11. Vererbung und Polymorphie
Die Filter gegen Codeverschmutzung

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 17-0.3, 07.04.17

1) Vererbung zwischen Klassen
2) Abstrakte Klassen und Schnittstellen
3) Polymorphie
4) Generische Klassen

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Begleitende Literatur

- Das **Vorlesungsbuch** von Pearson: **Softwaretechnologie für Einsteiger.** Vorlesungsunterlage für die Veranstaltungen an der TU Dresden. Pearson Studium, 2014. Enthält ausgewählte Kapitel aus:
  - UML: Harald Störrle. UML für Studenten. Pearson 2005. Kompakte Einführung in UML 2.0.
  - Softwaretechnologie allgemein: W. Zuser, T. Grechenig, M. Köhle. Software Engineering mit UML und dem Unified Process. Pearson.
  - Bernd Brügge, Alan H. Dutoit. Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java. Pearson Studium/Prentice Hall.
  - Erhältlich bei Thalia (Nürnberger Platz), Thalia (Haus des Buches)
- Noch ein sehr gutes, umfassend mit Beispielen ausgestattetes Java-Buch:
  - C. Heinisch, F. Müller, J. Goll. Java als erste Programmiersprache. Vom Einsteiger zum Profi. Teubner.
- Für alle, die sich nicht durch Englisch abschrecken lassen:
- Safari Books, von unserer Bibliothek SLUB gemietet:
  - http://proquest.tech.safaribooksonline.de/
- Free Books: http://it-ebooks.info/
  - Kathy Sierra, Bert Bates: Head-First Java http://it-ebooks.info/book/255/

Obligatorische Literatur

- ST für Einsteiger Kap. 9, Teil II (Störrle, Kap. 5.2.6, 5.6)
- Zuser Kap 7, Anhang A
- Java
  - http://docs.oracle.com/javase/tutorial/java/index.html is the official Oracle tutorial on Java classes
  - Balzert LE 9-10
  - Boles Kap. 7, 9, 11, 12

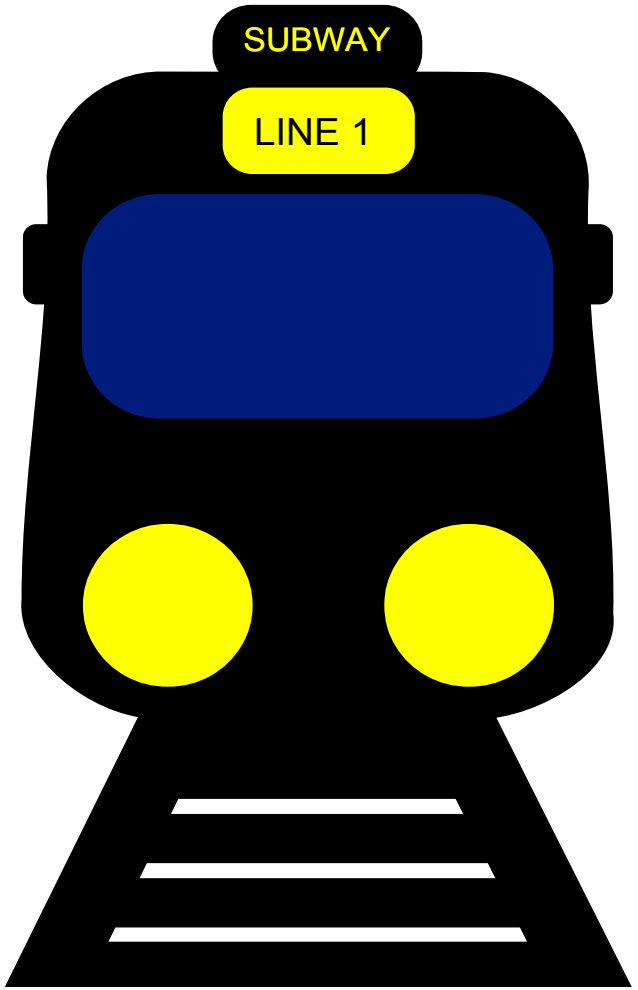
"Objektorientierte Softwareentwicklung"
Hörsaalübung
Fr, 13:00 HSZ 03, Dr. Demuth

Ziele

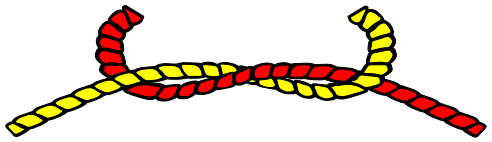
- Elementare Techniken der Wiederverwendung von objektorientierten Programmen kennen
  - Einfache Vererbung zwischen Klassen, konzeptuell und im Speicher
  - Abstrakte Klassen und Schnittstellen verstehen
  - Merkmalssuche in einer Klasse und in der Vererbungshierarchie aufwärts nachvollziehen können
  - Überschreiben von Merkmalen verstehen
  - Generische Typen zur Vermeidung von Fehlern
- Dynamische Architektur eines objektorientierten Programms verstehen
  - Polymorphie im Speicher verstehen
  - Objekte vs. Rollen verstehen

Java Herunterladen

- Das Java Development Kit (JDK) 8
- www.javasoft.com
- http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

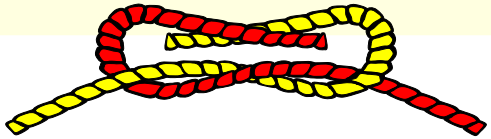


Problem: Was tut man gegen Codeverschmutzung bzw. Copy-And-Paste-Programming (CAPP)?

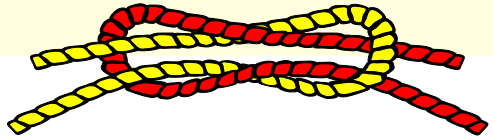


Codeverschmutzung durch CAPP: Nach einer Weile entdeckt man in einem gewachsenen System, dass jede Menge Code repliziert wurde
Große Software kann 10-20% an *Replikaten (code clones)* enthalten *(Code-Explosion, code bloat)*

Plagiat
Ignoranz
Aufwandsreduktion
Mangelnde Anforderungsanalyse der Anwendungsdomäne



Aufwandsreduktion:
Wiederverwendung von Tests



- http://c2.com/cgi/wiki?CopyAndPasteProgramming
- http://en.wikipedia.org/wiki/Copy_and_paste_programming

