abstrakt und öffentlich.
- hat keine Instanzvariablen (Ausnahme: **static final** Konstanten können enthalten sein. Umgekehrt sind alle enthaltenen „Variablen“ implizit **final** und **static**).

Beispiel für eine Interface Deklaration in Java: Wir möchten ein neues Interface erstellen.



Beispiel für eine Interface Implementierung in Java: Wir möchten eine Implementierung zu einem vorhandenen Interface erstellen.



Bemerkung zum Beispiel:

Das optionale Schlüsselwort **@Override** zeigt dem Java Compiler an, dass die angegebene Methode hier überschrieben wird. Wenn der Compiler das weiß, kann er kontrollieren, ob eine echte Überschreibung realisiert wurde. Es ist schnell passiert, dass der Entwickler eine Methode versehentlich z.B. überlädt statt überschreibt.

Das Schlüsselwort ist auch in sonstigen Überschreibungen im Rahmen der Vererbung einsetzbar.

Implementiert eine Klasse nicht alle Operationen ihrer Schnittstelle, so muss die Klasse als abstrakt deklariert sein.

Ein Interface wird insbesondere für Eigenschaften eingesetzt, die auf Klassen aus unterschiedlichen Klassenhierarchien zutreffen können.

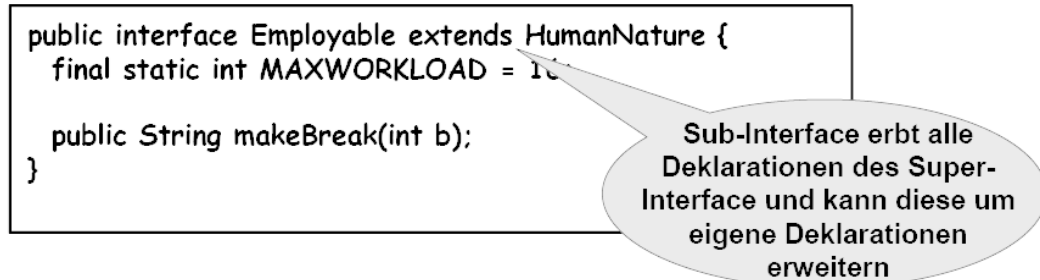
Beispiel in der Java-Klassenbibliothek: **Cloneable**, **Comparable**, **Runnable**

Interface-Vererbung

Analog zu Klassen können Interfaces in Java in einer Vererbungshierarchie stehen. Einige Regeln sind jedoch abweichend zu denen der Vererbung zwischen Klassen.

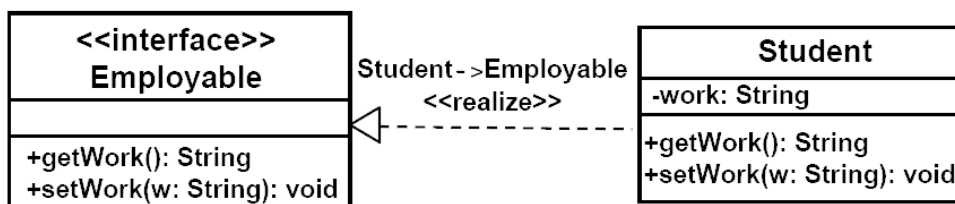
Eine Vererbung zwischen Interfaces wird wie teils auch eine Vererbung zwischen abstrakten Klassen *abstrakte Vererbung* genannt (im Gegensatz zu einer Implementationsvererbung).

Beispiel für eine Interface-Vererbung in Java:

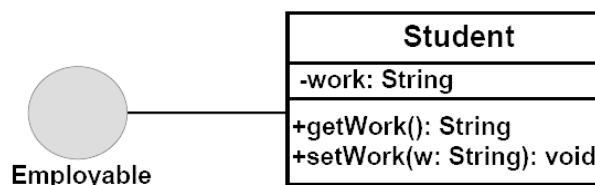


Das Interface im Beispiel hat eine Konstante (in Großbuchstaben per Konvention) und eine Methodendeklaration.

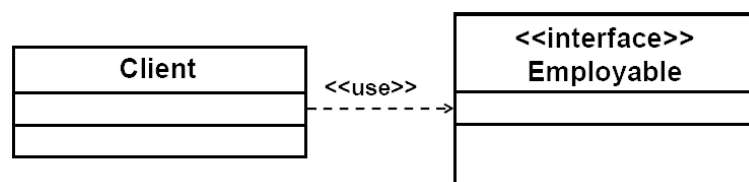
Darstellung einer Schnittstelle in UML



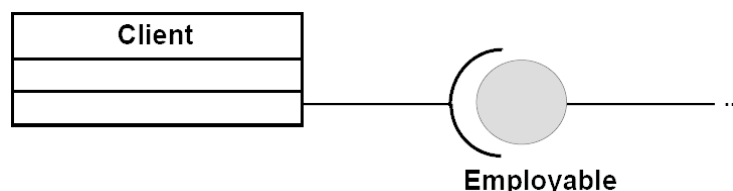
Alternative: „Lollipop-Notation“



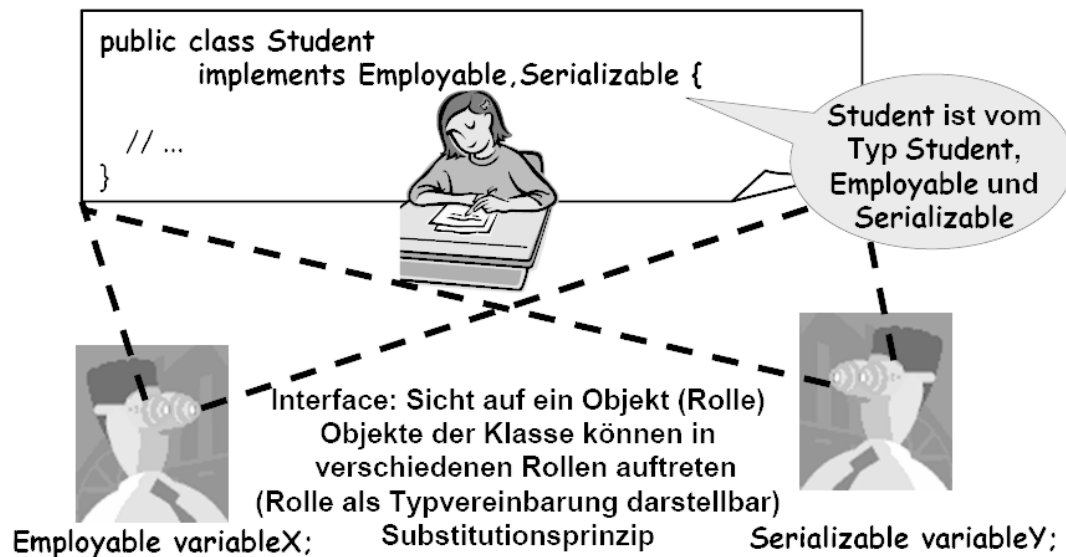
Darstellung einer Schnittstellennutzung in UML



Alternative:



Einsatz und Bedeutung von Schnittstellen



Eine Klasse, die eine bestimmte Schnittstelle implementiert drückt damit aus, dass sie ein bestimmtes, (hoffentlich) sinnvoll abgeschlossenes „Set“ an Eigenschaften realisiert.

Eine Klasse kann auch mehrere Schnittstellen haben.

Jede Schnittstelle stellt eine spezifische Sicht auf die Klasse (bzw. zur Laufzeit auf die jeweilige Instanz) dar.

Eine Instanz kann damit zur Laufzeit verschiedene Rollen spielen.

Im Beispiel kann eine Studentin die Rolle einer Angestellten (z.B. als studentische Hilfskraft) haben oder aber die Rolle eines serialisierbaren Objekts (Interface **Serializable**). Ein serialisierbares Objekt kann in ein Format transformiert werden, das z.B. über ein Netzwerk übertragen werden kann.

Alternativ könnte die Studentin mit Hilfe weiterer Schnittstellen weitere Rollen spielen.

Eine Einteilung in verschiedene Schnittstellen entspricht der Realität. Beispielsweise spielt jeder Mensch je nach Kontext eine andere Rolle. Bei den Eltern spielt ein Mensch die Rolle des Kinds, im Studium die Rolle der Studentin und Kommilitonin, in der Bibliothek die Rolle der Entleiherin, in der Freizeit die Rolle der Trommlerin usw.

Wir könnten nun eine große Schnittstelle erzeugen, die alle diese Eigenschaften enthält.

Beispiel in Java:

```
public interface AllInOneable {
    hug();
    beat();
    borrow();
    ...
}
```

Man kann schnell sehen, dass ein solches „All-in-One“-Interface viele Nachteile hat:

- Das Interface wird sehr schnell sehr groß und unübersichtlich.
- Das Interface ist schlecht wiederverwendbar. Andere Trommler können beispielsweise nur einen Teil der im Interface vorgegebenen Eigenschaften brauchen (im Sinne einer Implementierung des Interface).

- Die prinzipiell klare Rollentrennung wird verwischt. Einige vorgeschriebene Methoden haben in den verschiedenen Kontexten vollkommen unterschiedliche Bedeutungen (z.B. `beat()`).
- Ändert ein Objekt seine Rolle in einem System (z.B. der Student hat sein Studium abgeschlossen und startet eine neue Rolle als Vollzeitangestellter), muss die Schnittstelle überarbeitet werden. Durch die Vermischung von Rollen in der Schnittstelle kann diese Änderung schnell zu Fehlern auch in den Eigenschaften führen, die von dieser Rollenänderung eigentlich nicht betroffen sind. Der ehemalige Student bleibt beispielsweise weiterhin Kind seiner Eltern. Diese Rolle soll also vor der Änderung möglichst gut geschützt sein.

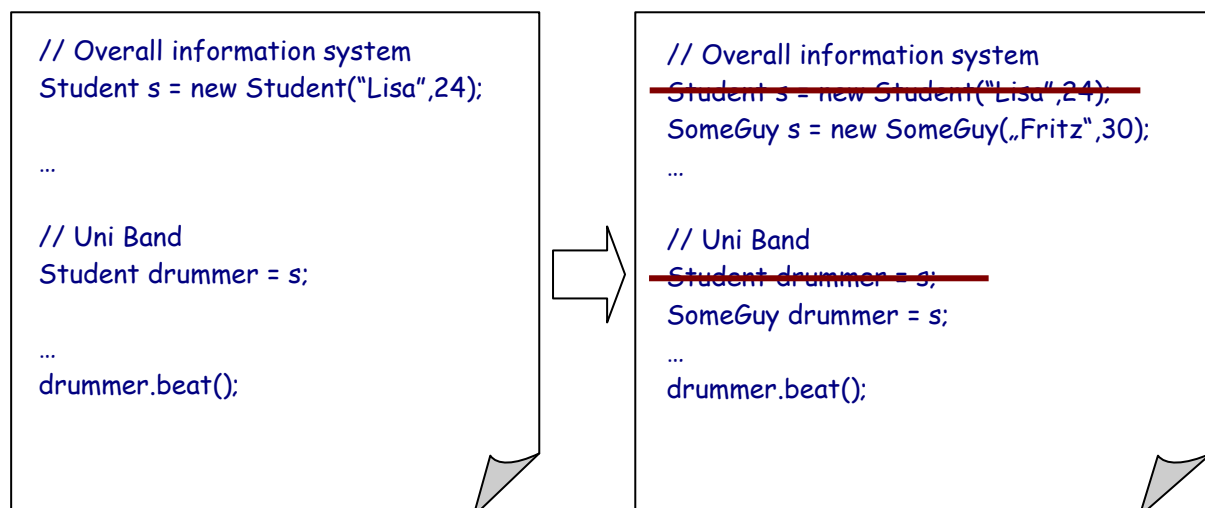
Für die Gestaltung einer Schnittstelle gilt damit Ähnliches wie für die Gestaltung von Modulen im Allgemeinen. Schnittstellen sollten übersichtliche, klar abgegrenzte Einheiten sein. Sie sollten so strukturiert werden, dass nach dem Kohärenz- bzw. Konsistenzprinzip und dem Verantwortlichkeitsprinzip pro Schnittstelle eine klare und zusammenhängende Menge an Eigenschaften zusammengefasst sind. Die Mehrfachimplementierung und Vererbungsbeziehungen zwischen Schnittstellen helfen bei deren Strukturierung.