Linking Replicates

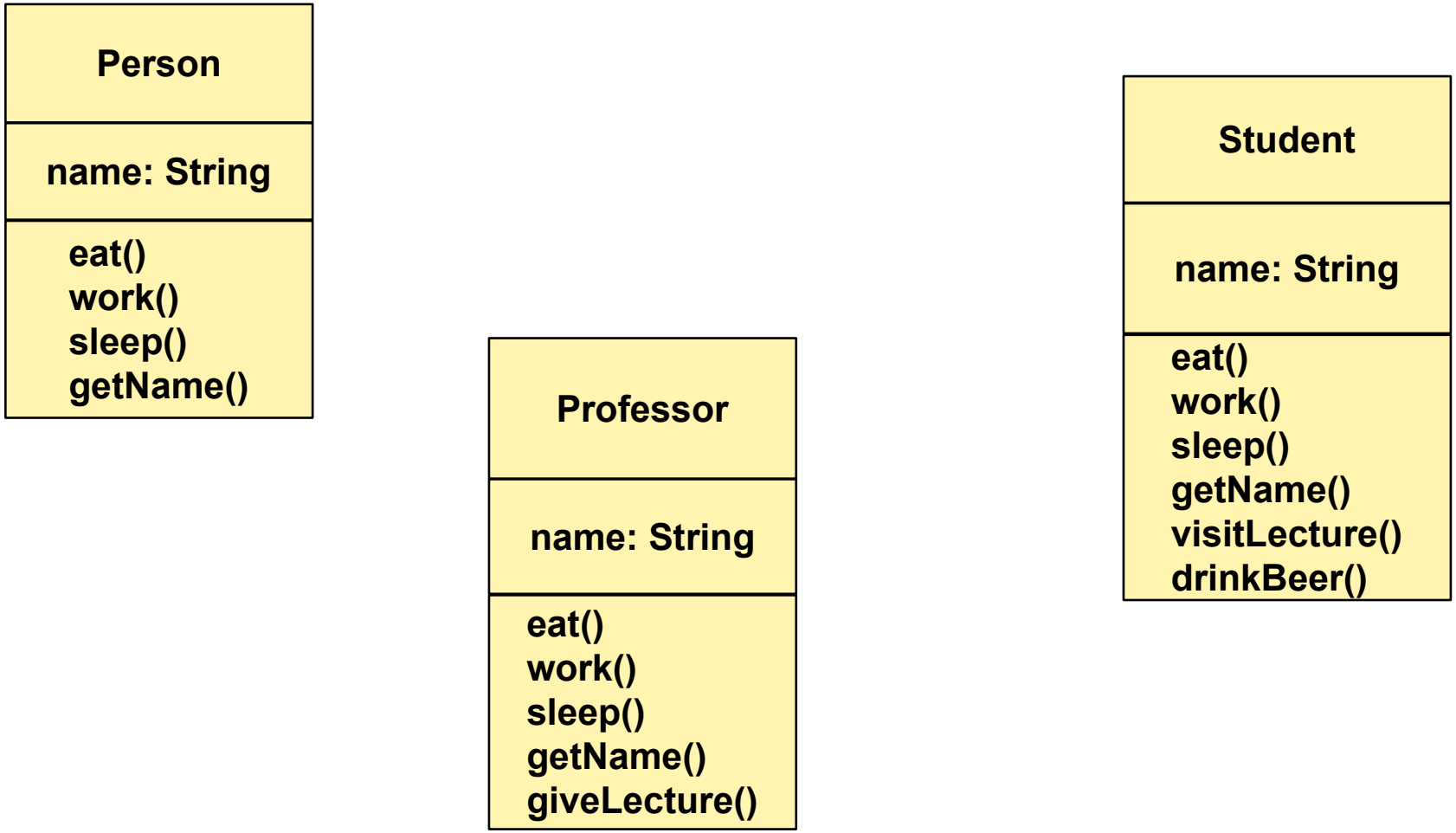
- Interessante Technik, Code-Replikate zu finden und dauerhaft zu verlinken:
  - Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In VL/HCC, pages 173-180. IEEE Computer Society, 2004.
  - http://harmonia.cs.berkeley.edu/papers/toomim-linked-editing.pdf
- Optional, mit vielen schönen Visualisierungen von Code Clones:
  - Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In WCRE, pages 100-109. IEEE Computer Society, 2004.
  - http://rmod.lille.inria.fr/archives/papers/Rieg04b-WCRE2004-ClonesVisualizationSCG.pdf

11.1 Vererbung zwischen Klassen
beseitigt Codereplikate

Ähnlichkeit von Klassen sollten in Oberklassen ausfaktorisiert werden

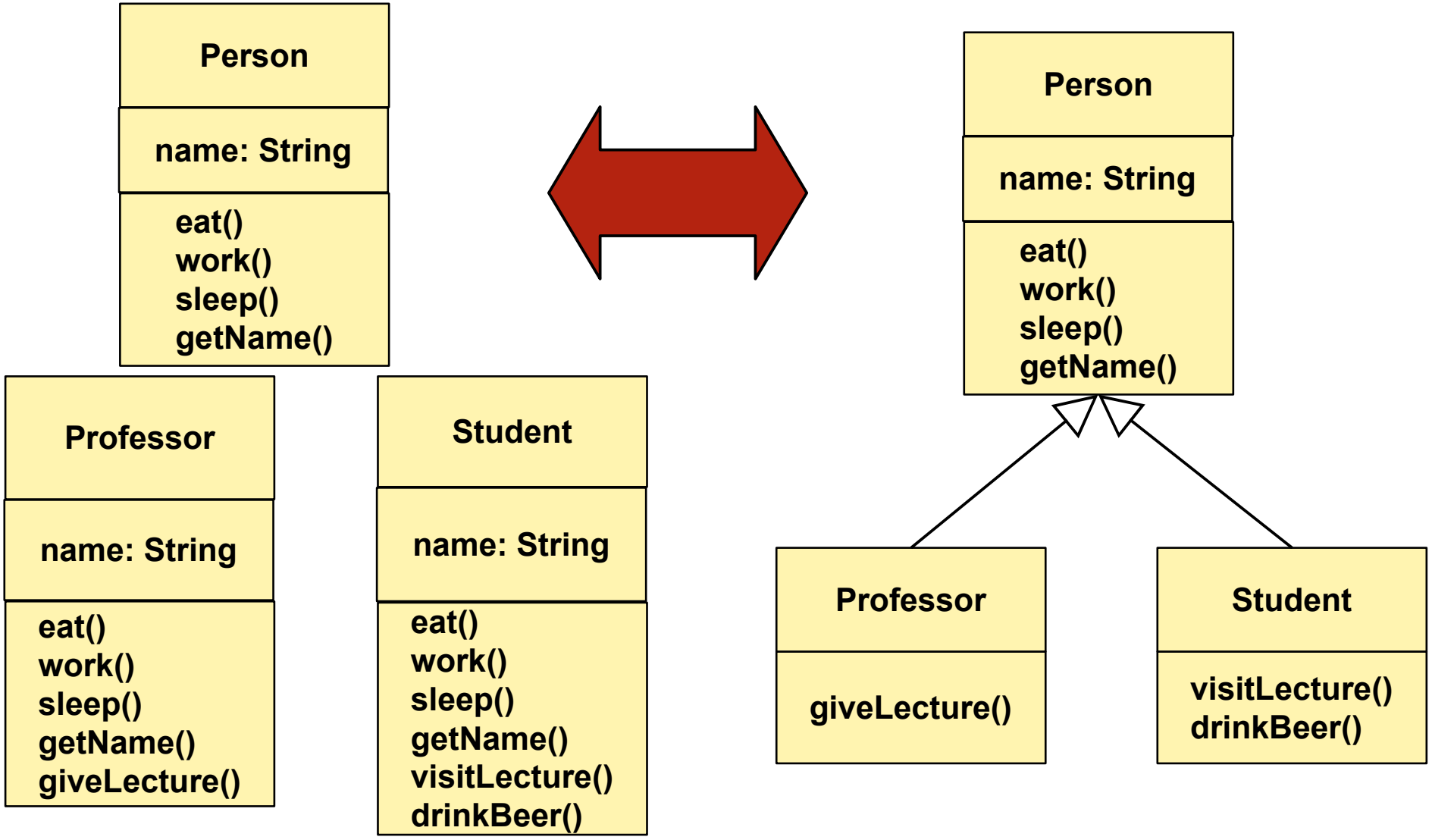
Codeverschmutzung am Beispiel

- **Hier:** Person wurde zu Professor und Student kopiert und danach erweitert



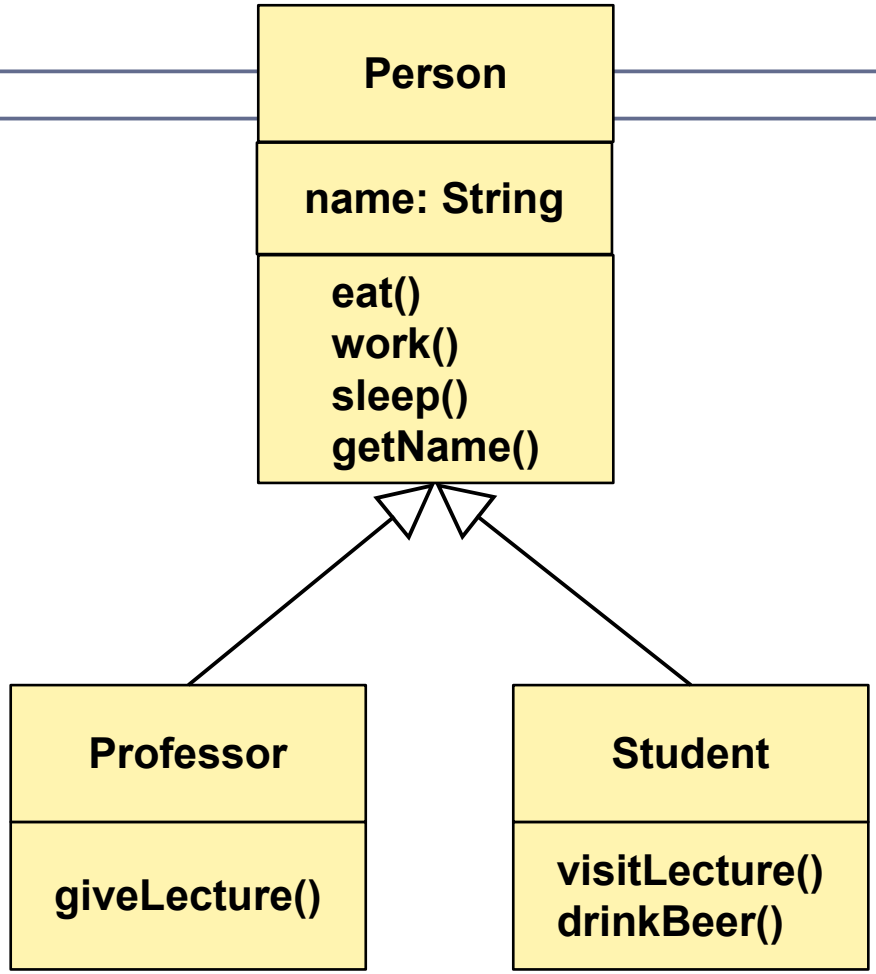
Einfache Vererbung

- **Vererbung**: Eine Klasse kann Merkmale von einer Oberklasse **erben**
- **Hier:** Oberklasse Person enthält alle gemeinsamen Merkmale der Unterklasse