Das Prinzip, mit unterschiedlichen „Brillen“ auf einen Speicherbereich zu schauen, haben wir bereits bei den Typanpassungen kennengelernt.

Tatsächlich entspricht jede Betrachtung eines Objekts von einer bestimmten Schnittstelle aus einer Typisierung. Unsere Studentin von oben kann also – je nach Betrachtungsweise – vom Typ **Student**, vom Typ **Child**, vom Typ **Serializable** oder vom Typ **Drummer** sein.

Diese Eigenschaft wird in der Programmierung als wichtiges Konzept ausgenutzt: In einem bestimmten Teil meines Programms (z.B. dem Verwaltungssystem der Uni Band) interessiert mich beispielsweise nur die Rolle der Trommlerin. Tritt die Studentin aus der Uni Band aus und wird durch einen anderen Trommler ersetzt, müssten wir unser Programm wie unten dargestellt anpassen, obwohl uns nur die Eigenschaften als Trommler interessieren. Ob der neue Trommler ein Student ist, interessiert uns in diesem Kontext nicht.

Beispiel mit Java Code-Auszug:



Der Einsatz von Schnittstellen ist eine Abhilfe und reduziert den Änderungsaufwand an eigentlich nicht betroffenen Programmstellen.

Statt `Student drummer = s;` oder `SomeGuy drummer = s;` verwenden wir den für diesen Programmteil am besten passenden Typ: `Drummer drummer = s;`

Wir nehmen dabei an, dass der Typ **Drummer** in Form eines Interface vorliegt.

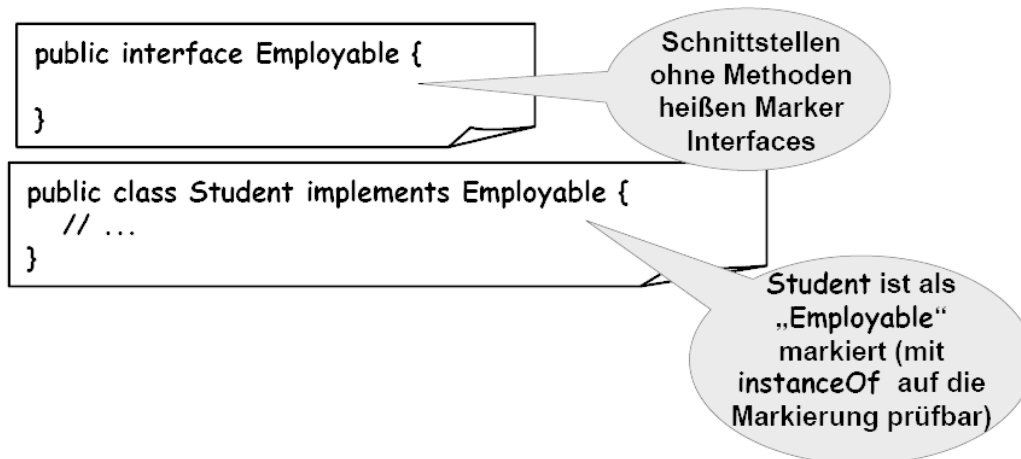
Der so deklarierten Variablen **drummer** können wir alle Referenzen auf **Drummer**-Instanzen zuweisen. **Drummer**-Instanzen entstehen durch Instanziierung aus Klassen, die das **Drummer**-Interface implementieren.

Instanzen haben mehrere Typen:

In Java können wir den Typ mit **instanceOf** prüfen. Das gilt auch für Interface-Implementierungen wenn wir z.B. wissen möchten, ob unsere Instanz vom Typ **Employable** ist.

Eine Instanz hat damit gleichzeitig mehrere Typen: Sie ist vom Typ der Klasse, aus der sie erzeugt wurde, vom Typ aller Superklassen dieser und vom Typ aller Interfaces, die alle diese Klassen implementieren.

Markierungsschnittstelle (Marker Interface, Tag Interface)



Beispiel für ein Marker Interface:

Auf jedem Objekt kann die Methode **clone()** aufgerufen werden.

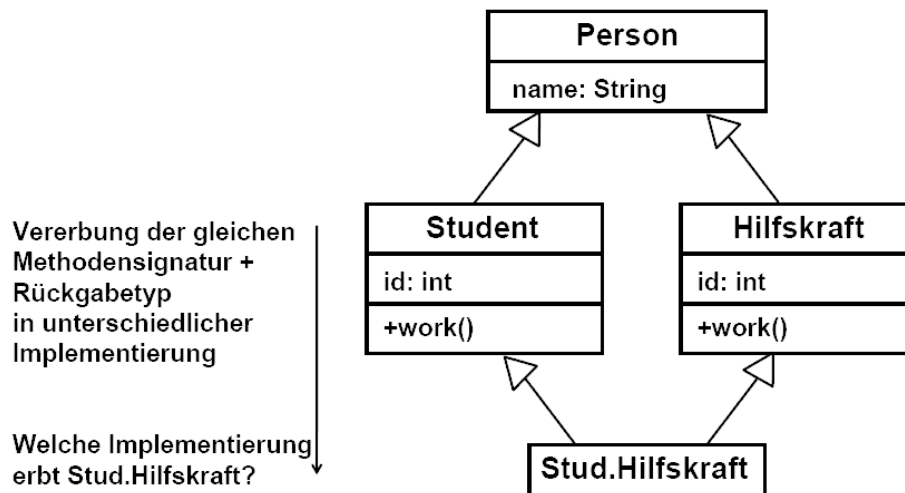
Voraussetzung: Die Klasse des Objekts wurde mit dem Interface **Cloneable** markiert.

Beispiel für ein Marker Interface in der Oracle/Sun-Bibliothek:

```
package java.io;  
interface Serializable {  
}
```

Mehrfachvererbung und Interfaces

Im Rahmen der Vererbung hatten wir das Diamond Inheritance Problem kennengelernt.



Um das Diamond Inheritance Problem zu umgehen und wo Mehrfachvererbung auf der Implementierungsebene in der Programmiersprache nicht unterstützt wird, können Interfaces als Ersatz eingesetzt werden, sofern die Sprache solche anbietet.

Der Ersatz kann eine Mehrfachvererbung der Implementierung natürlich nicht vollständig ersetzen. Er ist ein Kompromiss zwischen der Vermeidung von Nachteilen der Mehrfachvererbung und der Nutzung einiger ihrer Vorteile.

Als „Ersatz“ für eine Mehrfachvererbung der Implementierung gelten folgende beiden Wege:

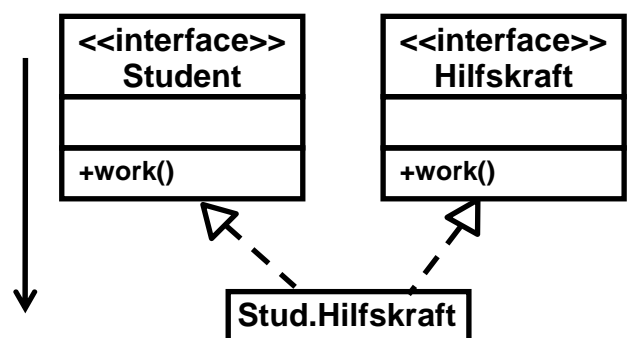
- Implementiert eine Klasse mehrere Schnittstellen, wird dies gelegentlich auch „Mehrfachvererbung“ genannt. Eigentlich ist es aber eine Mehrfachimplementierung. Wie bei einer Mehrfachvererbung schreibt die Mehrfachimplementierung ebenfalls mehrere Schnittstellen bzw. Rollen vor.
- Mehrfachvererbung zwischen Interfaces.

Mehrfachimplementierung

Mehrere Interfaces schreiben die gleiche Methode + Rückgabtyp vor (ohne Implementierung)

Die implementierende Klasse bekommt zweimal die Aufforderung, die Operation zu implementieren.

⇒ prinzipiell kein Problem



Klasse Stud.Hilfskraft besitzt unterschiedliche Typen

instanceOf: Oberklassen, eigene Klasse, implementierte Schnittstellen

Es gibt zwei grundsätzliche Fälle, in denen eine Mehrfachimplementierung in Java doch Probleme bereitet:

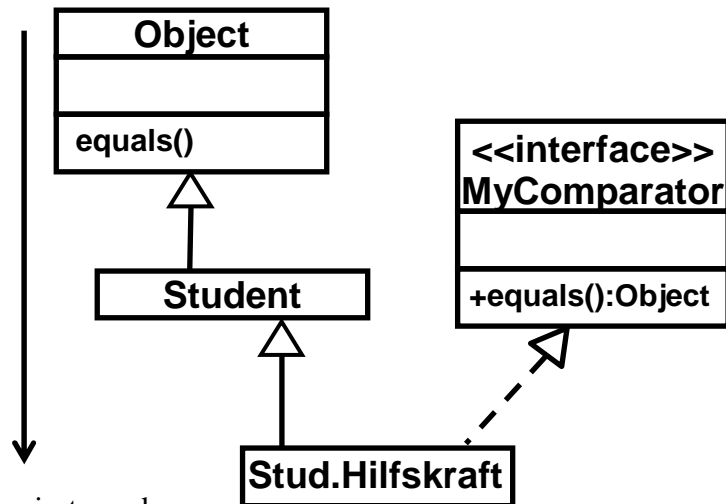
1. Eine Klasse implementiert zwei Interfaces, die die gleichen Methoden deklarieren, wobei die Methoden unterschiedliche Rückgabetypen haben.
2. Eine Klasse implementiert zwei Interfaces, die jeweils eine Konstante mit gleichem Namen aber unterschiedlichem Wert haben.

Eine Klasse kann von einer Klasse erben und gleichzeitig Interfaces implementieren.

Stud.Hilfskraft erbt `equals()`

⇒ Klasse `Stud.Hilfskraft` muss die Schnittstellenvorgabe `equals()` nicht selbst implementieren (kann aber)

Nutzen eines solchen Aufbaus:
`equals()` kommt über `MyComparator` in den Quellcode und es kann so eine spezifischere Dokumentation generiert werden.



Mehrfachvererbung zwischen Interfaces

Bei der Vererbung zwischen Interfaces gilt das gleiche Prinzip wie bei der Vererbung von Klassen (d.h. Vererbung von Implementierung), allerdings sind (z.B. in Java) mehrere Super-Interfaces möglich.

Damit haben wir in Java eine echte Mehrfachvererbung auf Interface-Ebene. Eine „scheinbare“ Mehrfachvererbung auf Klassenebene haben wir dadurch, dass eine Klasse mehrere Interfaces implementieren kann.

Abstrakte Klasse versus Interface

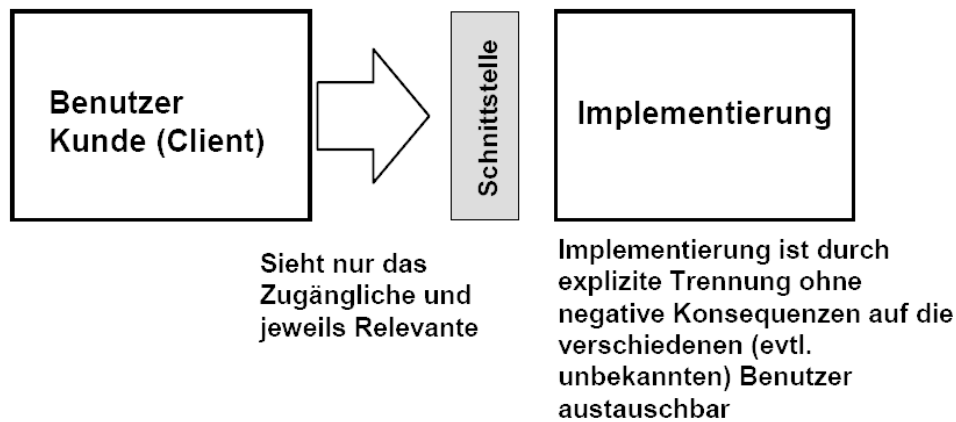
Ein Entwickler muss die Unterschiede und Gemeinsamkeiten zwischen abstrakter Klasse und Interface kennen, um in der Programmgestaltung das jeweils passende Konzept auswählen zu können.

Für die Programmiersprache Java können die beiden Konzepte wie in der folgenden Tabelle gegenübergestellt werden:

	Abstrakte Klasse	Interface
Einsatz: Vertragsdefinition/Vorschrift für Subklassen bzw. implementierende Klassen	Ja	Ja
Mögliche Anzahl / Vererbung	Einfachvererbung	Mehrfach (Mehrfachvererbung und mehrfache Implementierung)
Programmcode kann enthalten sein	Ja	Nein
Konsequenzen einer Änderung	Konkrete Methoden können ergänzt werden ohne, dass Subklassen angepasst werden müssen (Achtung: dadurch gleichzeitig Fragile Base Class Problem!)	Erfordert die Anpassung an vielen Stellen (in allen implementierenden Klassen)

3.6 Entwicklung mit Schnittstellen

Schnittstellen sind ein zentrales Konzept in der Entwicklung von Systemen.



Eine Schnittstelle ist die Menge der zugänglichen Elemente einer Einheit bzw. eines Moduls. Speziell in der Objektorientierung gilt: Eine Einheit ist typischerweise ein Objekt bzw. eine Klasse oder etwas Abstrakteres bzw. eine gekapselte Gruppe von zusammenarbeitenden Objekten oder Klassen.

Es gilt: Schnittstelle in der OO Denkweise \neq Java Interface!

Das Konzept der Schnittstelle ist unabhängig von der Programmiersprache.

Verschiedene Realisierungsmöglichkeiten von Schnittstellen:

In Java findet sich mit dem Programmkonstrukt **Interface** eine passende Abbildung von Schnittstellen auf Programmebene. Eine Schnittstelle auf Modellierungsebene muss aber (sogar in Java Programmen) nicht zwingend auf ein Java **Interface** abgebildet werden.

Abbildungsmöglichkeiten von Schnittstellen auf Programmebene sind beispielsweise (auch jenseits von Java):

- Deklaration öffentlicher Methoden (und Attribute)
In Java: **public**
- Abstrakte Klassen
In Java: **abstract**
- Separate Schnittstellendeklaration in der Sprache
In Java: **interface**
- Sprachunabhängige Schnittstellenbeschreibung
Beispiel: Interface Definition Language (IDL) der OMG
Bemerkungen:
 - Die OMG (Object Management Group) haben wir schon vorn zum Thema UML kennengelernt.
 - Sprachunabhängige Schnittstellenbeschreibungen werden beispielsweise in einem verteilten System benötigt oder aber in einem System, das sich aus Bausteinen zusammensetzt, die in verschiedenen Programmiersprachen geschrieben sind.

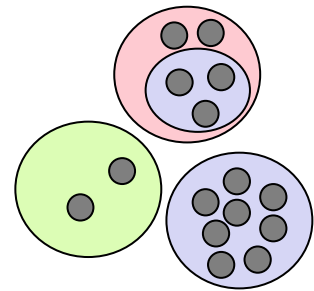
Wie eine Schnittstelle aussehen soll ist genauso wie die Abbildung in die Programmiersprache eine Entwurfsentscheidung. Es gibt also kein „Richtig“ oder „Falsch“, sondern nur ein „Besser“ und „Schlechter“.

Nutzen von Schnittstellen:

- Eine isolierte Entwicklung von Implementierungen wird möglich: Nach der Festlegung von Schnittstellen, kann die dazugehörige Implementierung isoliert vom Restsystem entwickelt werden.
- Anwendungen können nur auf Schnittstellen aufbauen: Eine konkrete Implementierung kann erst später entwickelt werden bzw. einfach durch eine verbesserte Implementierung ausgetauscht werden.
- Aufgabenaufteilung im Team: Anhand von Schnittstellen aufgeteilte Systembausteine können getrennt entwickelt und relativ einfach wieder zusammengefügt werden.

3.7 Paket (engl. Package)

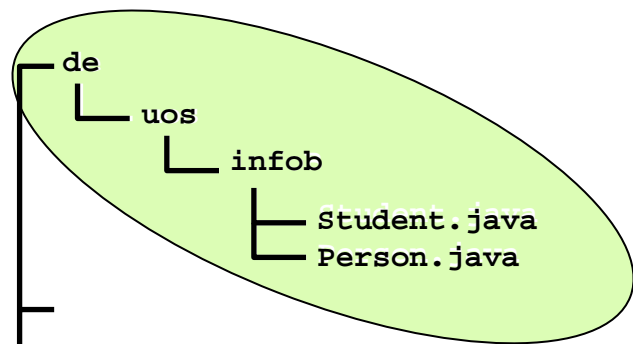
- Pakete sind - ähnliche wie Schnittstellen – zunächst einmal ein vollkommen sprachunabhängiges Konzept. Sie dienen der übergeordneten Strukturierung von Elementen zu größeren Einheiten und sind damit eine Form eines Moduls. Ein Paket sollte Elemente zusammenfassen, die zu einem gemeinsamen Aufgabenbereich gehören. Elemente können Klassen aber auch UML-Diagramme oder andere Dokumente sein.
- Alternative Bezeichnungen: Subsystem, Subject, Kategorie
- Auf objektorientierter Programmiersprachenebene (z.B. in Java) ist ein Paket primär eine logische Gruppierung von Klassen.
- Eine Klasse ist typischerweise Bestandteil genau eines Pakets.

**Nutzen und Einsatz von Paketen**

- Strukturierung durch Gruppierung und Hierarchisierung
- Vermeiden von Namenskonflikten (jedes Paket spannt einen eigenen Namensraum auf)
- Kontrolle von Zugriffsrechten und Sichtbarkeiten

Hierarchische Strukturierung

- Pakete können hierarchisch organisiert werden: Pakete enthalten wiederum Unterpakete usw.
In Java wird die Pakethierarchie durch eine Punktnotation ausgedrückt (paket1.paket11.paket111).
- Klassen, die zu einem Paket gehören, befinden sich im gleichen (virtuellen) Verzeichnis.
Bemerkung:
Es muss nicht zwingend auf ein reales Verzeichnis im Betriebssystem abgebildet werden.
- Paketname „=“ Verzeichnisname
- Typische Pakethierarchie:
umgedrehte Domänennamen (top-down)
Beispiel: Aus einer Domäne:
uos.de/infob wird so: de/uos/infob/
Eine solche Pakethierarchie ist nicht zwingend vorgeschrieben, aber bewährte Konvention. Sie hilft zur Vermeidung von Namenskonflikten z.B. beim Zusammenschalten von



zufällig gleich benannten Klassen.

Oft wird das Land (z.B. de) in der Hierarchie weggelassen und die Pakethierarchie beginnt direkt mit dem Hersteller. Paketnamen bestehen per Konvention nur aus Kleinbuchstaben.

- Reservierte Paketnamen von Sun/Oracle: **java**, **javax**, **sun** (z.B. **java.util**, **java.awt**)
Diese Paketnamen zeigen an, dass es ein Sun/Oracle-Paket und Teil der Distribution ist. Eigene Klassen sollten nicht in solche Pakete!
- Vollqualifizierter Name für Klassen, in Java z.B.
java.lang.String
de.uos.infob.Person

Zuordnung von Elementen zu Paketen in Java

- Definition der Paketzugehörigkeit innerhalb einer Klasse (erste Programmzeile):

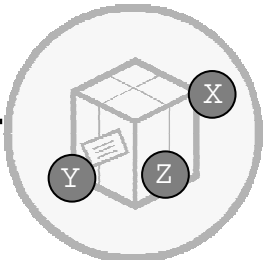
```
package de.uos.infob;           // Paketzugehörigkeit in der ersten Anweisung
                                // im Quellcode

public class Student { ...
```
- Ohne Paket-Angabe: Die Klasse ist automatisch Teil des unbenannten Pakets (unnamed package, Default-Paket).
Die Zuordnung zum Default-Paket sollte möglichst vermieden werden, da dadurch das Konzept der bewussten und gezielten Sichtbarkeitsbereichskontrolle aufgegeben wird. Das Default-Paket ist allenfalls für kleinere Programme sinnvoll.
- Erzeugen einer Verzeichnisstruktur, die der Paketstruktur entspricht (z.B. Verzeichnis **de**, Subverzeichnis **uos**, Sub-Subverzeichnis **infob** und darin liegend die Klasse **Student**).