Einfache Vererbung

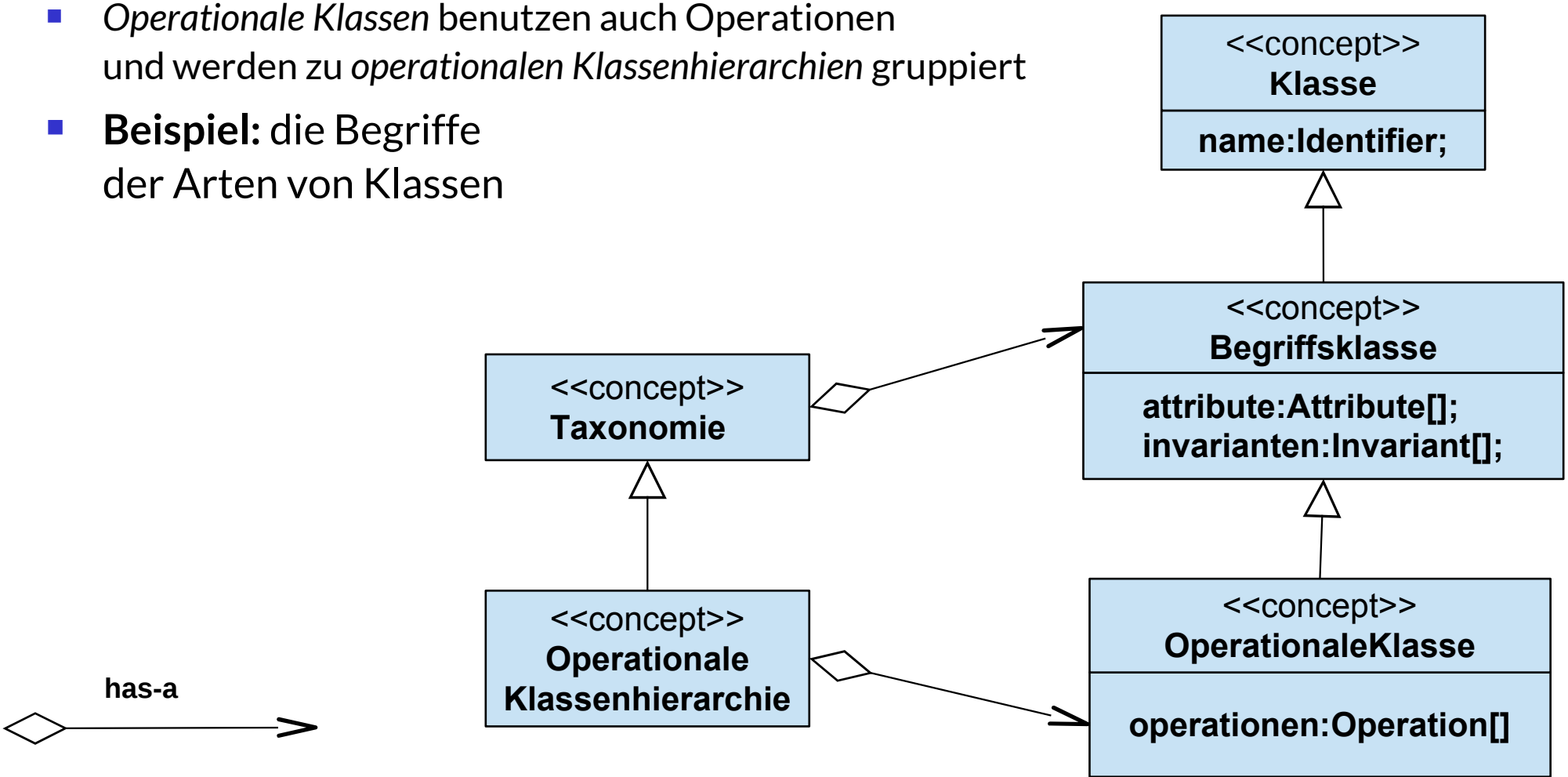
- **Vorteil:** Vererbung drückt Gemeinsamkeiten aus
  - Die Unterklasse ist damit ähnlich zu dem Elter und den Geschwistern
  - Vererbung stellt *is-a*-Beziehung her (<)
- **Hier**: Oberklasse Person enthält alle gemeinsamen Merkmale der Unterklasse
- Bei **einfacher Vererbung** hat jede Klasse nur *eine* Oberklasse
  - Dann ist die Vererbungsrelation ein Baum




```
// Java
Professor extends Person {}; Student extends Person{};
```

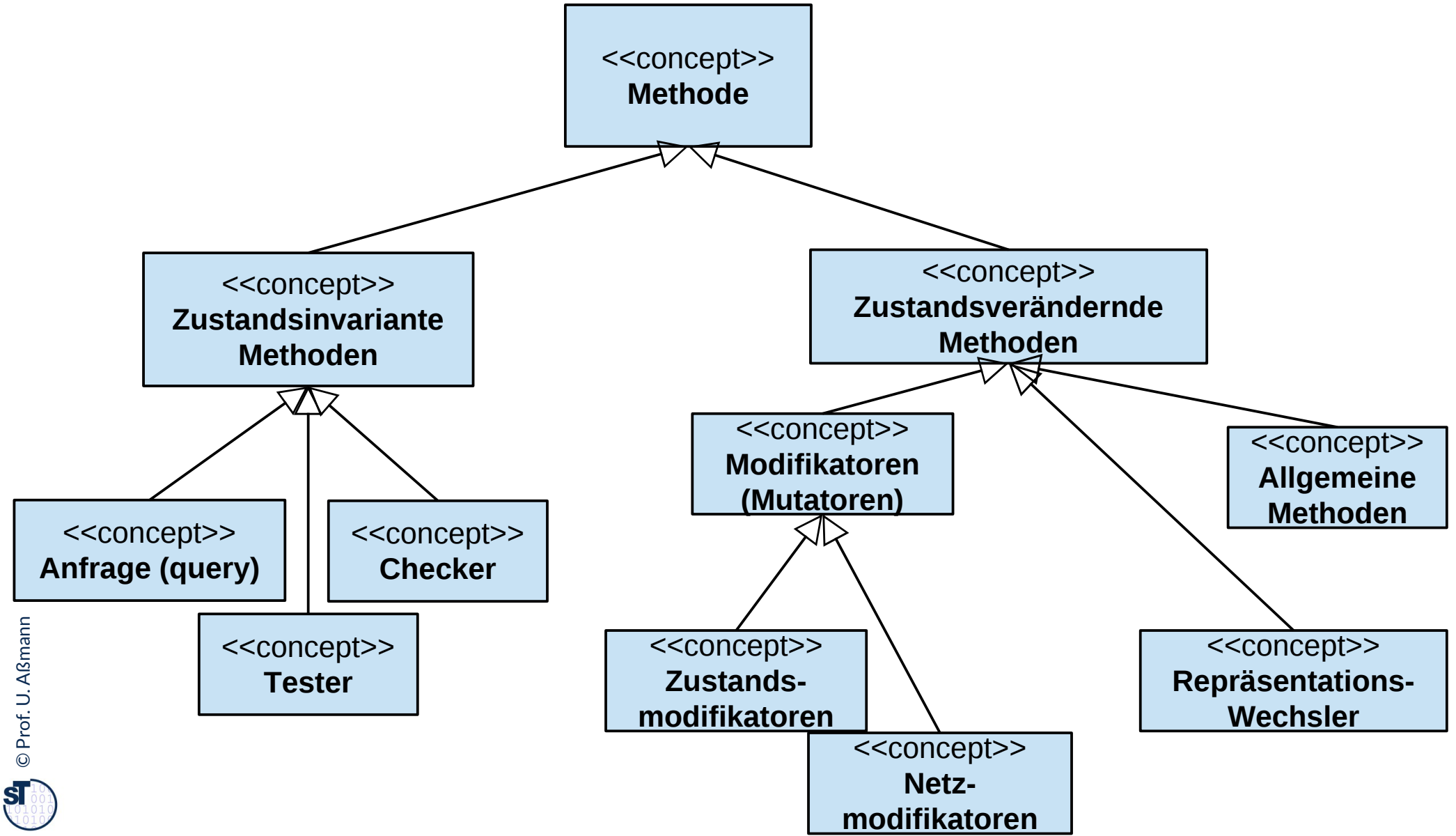
Q1: Begriffshierarchien (Taxonomien) nutzen einfache Vererbung