Zugriff auf Paketinhalte in Java

- Zugriff auf spezifische Klassen eines anderen Pakets:
 - durch volle Qualifizierung des Klassennamens,
 - durch Import des Pakets (Information für den Compiler),
 - automatisch ohne expliziten Import speziell für das elementare Paket **java.lang**.
- Import mit sogenannten Wildcards, damit nicht alle Klassen eines Pakets einzeln aufgeführt werden müssen, z.B. **import java.util.*;**
Der Wildcard schließt in Java die Klassen in den Unterpaketen nicht ein.
- Generell gilt: Vorsicht mit solchen Wildcards!
Beispiel: Import eines Pakets, das bereits eine Klasse **Student** enthält.

```
import de.uos.infob.*;
public class Student {
    String name;
    public String getName() { ... }
```



Achtung: Namenskonflikte

Statischer Import in Java: Import von Klasseneigenschaften

- Durch statischen Import werden Klassenvariablen und Klassenmethoden importiert.
- Nach dem statischen Import muss nicht mehr der Klassenname explizit angegeben werden, wenn deren Klasseneigenschaften genutzt werden sollen.
- Achtung: Damit entsteht Verwechslungsgefahr mit Instanzvariablen und –methoden.
- Beispiel:

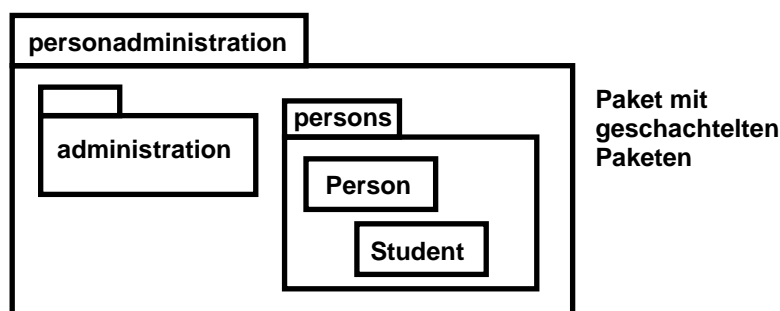
```
import static java.lang.System.out;
import static java.lang.Math.max;
. . .

public class StaticImport {
    public static void main(String[] args) {
        out.println(„Nutzung der statischen Variablen
                    out ohne Klassenname System“);
        out.printf(„Maximum:“, max(1,2));
    }
}
```

Auch beim statischen Import kann in Java mit Wildcards gearbeitet werden, wenn alle Klasseneigenschaften einer bestimmten Klasse importiert werden sollen (nicht zu Verwechseln mit dem Import aller Klassen bei der Verwendung von Wildcards beim nicht statischen Import).

Darstellung von Paketen in UML

- Mit Paketen können Systeme auf höherer Abstraktionsebene modelliert werden.
- Qualifizierter Name in textueller UML Schreibweise: Paket1::Paket11::Paket111::Klasse (z.B. personadministration::persons::Person)
- Beziehungen zwischen verschiedenen Paketen (z.B. aufgrund einer Nutzung) können durch eine Abhängigkeitsbeziehung (ggf. mit Stereotyp) dargestellt werden. Abhängigkeitsbeziehungen werden uns im Abschnitt über Beziehungen (unten) nochmal begegnen.
- UML Notation für ein Paket mit eingeschachtelten Paketen:



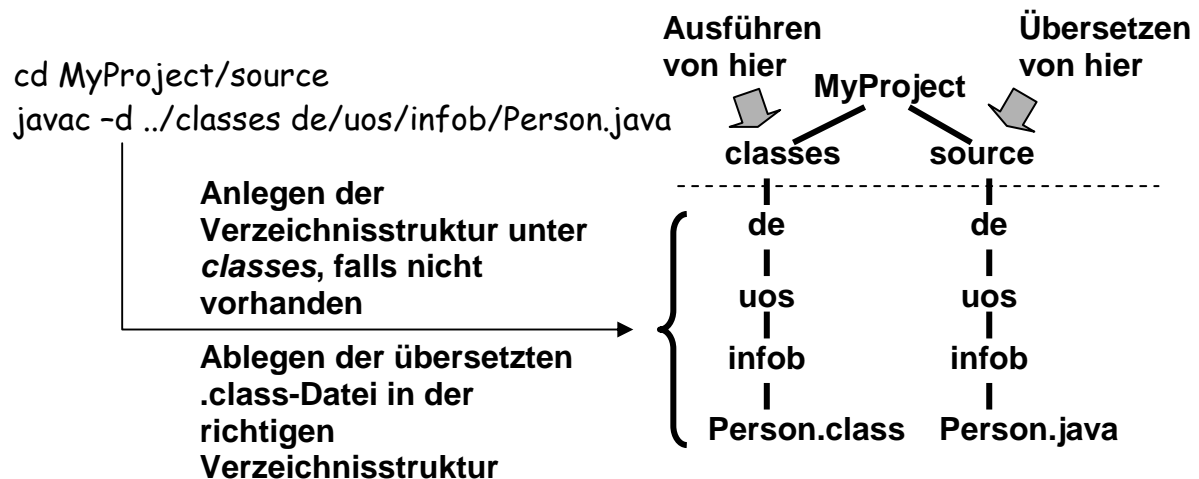
Inbetriebnahme der Software (engl. Deployment)

Irgendwann ist der Zeitpunkt gekommen, die selbst entwickelte Software auf die Menschheit zu deren Nutzen loszulassen. Das bedeutet, dass die Software geeignet aufbereitet und ausgeliefert werden muss.

Software, die ausgeliefert wird, wird typischerweise in Form von Paketen zur Verfügung gestellt (Klassenbibliotheken für bestimmte Anwendungsbereiche).

Quellen und übersetzte Dateien werden in getrennten Verzeichnissen abgelegt, so dass die ausführbaren Dateien ohne Quelldateien zur Auslieferung vorbereitet werden können.

Für die Inbetriebnahme geeignete Übersetzung in Java:



Ausführen des so strukturierten Programms in Java:

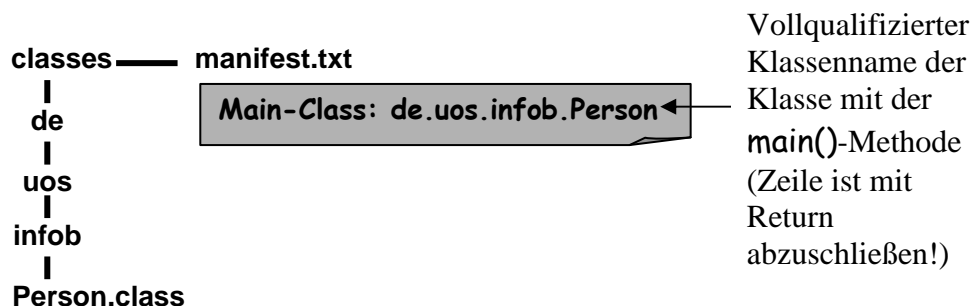
```
cd MyProject/classes
java de.uos.infob.Person
```

Schritte zum Aufbau einer auslieferbaren JAR-Datei in Java:

Eine JAR-Datei wird mit Hilfe des *jar*-Werkzeugs gebaut.

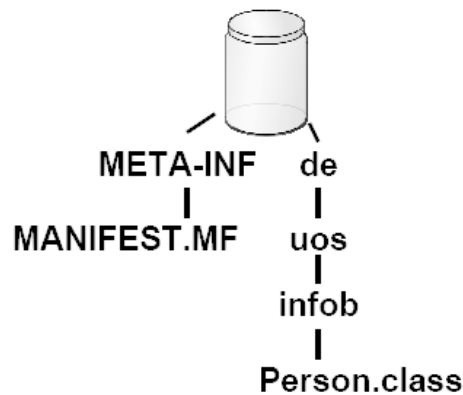
Das *jar*-Werkzeug basiert auf dem Werkzeug *pkzip*, welches Dateien in einer Datei komprimiert zusammenpackt.

Schritt 1: Die Dateien werden in die richtige Verzeichnisstruktur gebracht und die Entwicklerin schreibt eine Manifest-Datei.



Schritt 2: Es wird mit dem jar-Werkzeug eine JAR-Datei (im richtigen Verzeichnis) erzeugt
`cd MyProject/classes`
`jar -cvfm manifest.txt pers.jar de`

Ergebnis:



Schritt 3: JAR-Datei ausführen:

`java -jar pers.jar`

Wenn (eingekaufte) Pakete bzw. JAR-Dateien nicht in dem Ordner liegen (sollen), in dem die eigenen Programme liegen, muss der Pfad (PATH) und Klassenpfad (CLASSPATH) gesetzt werden, damit der Compiler die Klassen in dem Paket finden kann. PATH und CLASSPATH sowie andere derartige Informationen nennt man auch *Environment* bzw. *Umgebungsvariablen*.

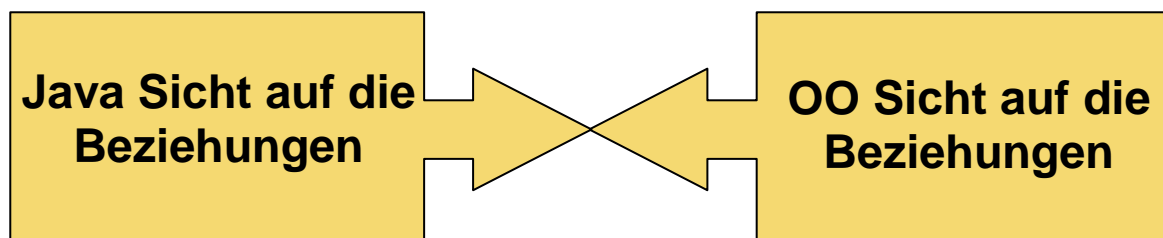
3.8 Beziehungen und ihre Modellierung

Wir haben vorn bereits Formen von Beziehungen kennengelernt, die wir hier zusammenfassen, erweitern und aus der Modellierungsperspektive betrachten.

Grundsätzliche Arten von Beziehungen:

- Instanziierungsbeziehung: Beziehung zwischen Klassen und ihren Instanzen.
- Beziehungen zwischen Klassen bzw. Beziehungen zwischen Instanzen.

Beziehungen aus Modellierungssicht und Abbildung der Beziehungen in die Programmwelt einer bestimmten Programmiersprache:



Es ist zu unterscheiden zwischen Beziehungen in der objektorientierten Modellierung (Modellierungsebene) und Beziehungen auf Programmiersprachenebene. Die modellierten Beziehungen können unter Umständen nicht 1:1 in eine Programmiersprache übertragen werden.

Umgekehrt sollten aber Beziehungen zunächst ohne Fixierung auf eine bestimmte Programmiersprache modelliert werden.

Beziehungen zwischen Klassen bzw. zwischen Instanzen aus Sicht der Programmiersprache Java

Beziehungen können aus Sicht von Java realisiert werden durch:

- durch gemeinsamen “Speicher”,
- durch Methodenaufruf,
- durch Vererbung.

Für Beziehungen zwischen Klassen bzw. Beziehungen zwischen Instanzen kann aus Modellierungssicht genauer unterschieden werden:

- Assoziation (engl. Association),
- Aggregation,
- Komposition (engl. Composition),
- Benutzt-Beziehung (engl. Using), auch als gerichtete Assoziation bezeichnet,
- Vererbung.

Daneben gibt es in UML noch eine ganz allgemeine Abhängigkeitsbeziehung, die sonstige Abhängigkeiten zwischen Modellelementen ausdrücken kann.

Bei den Paketen hatten wir diese bereits erwähnt.

Abbildung Modell – Programm:

Für die Vererbung ist eine Abbildung vom Modell auf ein Programm in einer objektorientierten Sprache in der Regel relativ einfach. Beide Ebenen unterstützen in objektorientierten Programmiersprachen und objektorientierten Modellierungssprachen das Konzept der Vererbung direkt. Hier sind gegebenenfalls semantische und konzeptuelle Unterschiede (z.B. Einfach- oder Mehrfachvererbung oder Repräsentation von Diskriminatoren) zu überbrücken.

Für die übrigen Beziehungen ist eine Abbildung oft komplizierter. Die Modellierungsebene ist dabei in der Regel näher an der Realität und kann daher mehr und genauer darstellen. Eine Modellierungssprache sollte unterstützen, dass reale Anforderungen und Systemgegebenheiten so genau wie möglich repräsentiert werden können. Nicht modellierbares Wissen ist für die nachfolgenden Entwicklungsschritte anderenfalls unter Umständen unwiederbringlich verloren. Auf der Programmebene muss der Entwickler gegebenenfalls Kompromisse eingehen und Umwege zur Implementierung einplanen, um das Modell in ein ausführbares und gut einsetzbares System zu bringen. Die Kompromisse und Umwege werden z.B. durch die Einschränkungen der verwendeten Programmiersprache, der Plattform sowie durch Optimierungsbestrebungen notwendig.

Beziehungen zwischen Klassen und Beziehungen zwischen Objekten:

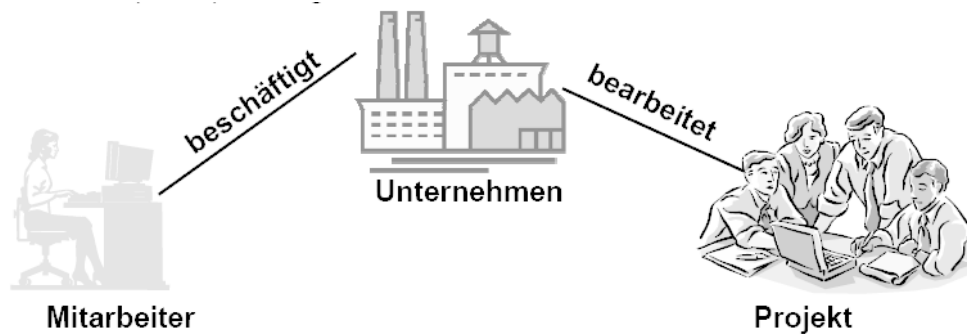
Genauso wie zwischen Klassen und Instanzen/Objekten muss auch bei Beziehungen zwischen der allgemeinen und für alle Instanzen/Objekte gültigen Beziehung und der Beziehung in konkreter Ausprägung unterschieden werden.

Auf der Klassenebene drückt eine Beziehung aus, dass für alle Instanzen der Klasse z.B. eine bestimmte Nachrichtenbeziehung besteht, über die jeweils Anforderungen ausgetauscht werden.

Die Bezeichnung *Objektbeziehungen* bzw. *Objektverbindungen* (engl. *Link*) steht dagegen speziell für Exemplare einer Beziehung (analog zu: “Objekte sind Exemplare einer Klasse”). In UML Klassendiagrammen modellieren wir auf Klassenebene die Beziehungen, die die Instanzen allgemein in unserem System haben.

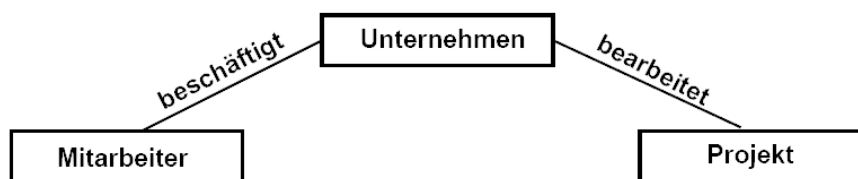
3.8.1 Assoziation (engl. Association)

- Eine Assoziation ist ein unspezifischer Beziehungstyp.
- Wir wissen nur, dass es eine Art von Beziehung und Abhängigkeit gibt, aber (noch) nicht genau welche.



UML Darstellung einer Assoziation

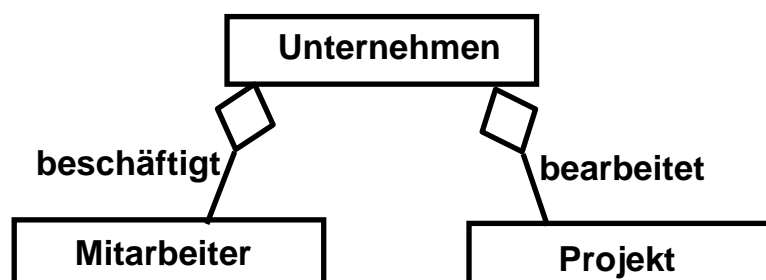
- Einfache Darstellung in den meisten Modellierungsnotationen (einfache Linie).
- Keine Aussage über die genaue Realisierung.
- Typisch für ein frühes Stadium im Softwareentwicklungsprozess: Eine Assoziation kann später zu einer der anderen Beziehungsarten verfeinert werden.



3.8.2 Aggregation

- Ebenfalls verwendete Namen: Shared Aggregate, Whole/Part-Beziehung, Ganzes/Teil-Beziehung
- Eine Aggregation ist eine spezielle Assoziation.
- Sie drückt aus, dass ein Objekt Teil eines anderen ist.
- Ein Komposit (engl. Composit) bzw. Ganzes (engl. Whole) bzw. Aggregat besteht aus (evtl. mehreren) Komponenten (engl. Components) bzw. Teilen (engl. Parts).

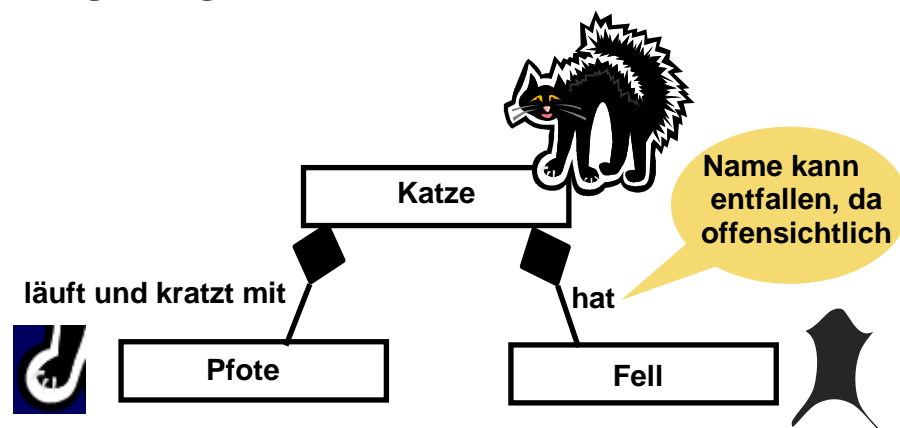
UML Darstellung



3.8.3 Komposition (engl. Composition)

- Alternativer Name: Composite Aggregate
- Eine Komposition ist eine spezielle (stärkere) Form einer Aggregation.
- “Strong Ownership”-Eigenschaft: Die Teile können nicht das Ganze überleben (Übereinstimmung der Lebensdauern; existenzabhängige Teile).
- Das aggregierte Objekt wird nicht mit anderen Objekten geteilt: Zu einem Zeitpunkt kann ein Teil nur einem Aggregatobjekt zugeordnet sein (und von diesem existenzabhängig sein).
- Das Ganze (Whole) erzeugt (meist bei dessen Erzeugung) auch seine Teile.
- Wird das Komposit gelöscht, werden automatisch auch seine Teile gelöscht.
- Achtung: Manchmal (z.B. in UML) gilt auch:
Teile dürfen (z.B. vor einer Löschung) einem anderen Aggregat-Objekt zugeordnet werden / verantwortlich übergeben werden.

UML Darstellung (mit gefüllter Raute):



Umsetzung der Komposition

Eine modellierte Beziehung muss in der Regel irgendwann in einer Programmiersprache umgesetzt werden. Die Umsetzung ist immer eine Entwurfsentscheidung. Das bedeutet insbesondere, dass es auch verschiedene gültige Alternativen in der Umsetzung gibt und diese in einem bestimmten Kontext „besser“ oder „schlechter“ sein können.

Für die Umsetzung einer Aggregation und einer Komposition in einer objektorientierten Sprache gilt in der Regel einheitlich:

- Das Komposit besitzt eine Referenz (in der Regeln in Form eines Attributs) auf die Komponentenobjekte.

Eine Aggregation wird ähnlich wie eine Komposition umgesetzt. Allerdings entfällt die zusätzliche Absicherung zur Einhaltung der Eigenschaft der Existenzabhängigkeit und der Unteilbarkeit. Für eine Kompositionsbeziehung gilt demnach zusätzlich:

- Das Komposit initiiert die Instanziierung und Zerstörung der referenzierten Komponentenobjekte (Kontrolle der Lebensdauern der Komponenten durch das Komposit).
- Die Komponenten dürfen nicht in Attributen anderer Objekte (jenseits des Komposits) referenziert werden.

Beispiel für eine Umsetzung einer Komposition in Java:

```
public class Katze {  
    private Pfote meineRVorderPfote;  
    private Fell meinFell;  
  
    public Katze() {  
        meineRVorderPfote = new Pfote();  
        meinFell = new Fell();  
    }  
  
    protected finalize() {  
        // ohne Garbage Collector  
        // müssen wir die Komponenten  
        // im Destruktor zerstören  
    }  
}
```

Das Umsetzungsbeispiel kann nur die grundsätzliche Idee zeigen, da die jeweilige Umsetzung kontextabhängig ist. Die Kontrolle der Referenzierung durch andere Objekte ist beispielsweise im Beispiel nicht sichtbar.

Typisch ist, dass die Komponenten als Attribute auftauchen und die zugehörigen Instanzen im Konstruktor erzeugt werden. Denkbar wäre auch eine spätere Erzeugung.

Für Kompositionen ist zudem zu beachten, dass die Existenzabhängigkeit in irgendeiner Form im Programm verankert und möglichst abgesichert ist. In C++ ist das übliche Vorgehen, die Komponenten im Destruktor zu löschen. In Java haben wir einen Garbage Collector und können (bzw. müssen) daher nicht mehr selbst für die Instanzvernichtung sorgen. Ein schwacher Ersatz wäre die explizite **null**-Setzung der entsprechenden Attribute. Aufgrund der schon vorn gesehenen Finalizer-Problematik in Java ist bei der Zerstörung des Komposits an dieser Stelle nicht sichergestellt, dass die Komponenten ebenfalls (sofort) zerstört werden. Die Java-Entwicklerin muss sich also gegebenenfalls einen eigenen Mechanismus einfallen lassen, um abzusichern, dass mit Zerstörung des Komposits auch alle Komponenten abgeräumt werden können. Es kann z.B. gelegentlich auch eine eigene Referenzverwaltung mit eigener **destroy**-Methode sinnvoll sein.