- Domänenmodelle werden durch *Klassifikation* der Domänenobjekte und Domänenkonzepte ermittelt
- Klassifikationen führen zu ***Begriffshierarchien (Taxonomien)***
  - *Begriffsklassen* besitzen nur Attribute und Invarianten (leicht blau)
- *Operationale Klassen* benutzen auch Operationen und werden zu *operationalen Klassenhierarchien* gruppiert
- **Beispiel:** die Begriffe der Arten von Klassen



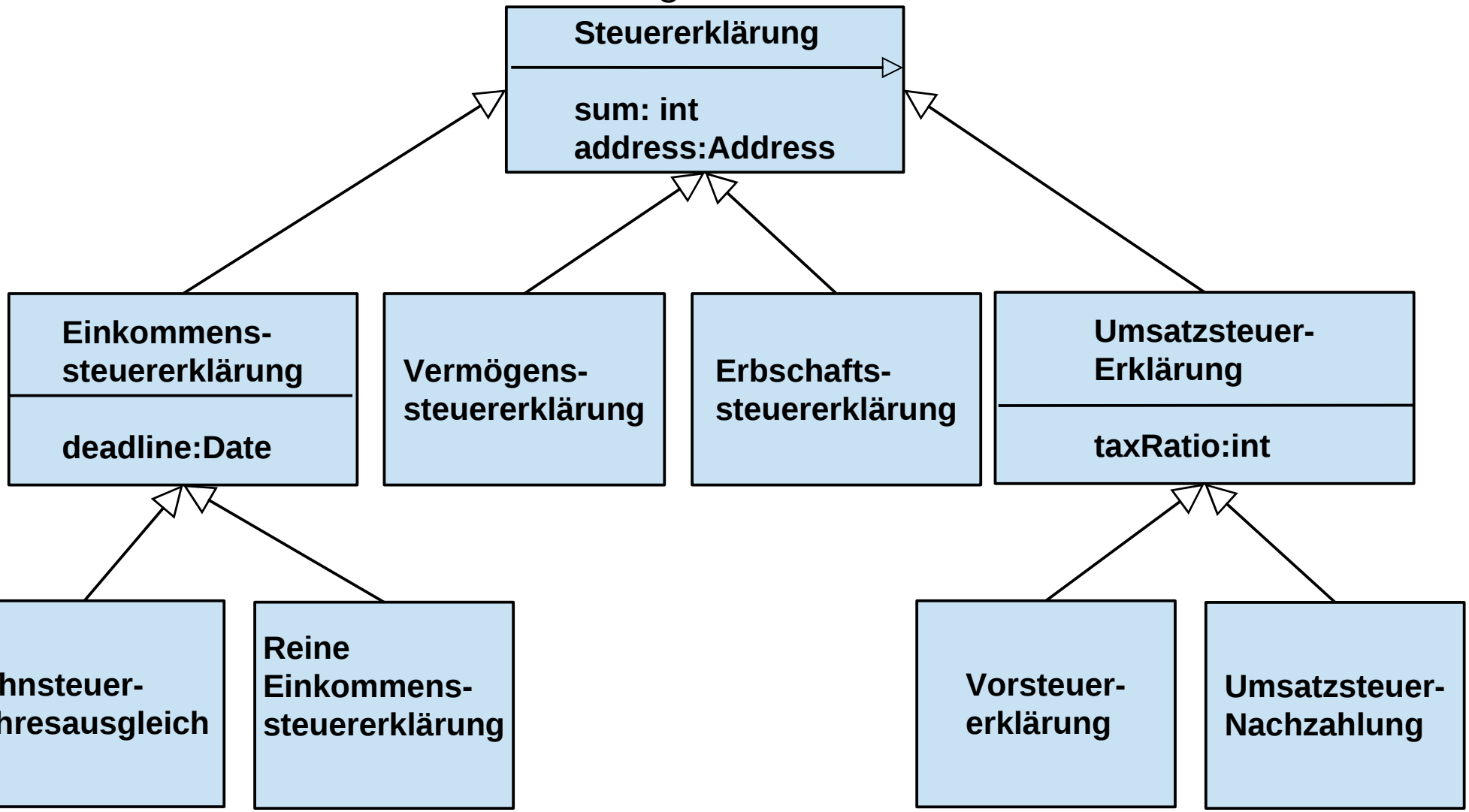
Bsp: Begriffshierarchie (Taxonomie) der Methodenarten

- **Wiederholung**: Welche Arten von Methoden gibt es in einer Klasse?



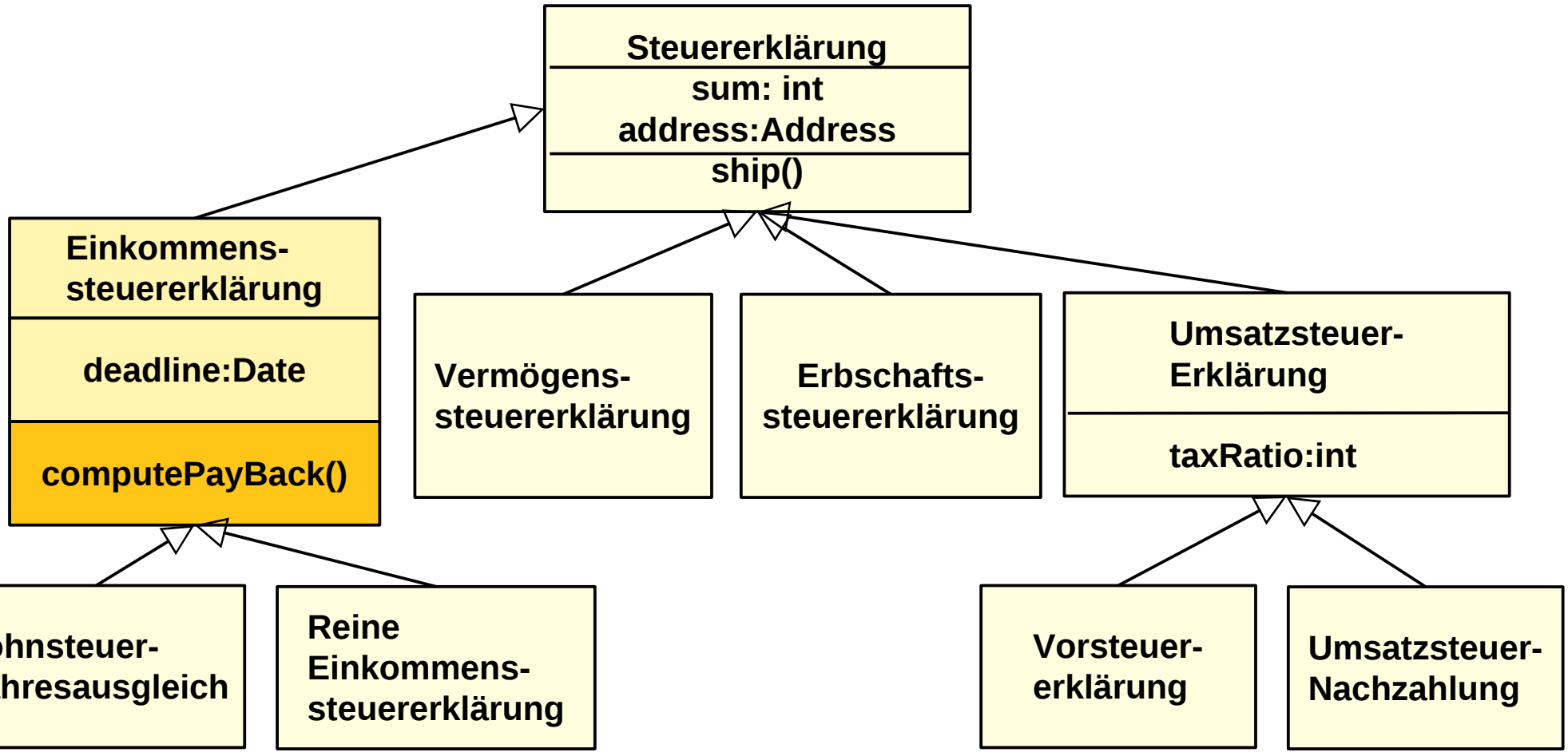
Bsp. Taxonomie der Steuererklärungen

- Domäne: Finanzbuchhaltung
- Das deutsche Steuerrecht kennt viele Arten von Steuererklärungen
- Eine Klassifikation führt zu einer Begriffshierarchie