Bietet eine Programmiersprache nicht ausreichend Unterstützung für die Umsetzung von Modellierungskonzepten, werden diese (leider) oft auch in der Implementierung einfach ignoriert. Im Fall einer Komposition in Java bedeutet dies, dass eine Existenzabhängigkeit beispielsweise oft nicht im Programm abgesichert wird.

Bemerkung:

Es ist auch denkbar, aber eher selten, dass die Referenzen auf die Komponenten nicht als Attribut, sondern als lokale Variablen abgelegt sind. Eine modellierte Kompositions- bzw. Aggregationsbeziehung ist normalerweise eine dauerhaftere Beziehung. Demgegenüber ist eine lokale Variable kurzlebig.

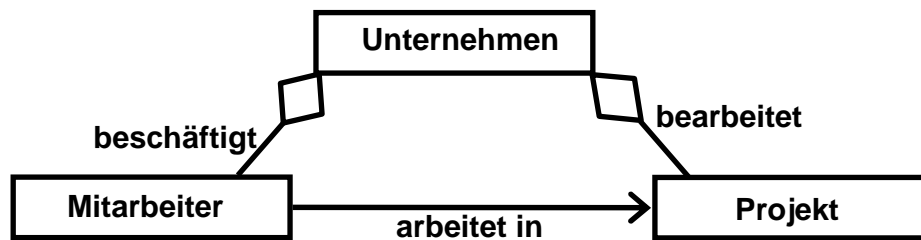
3.8.4 Benutzt-Beziehung (engl. Using)

Alternativer Name: Gerichtete Assoziation

Eine Benutzt-Beziehung ist ähnlich zur Aggregationsbeziehung, hat aber folgende Unterschiede:

- Keine eigene Instanziierung der benutzten Instanz.
- Keine Berechtigung, die benutzte Instanz zu zerstören und keine sonstige Kontrolle über das benutzte Teil.



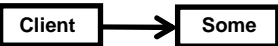
Beispiel in UML-Darstellung: Using-Beziehung zwischen Mitarbeiter und Projekt



Im Beispiel muss die Mitarbeiter-Instanz vom Unternehmen eine Referenz auf das Projekt bekommen. Anschließend darf der Mitarbeiter das Projekt benutzen. Die Kontrolle über das Projekt bleibt aber beim Unternehmen.

3.8.5 Abgrenzung: Komposition, Aggregation, Using

In der folgenden Tabelle werden die drei Beziehungsarten Komposition, Aggregation und Using anhand einiger Kriterien gegenübergestellt:

Kriterium	Komposition	Aggregation	Using
UML			
Besitzt / Hat (mit Hat-Beziehung \subseteq Besitzt- Beziehung)	Whole besitzt Part	Whole hat Part	Client hat nur Zugriff auf Some
Ort der Erzeugung der Part-Instanz bzw. Some-Instanz	Im Whole	Im Whole, evtl. auch außerhalb und Übergabe an Whole	Nur außerhalb von Client
Zeitpunkt der Erzeugung der Part- Instanz bzw. Some- Instanz	Meist im Konstruktor von Whole (selten später)	Oft zusammen mit Whole (evtl. später), evtl. auch Erzeugung außerhalb von Whole	Offen: vor der Nutzung durch einen Client
Eine Part-Instanz bzw. Some-Instanz kann <u>zu einem</u> <u>Zeitpunkt</u> mehrfach in der gleichen Beziehungsart sein	Nein	Zwei Ansichten: 1. Nein 2. Ja (meist in Verbindung mit der Bezeichnung „Shared Aggregate“)	Ja
Eine Part-Instanz bzw. Some-Instanz kann <u>zu</u> <u>verschiedenen</u> <u>Zeitpunkten</u> in der gleichen Beziehungsart sein	Zwei Ansichten: 1. Nein 2. Ja (z.B. erlaubt UML eine verantwortliche Weitergabe der Instanz z.B. vor der Zerstörung der Whole-Instanz)	Ja	Ja
Die Part-Instanz bzw. Some-Instanz hängt mit seiner Existenz von Whole bzw. Client ab	Ja	Nicht zwingend	Nie
Kontrolle über die Part-Instanz bzw. Some-Instanz von Whole bzw. Client aus	Stärkste Kontrolle	Mittlere bis starke Kontrolle	Keine Kontrolle

Kontrollstärke grafisch



3.8.6 Verfeinerungen der Beziehungen (mit UML)

Die oben kennengelernte Assoziation ist die allgemeinste Form einer Beziehung zwischen Klassen. Neben der Verfeinerung einer Assoziationsbeziehung in spezifischere Beziehungen (z.B. Using oder Aggregation) sind weitere Verfeinerungen möglich. Diese Verfeinerungen erlauben es, noch mehr Wissen und Bedeutung (Semantik) aus der Realität im Modell festzuhalten. Ein gutes Modell sollte alle wichtigen Informationen abbilden – nicht mehr aber auch nicht weniger. Eine gute Modellierungssprache sollte demnach entsprechende Beschreibungsmittel anbieten – nicht weniger aber, aus Gründen der Bedienbarkeit, möglichst auch nicht viel mehr.

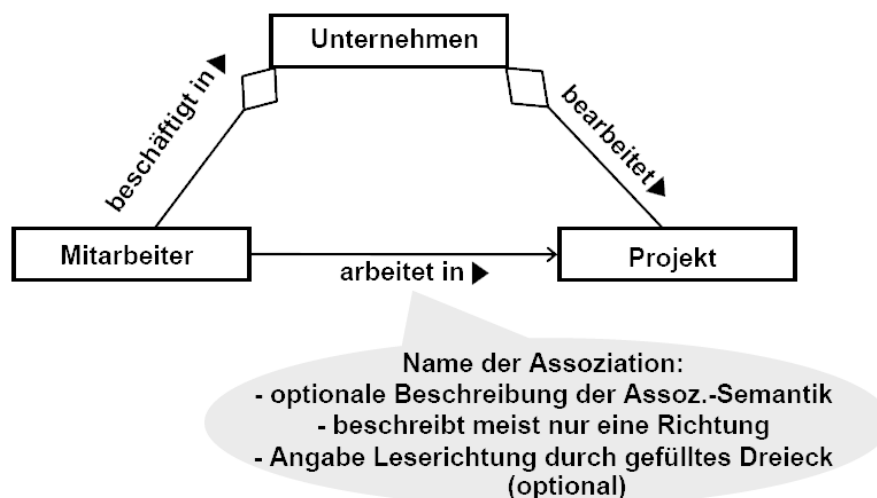
Namen und Leserichtung

Beziehungen sollten (wie in den vorhergehenden Beispielen) Namen tragen. Die Namen sind optionale Beschreibungen der Assoziationssemantik. Sie drücken also aus, was die Beziehung tatsächlich bedeutet.

Die Bedeutung eines Beziehungsnamens hat meistens eine Richtung. Die Beschriftung „arbeitet in“ an der Using-Beziehung zwischen *Mitarbeiter* und *Projekt* kann beispielsweise nur sinnvoll vom *Mitarbeiter* in Richtung *Projekt* gelesen werden.

In UML ist eine Angabe der Leserichtung optional möglich.

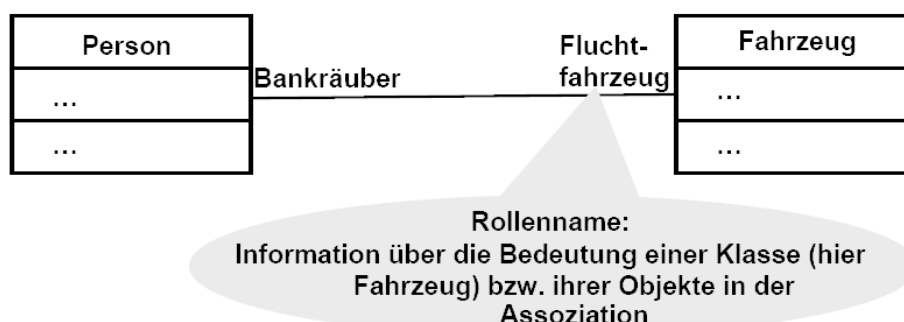
Beispiel für Beziehungsnamen und Leserichtung in UML:



Rollen

Eine Beziehung kann mit Rollen beschriftet werden. Eine Rolle drückt aus, welche Aufgabe bzw. Bedeutung eine bestimmte Instanz innerhalb einer Beziehung hat.

Beispiel für Rollenangaben in UML:



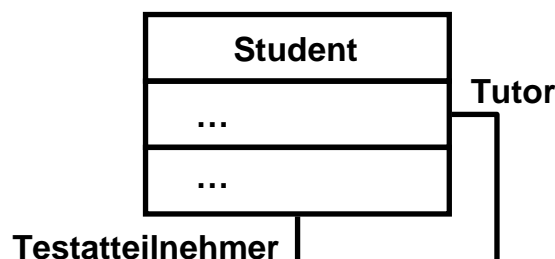
Bemerkung:

Den Begriff der „Rolle“ hatten wir in Zusammenhang mit Schnittstellen bereits gehört. Das deutet bereits an, wie eine gute Umsetzung von Rollen aussehen kann. Spielt eine Instanz eine bestimmte Rolle innerhalb einer Beziehung, bedeutet dies, dass nur bestimmte Eigenschaften der Instanz relevant sind. Diese bestimmte (abgeschlossene) Menge an Eigenschaften kann sehr gut in eine eigene Schnittstelle gefasst und z.B. als **Interface** in Java umgesetzt werden.

Reflexive Assoziation

- Alternative Bezeichnungen: Zirkuläre oder rekursive Assoziation
- Eine reflexive Assoziation ist eine Beziehung zwischen Objekten, die zur gleichen Klasse gehören (der Typ verweist auf sich selbst).
- Rollenangaben sind hier besonders wichtig.

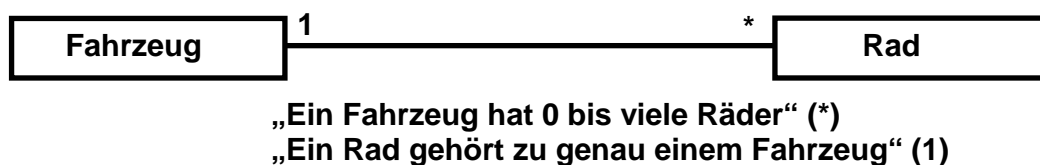
In UML-Darstellung:



Multiplizität (Kardinalität, Anzahlangabe, Vielfachheit)

- Eine Multiplizität verfeinert Assoziationen (allgemeine oder spezifischere).
- Die Multiplizität spezifiziert, wie viele Objekte ein bestimmtes Objekt kennen kann (bzw. mit wievielen Objekten es in Verbindung steht).

Beispiel in UML:



Beispiele für andere, mögliche Multiplizitäten in UML-Darstellung:



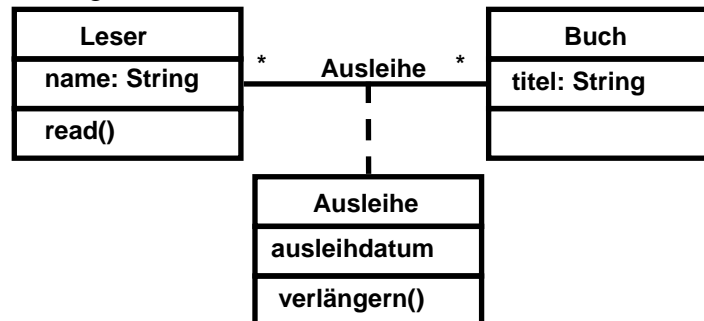
Bemerkung:

Statt der *-Schreibweise wird auch manchmal „n“ oder „m“ verwendet, um auszudrücken, dass keine (Null) bis beliebig viele Objekte möglich sind.