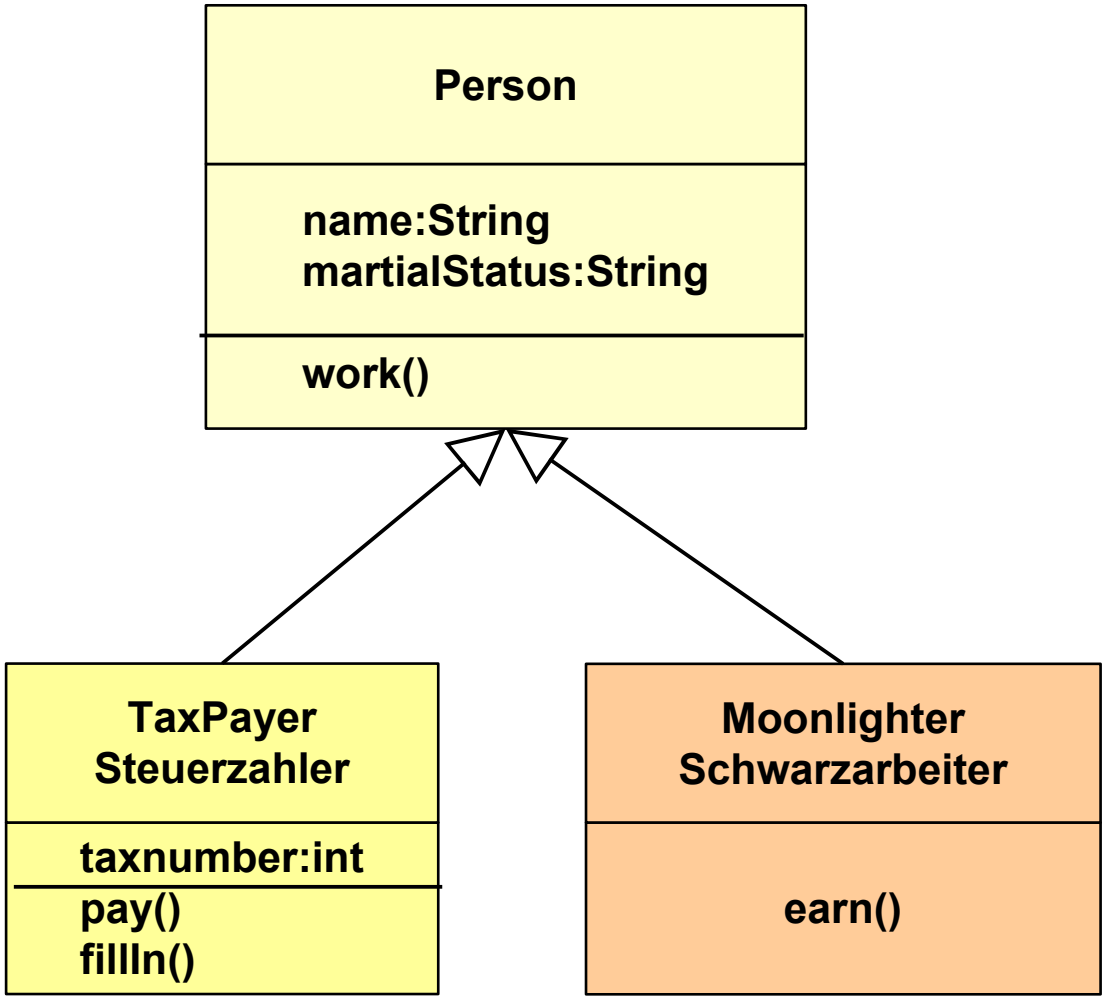
Bsp. Erweiterung einer Begriffshierarchie hin zu operationalen Klassenhierarchie

- Programmiert man eine Steuerberater-Software, muss man die Begriffshierarchie der Steuererklärungen als Klassen einsetzen.
- Daneben sind aber die Klassen um eine neue **Abteilung (compartment)** mit **Operationen** zu erweitern, denn innerhalb der Software müssen sie ja etwas tun.



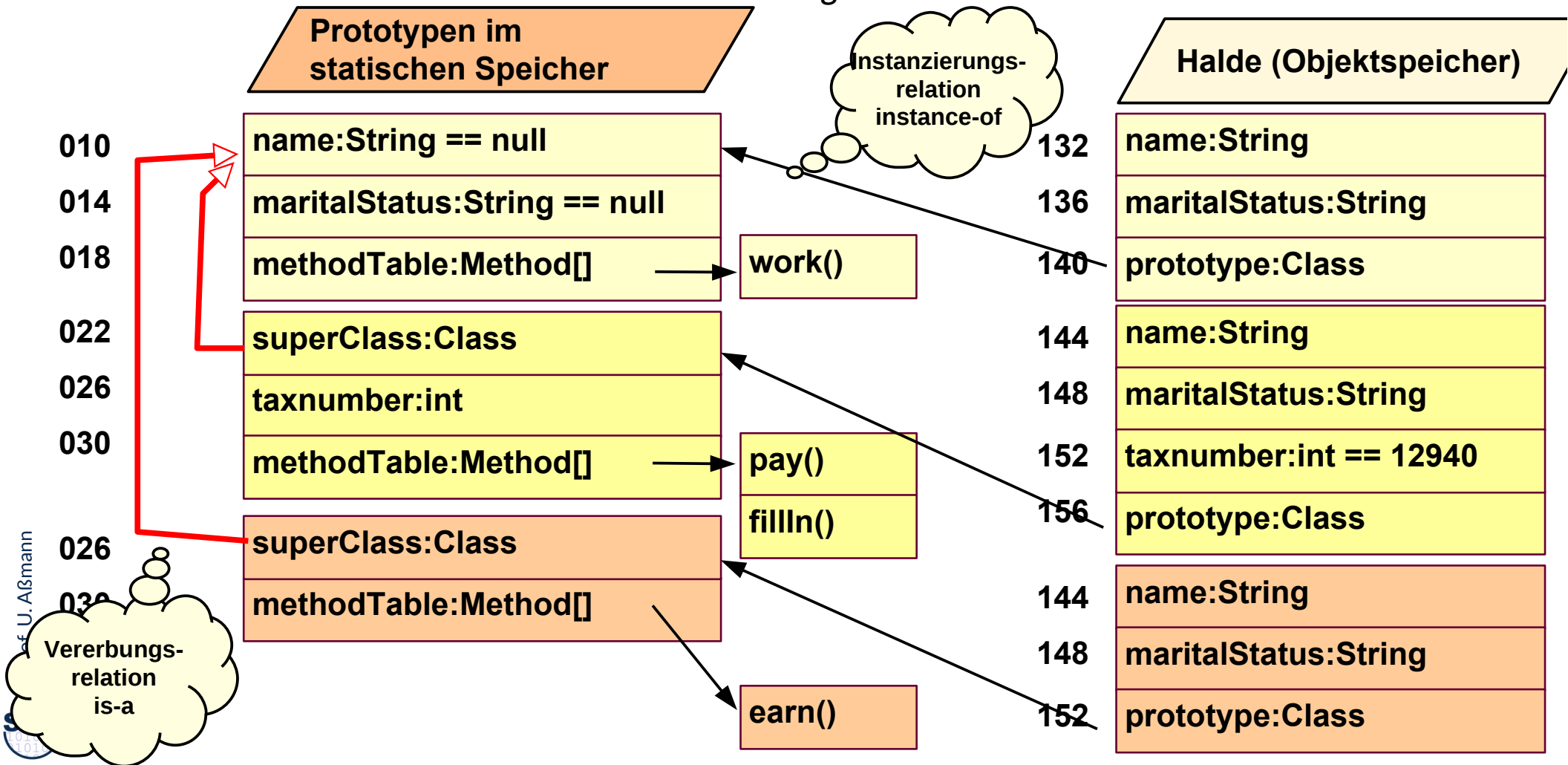
11.1.2 Vererbung im Speicher

- ... am Beispiel Steuerzahler



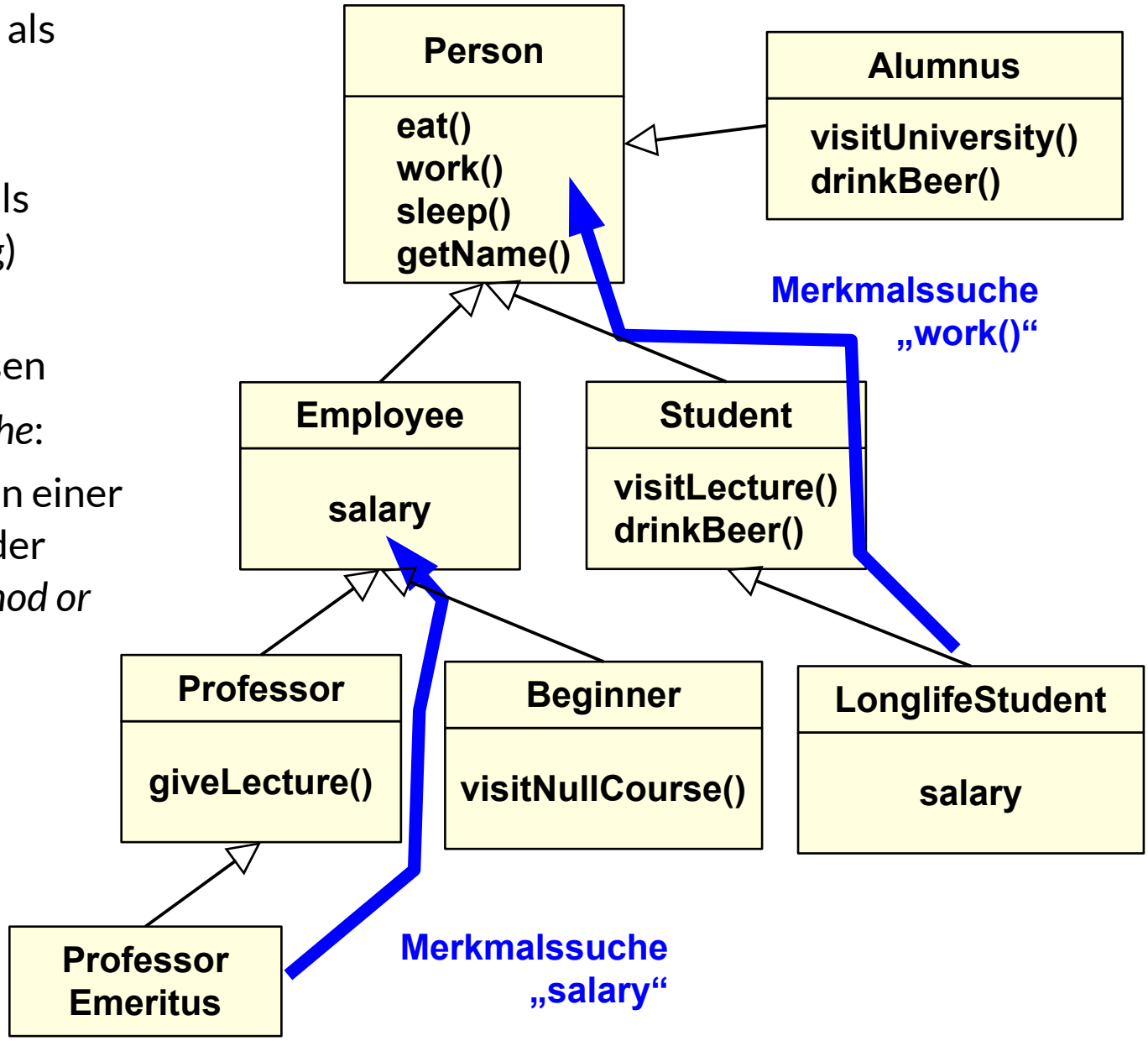
Vererbung im Speicher

- Die Vererbungsrelation wird im Speicher als *Baum* zwischen den Prototypen der Ober- und Unterklassen dargestellt (Verzeigerung von unten nach oben)
  - Unterscheide davon die Objekt-Prototyp-Relation instance-of!
- Methoden werden zwischen den Klassen *geteilt*



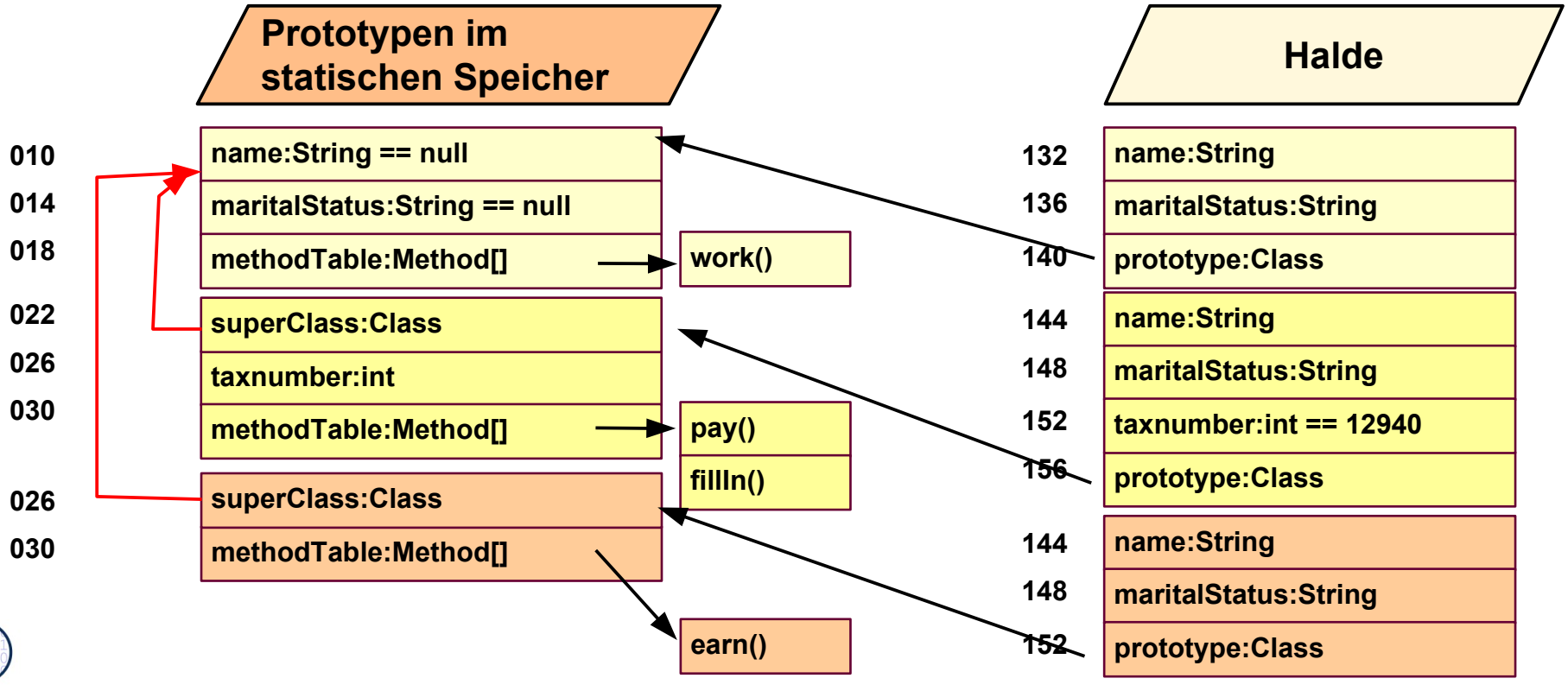
Merkmalssuche im Vererbungsbaum

- Oberklassen sind *allgemeiner* als Unterklassen (Prinzip der Generalisierung)
- Unterklassen sind *spezieller* als Oberklassen (Spezialisierung)
  - Unterklassen *erben* alle Merkmale der Oberklassen
- *Methoden-* bzw. *Merkmalssuche*:
  - Wird ein Merkmal nicht in einer Klasse definiert, wird in der Oberklasse *gesucht (method or feature resolution)*