Assoziationsklasse (engl. Association Class)

- Alternative Bezeichnung: Attributierte Assoziation
- Assoziationsklassen sind Assoziationen, die zusätzlich Eigenschaften einer Klasse besitzen sollen.

Beispiel in UML-Darstellung: Assoziationsklasse *Ausleihe*



Wichtig: Die Assoziationsklasse heißt gleich wie die Beziehung, der sie zugeordnet ist.

Assoziationsklassen können dann sinnvoll sein, wenn die Beziehung aufgewertet werden soll, weil sie mehr als nur eine einfache Beziehung ist.

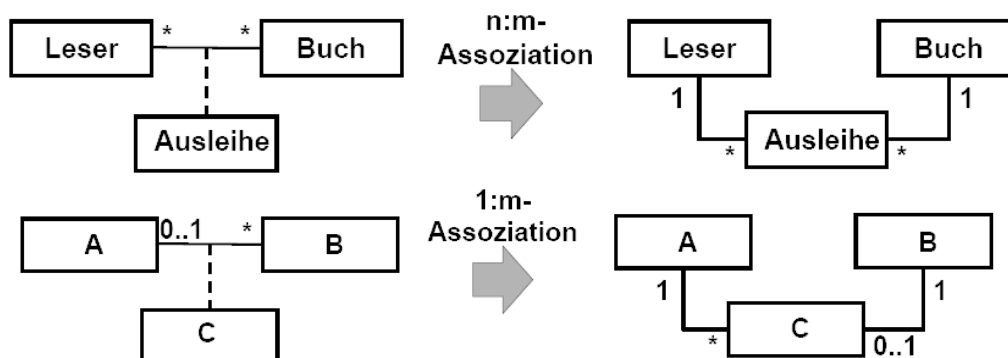
Assoziationsklassen stammen aus dem Gebiet der Datenbankmodellierung.

In der objektorientierten Modellierung sollten Assoziationsklassen in Klassen und Beziehungen aufgelöst (transformiert) werden.

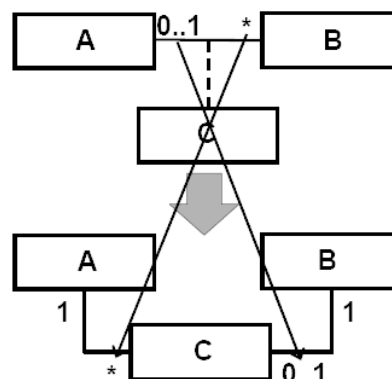
Transformation von Assoziationsklassen:

Eine Assoziationsklasse kann immer in zwei Assoziationen und eine eigenständige Klasse transformiert werden.

Beispiel: Jeweils ein Ausgangsmodell (links) wird in ein Modell ohne Assoziationsklasse (rechts) transformiert.



Allgemeine Transformationsregel:



Navigierbarkeit

Die Navigierbarkeit ist eine Detaillierung bzw. Erweiterung dessen, was wir bereits in Ansätzen in der Benutzt-Beziehung (Using) kennengelernt haben.

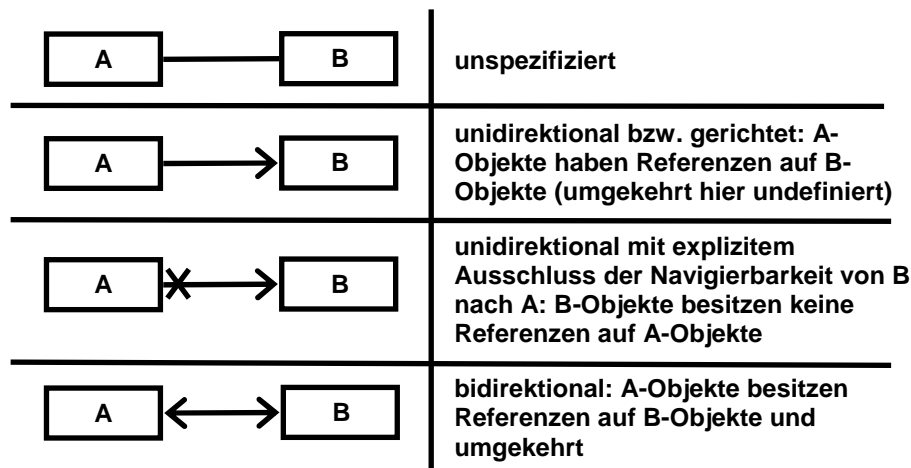
Beispiel für die UML-Darstellung der Navigierbarkeit an einer Assoziation:



Im Beispiel gilt:

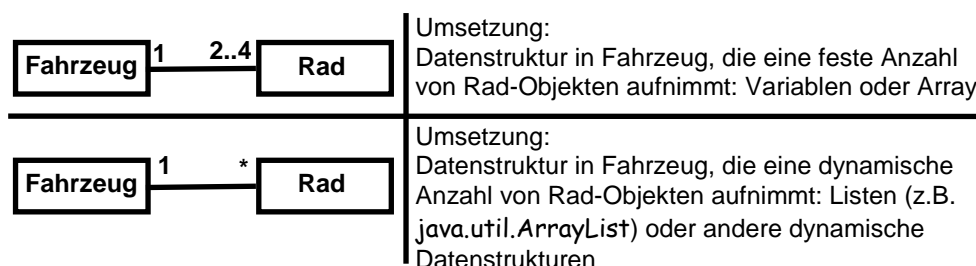
“Die Assoziation ist von *Mitarbeiter* nach *Projekt* navigierbar”. Das bedeutet: Objekte von *Mitarbeiter* können auf Objekte von *Projekt* zugreifen (aber nicht unbedingt umgekehrt).

Neben der Navigierbarkeit, die in einer uns schon bekannten Benutzt-Beziehung resultiert, gibt es noch weitere Arten von Navigierbarkeit:



Umsetzung von Beziehungen (unter Berücksichtigung der Navigierbarkeit)

- Modellerte Beziehungen werden in der Regel in objektorientierten Programmiersprachen durch Referenzen und Nachrichten an die in den Referenzen gespeicherten Objekte umgesetzt.
- Konsistenzbedingungen müssen in den meisten Fällen für die jeweils modellierte Beziehungsart vom Programmierer selbst kontrolliert werden (z.B. 1:* Kardinalität). Das bedeutet, dass der Entwickler im Programm für die Einhaltung der Modellierungsvorgabe sorgen muss. Wo möglich sollten Sicherungen eingebaut werden, die einen Verstoß gegen die Modellierungsvorgabe im Programm (zur Compile-Zeit oder notfalls erst zur Laufzeit) erkennen.
- Für Beziehungen zu mehreren Objekten (z.B. 1:*-Beziehungen) genügt eine Referenzvariable nicht. Es werden dynamische Datenstrukturen benötigt.



Abgeleitete Beziehung (engl. Derived Association)

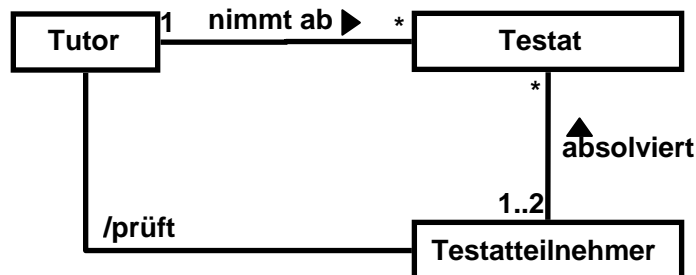
Eine abgeleitete Beziehung ist eine Beziehung, für die die gleiche Abhängigkeit bereits durch andere Assoziationen beschrieben ist.

Eine Implementierung setzt eine abgeleitete Beziehung - analog zu einem abgeleiteten Attribut - in der Regel nicht direkt um.

Beispiel und Darstellung in UML:

Im Beispiel ist die Beziehung „prüft“ bereits durch die Beziehungen „nimmt ab“ und „absolviert“ beschrieben. Die Beziehung „prüft“ ist daher als abgeleitete Beziehung modelliert.

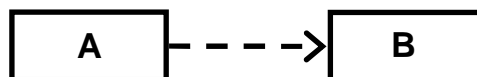
Die UML-Notation für abgeleitete Beziehungen entspricht derjenigen für abgeleitete Attribute.



3.8.7 Abhängigkeit (engl. Dependency)

Neben den bisher betrachteten Beziehungen, ist in UML eine weitere, sehr allgemeine Abhängigkeitsbeziehung definiert. Diese Beziehungsart bezieht sich zunächst auf Modellelemente, während sich eine Assoziation auf Objekte/Klassen bezieht.

Darstellung der Abhängigkeitsbeziehung in UML:



Im Beispiel ist A abhängig von B (erst durch B wird A vollständig).

Die Art der Abhängigkeit kann durch einen *Stereotypen* (engl. Stereotype) näher spezifiziert werden.

Eine Abhängigkeit ist zunächst nicht näher festgelegt, sondern individuell in verschiedenen Modellierungskontexten unterschiedlich verwendbar.

Beispiel für den Einsatz von Abhängigkeitsbeziehungen: Modellierung von Paketabhängigkeiten.

Beispiel für eine mögliche Ergänzung durch einen Stereotypen `<<permit>>`:

Eine Abhängigkeitsbeziehung mit diesem Stereotyp als Zusatz hat folgende Bedeutung.

A hat die Erlaubnis, (spezielle) private Eigenschaften von B zu verwenden.

UML schlägt einige derartige Stereotypen bereits vor.

Weitere können vom Modellierer – wo notwendig – selbst gebildet werden.

Sie müssen dann semantisch genau definiert werden.

3.8.8 Dynamische Modellierung der Beziehungen: UML Sequenzdiagramme

Die Modellierungssprache UML umfasst eine Reihe von Diagrammen, um verschiedene Aspekte des Systems und verschiedene Sichten auf das System darstellen zu können.

Wir haben bisher in erster Linie zwei UML-Diagrammartenn kennengelernt:

1. Klassendiagramme (engl. Class Diagram): Repräsentation der statischen Struktur bzw. der statischen Beziehungen zwischen den Klassen.
2. Paketdiagramme (engl. Package Diagram): Darstellung der statischen Paketstruktur (höhere Abstraktionsebene).

Neben der statischen Darstellung der Klassen und Pakete mit ihren Beziehungen in einem UML Klassendiagramm, ist es oft notwendig, auch das dynamische Verhalten des Programms in wichtigen Auszügen zu modellieren.

Eine häufig verwendete Diagrammart dafür ist das Sequenzdiagramm (engl. Sequence Diagram). Es gehört zur Klasse der Interaktionsdiagramme (Interaction Diagrams). Ein weiteres UML Interaktionsdiagramm ist das Kommunikationsdiagramm, das wir hier aber nicht betrachten.