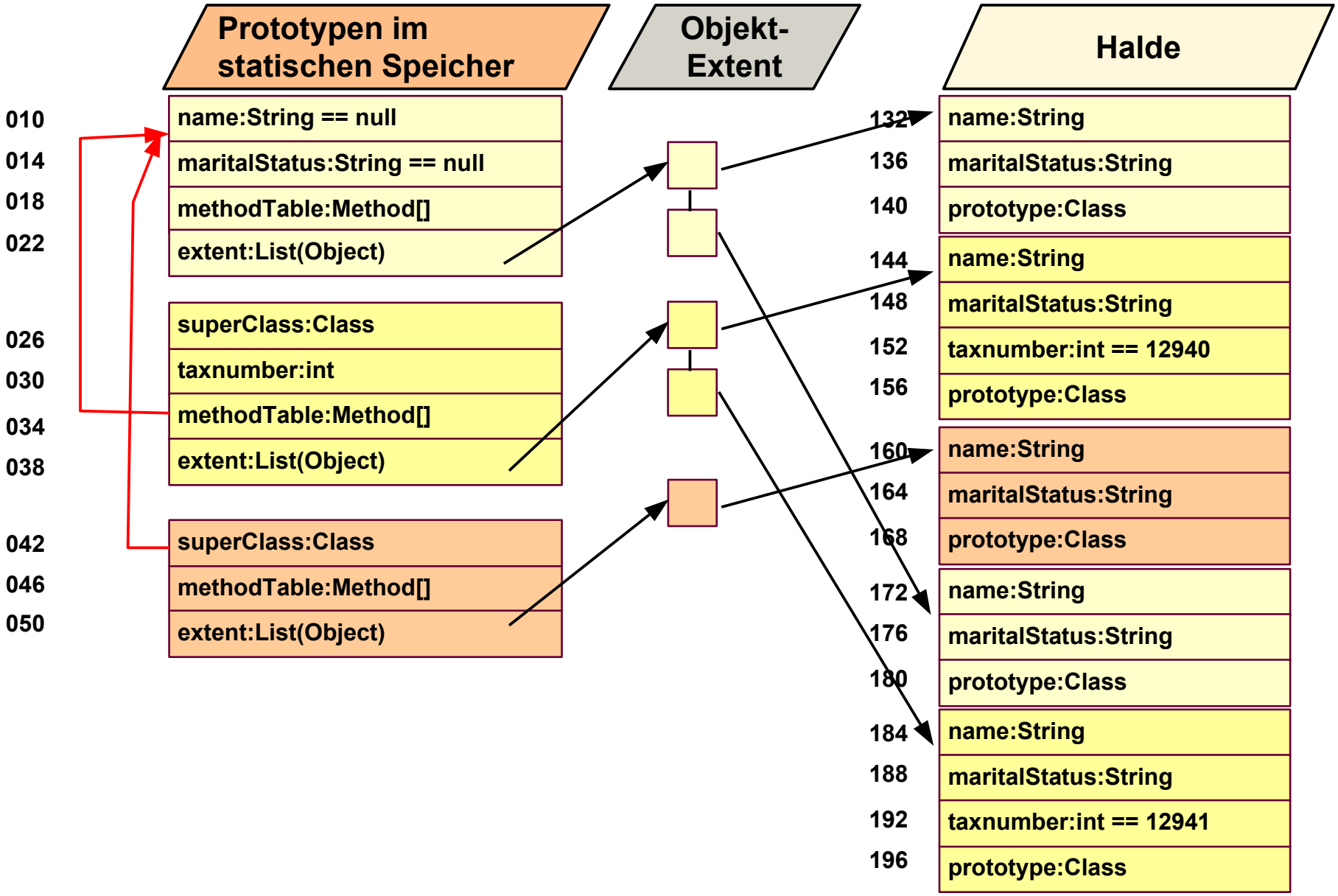
Merkmalssuche im Speicher - Beispiele

1) Suche Attribut *name* in Steuerzahler: direkt vorhanden
2) Suche Methode *pay()* in Steuerzahler: Schlage Prototyp nach, finde in Methodentabelle des Prototyps
3) Suche Methode *work()* in Steuerzahler: Schlage Prototyp nach, Schlage Oberklasse nach (Person), finde in Methodentabelle von Person
4) Suche Methode payback() in Steuerzahler: Schlage Prototyp nach, Schlage Oberklasse nach (Person); exisitert nicht in Methodentabelle von Person. Da keine weitere Oberklasse existiert, wird ein Fehler ausgelöst *"method not found" "message not understood"*



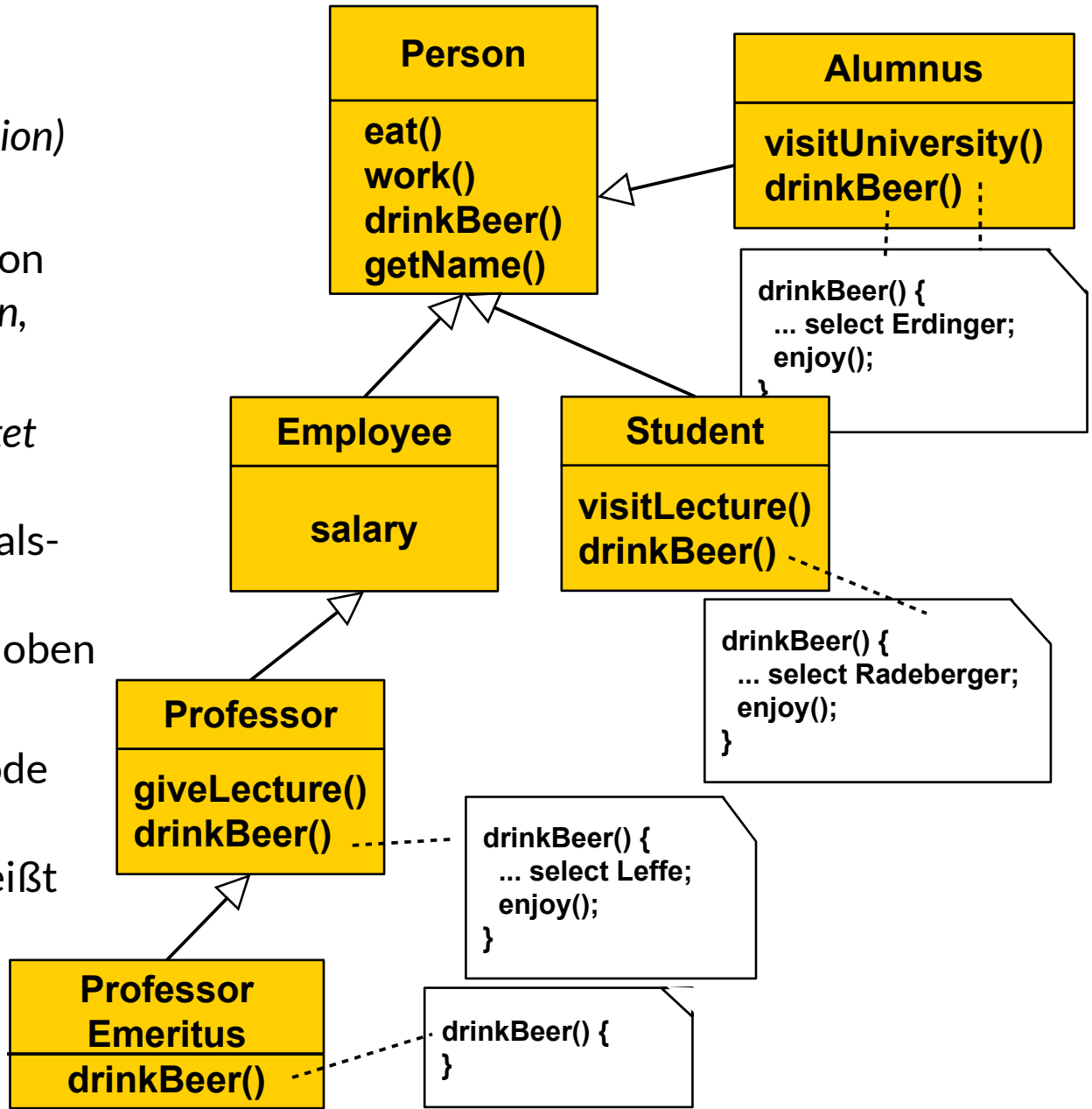
Objekt-Extent im Speicher, mit Vererbung