Vorläufer der Sequenzdiagramme sind die sogenannten Message Sequence Charts (MSCs). Sie stammen aus der Telekommunikationsindustrie. In die objektorientierte Modellierung sind die Sequenzdiagramme von Rumbaugh und Booch eingeführt worden. Deren Sequenzdiagramme sind zum UML Sequenzdiagramm verschmolzen worden.

UML Sequenzdiagramme stellen exemplarisch den zeitlichen Ablauf gesendeter Nachrichten zwischen den beteiligten Objekten dar (Auszug des Systemkontrollflusses, Aufrufsequenz).

Ein Sequenzdiagramm hat zwei Dimensionen:

1. von links nach rechts sind die Kommunikationspartner aufgereiht,
2. von oben nach unten verläuft eine (imaginäre) Zeitachse.

Elemente des zweidimensionalen Sequenzdiagramms

Ein Sequenzdiagramm besteht im Wesentlichen aus folgenden Bestandteilen:

- Kommunikationspartner (engl. Communication Partners): Objekte des Systems,
- Lebenslinien (engl. Lifelines): Repräsentation der Lebenszeit eines Objekts,
- Nachrichten (engl. Messages),
- Sprachmittel zur Ablaufkontrolle.

Sequenzdiagramme können, wie Klassendiagramme und alle übrigen UML Diagramme, durch Kommentarsymbole ergänzt werden.

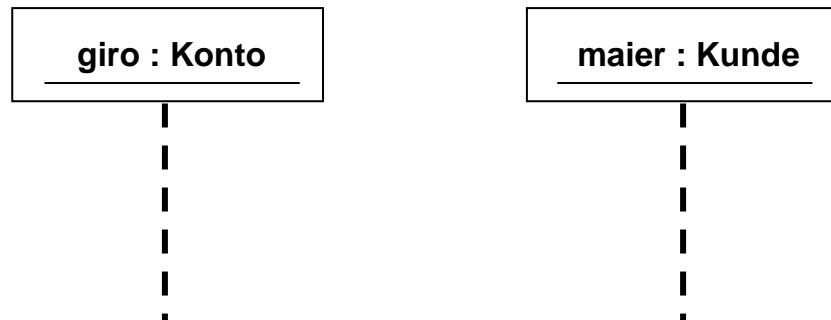
Kommunikationspartner

Als Kommunikationspartner kann jedes Objekt des Systems auftreten. Teilweise werden „Rollen“ statt Objekte eingetragen. Diese werden dann nicht unterstrichen dargestellt. In der Veranstaltung arbeiten wir im Sequenzdiagramm immer mit Objekten.



Lebenslinien

Eine Lebenslinie wird als gestrichelte Linie dargestellt und repräsentiert die Lebenszeit eines Kommunikationspartners.



Nachrichten

- Kommunikationspartner versenden Nachrichten untereinander.
- Nachrichten werden als Pfeile zwischen den Lebenslinien der Kommunikationspartner dargestellt.

Nachrichtenarten:

Es werden zwei Arten von Nachrichten unterschieden:

- Synchroner Nachrichten
Der Aufrufer wartet bis der aufgerufene Kommunikationspartner die Verarbeitung beendet hat und eine Rückantwort geliefert hat.
- Asynchrone Nachrichten
Der Aufrufer der Nachricht wartet mit der Verarbeitung nicht auf den aufgerufenen Kommunikationspartner.
Asynchrone Nachrichten werden verwendet, um mehrere, nebenläufige Prozesse zu modellieren.

Grafische Darstellung der Nachrichtenarten:

Asynchrone Nachricht

Synchrone Nachricht:

Antwortnachricht:
(zwei Alternativen)

Beschriftung der Nachrichten (vereinfacht; Auszug der wesentlichen Teile in EBNF):

Nachricht ::= “[Bedingung]” NameDerNachricht [“(“ [Argument { “,” Argument }] “)”]
Argument ::= ([NameDesParameters “=”] Argumentwert) | “-“

Beschriftung der Rückantwort (vereinfacht; Auszug der wesentlichen Teile in EBNF):

Antwortnachricht ::= [Attribut “=”] NameDerNachricht [“(“)] [“:“ Rückgabewert]

Für Rückantworten findet man auch gelegentlich die Schreibweise ohne Methodennamen in der Form *return(Rückgabewerte)*.

Eine Nachricht und insbesondere auch eine Rückantwort muss nicht zwingend beschriftet werden.

Da UML unabhängig zu einer Programmiersprache ist, sind manche Beschriftungsmöglichkeiten in einer bestimmten Sprache eventuell nicht direkt abbildbar. Die oben angegebenen Regeln zur Beschriftung von Nachrichten vereinfachen die Möglichkeiten, die die UML Spezifikation vorgibt.

Für Rückantworten könnten beispielsweise zusätzlich die Werte, die über Rückgabemethodenparameter zurückgegeben werden, angegeben werden. In Java gibt es allerdings keine direkte Rückgabe über Methodenparameter (im Gegensatz zu anderen Programmiersprachen). Allenfalls Seiteneffekte bei Referenzen als Parameter könnten als solche Rückgabewerte in den Parametern eingesetzt werden.

Beispiele für Nachrichtenbezeichnungen:

Im Klassendiagramm sei folgende Operation enthalten:

`findeName(telNr:String, name:String): boolean`

Dann können beispielsweise folgende Nachrichtenbezeichnungen in UML verwendet werden:

Aufruf: `findeName(05419692534,-)`

Aufruf: `findeName(telNr=nummer)` // nummer ist eine mit einem Wert belegte Variable

Bedingte Operation: `[nummer < > 0] findeName()`

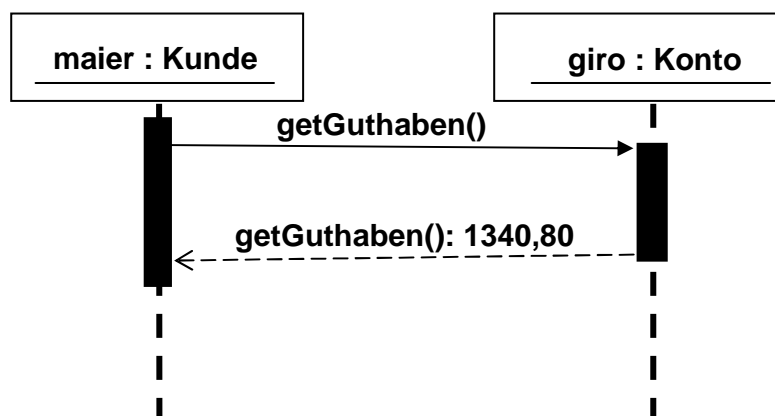
Rückantwort: `findeName():true`

Konsistenz: Alle im Sequenzdiagramm verwendeten Nachrichten müssen im Klassendiagramm mit entsprechender Signatur definiert sein. (Wichtig!)

Aktionssequenz (Aktivierung)

- Ein Kommunikationspartner kann neben dem Senden und Empfangen von Nachrichten verschiedene Tätigkeiten ausführen.
- Aktionssequenzen, in denen ein Kommunikationspartner etwas durchführt, werden als senkrechte Balken auf der Lebenslinie, die auch geschachtelt auftreten können, dargestellt. Man nennt sie Aktivierungsbalken (engl. Activation Bar).
- Eine Aktionssequenz ist durch ein Start- und ein Endereignis eingeschlossen.
- Die Ausführung der Aktionssequenz benötigt Zeit.

Beispiel (Ausschnitt):



Ein Sequenzdiagramm enthält eine bis viele Aktionssequenzen.

Ein Sequenzdiagramm ist von links nach rechts und von oben nach unten strukturiert.

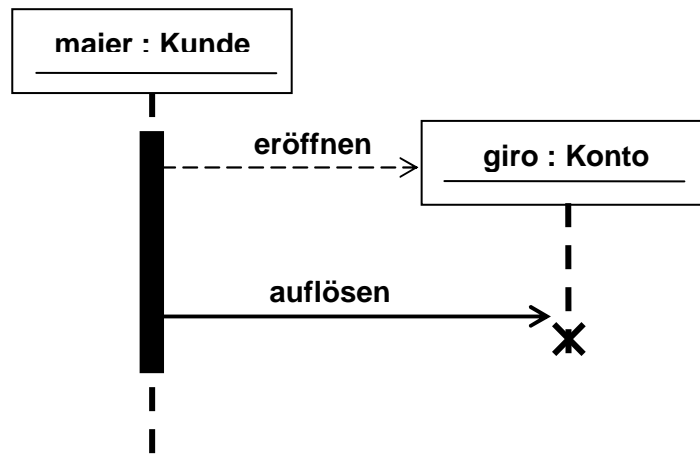
Das erste initiiierende Objekt sollte also ganz links oben stehen.

Nachrichtenpfeile können nicht nach oben zeigen (nur horizontal oder schräg nach unten).

Ausnahme: Nachrichten, die einen Schleifenrücksprung darstellen, werden durch rückwärtsgerichtete Pfeile (auf sich selbst) dargestellt.

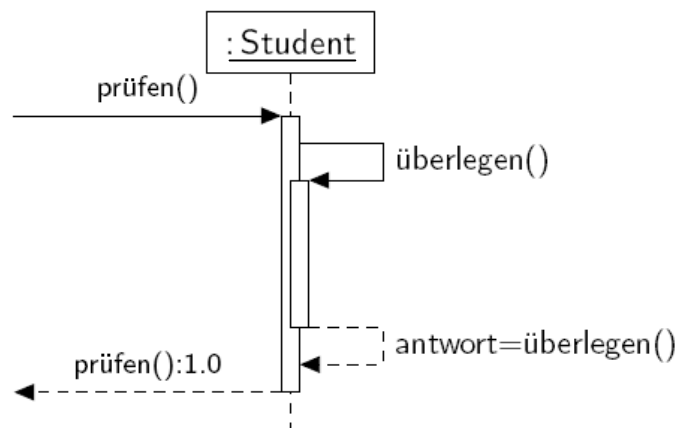
Erzeugen und Zerstören von Objekten

- Erzeugungsnachrichten erschaffen Kommunikationspartner und damit verbundene Lebenslinien. Gelegentlich findet man die Erzeugungsnachricht auch mit *create* beschriftet.
- Die Zerstörung eines Objektes wird durch den Abbruch der Lebenslinie und einem Kreuz an dessen Ende dargestellt.



Selbstdelegation („Aufruf von sich selbst“)

Selbstdelegation wird unter anderem verwendet, um den Aufruf von privaten Methoden, die nur innerhalb des Objekts sichtbar sind, darzustellen.



Ab UML 2.0 sind weitere Sprachmittel zur Ablaufkontrolle hinzugekommen. Diese betrachten wir hier nicht weiter.

Grundsätzlich gilt wie immer in der Modellierung: Nur die wesentlichen Aktionssequenzen sollten dargestellt werden und nur die wichtigen Elemente daraus.

Rückantworten werden beispielsweise häufig weggelassen, wenn kein (besonderer) Rückgabewert zurückgeliefert wird.