- Zu einer Klasse vereinige man alle Extents aller Unterklassen



Erweitern und Überschreiben von Merkmalen

- Eine Unterklasse kann neue Merkmale zu einer Oberklasse hinzufügen *(Erweiterung, extension)*
- Definiert eine Unterklasse ein Merkmal erneut, spricht man von einer *Redefinition (Überschreiben, overriding)*
  - Dieses Merkmal *überschattet (verbirgt)* das Merkmal der Oberklasse, da der Merkmals-suchalgorithmus in der Hierarchie von unten nach oben sucht.
  - Die überschriebene Methode hat mehrere Implementierungen und heißt *polymorph* oder *virtual*

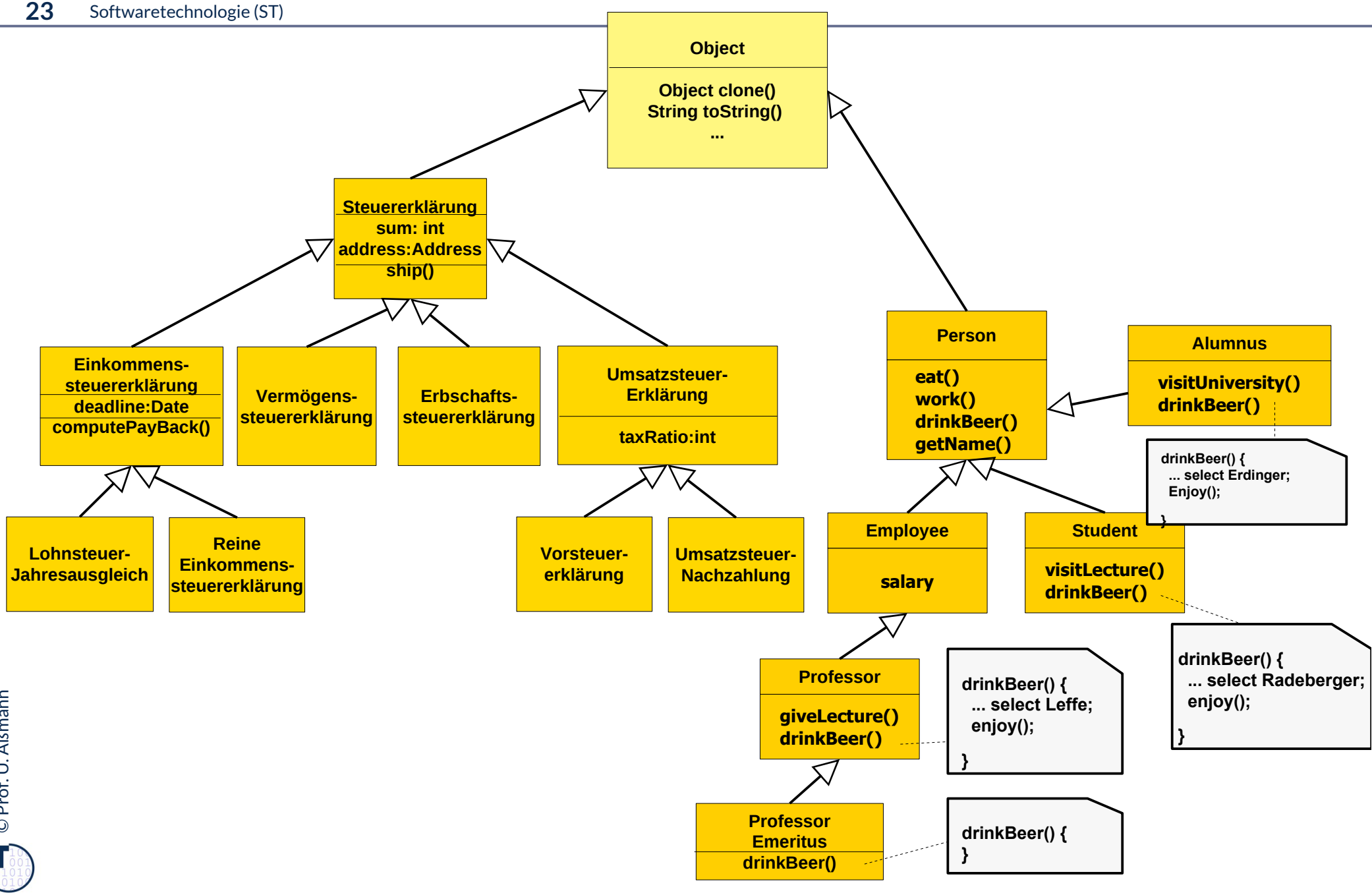


11.1.3. Die oberste Klasse von Java: "Object"

- **java.lang.Object:** allgemeine Eigenschaften aller Objekte und Klassen
  - Jede Klasse ist Unterklasse von Object ("extends Object").
  - Diese Vererbung ist *implizit* (d.h. man kann "extends Object" weglassen).
  - Wiederverwendung in der gesamten JDK-Bibliothek!
- Jede Klasse kann die Standard-Operationen überdefinieren:
  - equals: Objektgleichheit (Standard: Referenzgleichheit)
  - hashCode: Zahlcodierung
  - toString: Textdarstellung, z.B. für println()

```
class Object {
      protected Object clone (); // kopiert das Objekt
      public boolean equals (Object obj);
          // prüft auf Gleichheit zweier Objekte
      public int hashCode();      // produce a unique identifier
      public String toString();  // produce string representation
      protected void finalize(); // lets GC run
      Class getClass();          // gets prototype object
}
```

Vererbung von Object auf Anwendungsklassen



11.E1 Exkurs: Lernen mit Begriffshierarchien

Begriffshierarchien können zum Lernen eingesetzt werden

Der einzige Weg, auf welchem wahre Kenntnis erreicht werden kann, ist durch liebesvolles Studium.

Carl Hilty (1831 - 1909), Schweizer Staatsrechtler und Laientheologe

http://www.aphorismen.de/

Blooms Taxonomie des Lernens

- [Wikipedia, Lernziele] Die 6 Stufen im kognitiven Bereich lauten:
- **Lehrlingsschaft**
  - **Stufe 1) Kenntnisse / Wissen:** Kenntnisse konkreter Einzelheiten wie Begriffe, Definitionen, Fakten, Daten, Regeln, Gesetzmäßigkeiten, Theorien, Merkmalen, Kriterien, Abläufen; Lernende können Wissen abrufen und wiedergeben.
  - **Stufe 2) Verstehen:** *Lernende können Sachverhalt mit eigenen Worten erklären oder zusammenfassen; können Beispiele anführen, Zusammenhänge verstehen; können Aufgabenstellungen interpretieren.*
- **Gesellenschaft**
  - **Stufe 3) Anwenden:** Transfer des Wissens, problemlösend; Lernende können das Gelernte in neuen Situationen anwenden und unaufgefordert Abstraktionen verwenden oder abstrahieren.
  - **Stufe 4) Analyse:** Lernende können ein Problem in einzelne Teile zerlegen und so die Struktur des Problems verstehen; sie können Widersprüche aufdecken, Zusammenhänge erkennen und Folgerungen ableiten, und zwischen Fakten und Interpretationen unterscheiden.
  - **Stufe 5) Synthese:** Lernende können aus mehreren Elementen eine neue Struktur aufbauen oder eine neue Bedeutung erschaffen, können neue Lösungswege vorschlagen, neue Schemata entwerfen oder begründete Hypothesen entwerfen.
- **Meisterschaft**
  - **Stufe 6) Beurteilung:** Lernende können den Wert von Ideen und Materialien beurteilen und können damit Alternativen gegeneinander abwägen, auswählen, Entschlüsse fassen und begründen, und bewusst Wissen zu anderen transferieren, z. B. durch Arbeitspläne.

Lernlandkarten und Lernmatrizen als Hilfsmittel

- Erstellen Sie eine Strukturkarte (concept map) der Vorlesung zur Vorbereitung für die Klausur
- **Vorlesungslandkarte**: Quasi-hierarchische Darstellung der Inhalte der Vorlesung
  - gegliedert wie die Vorlesung
  - gefüllt mit Begriffen, die Sie erklären können (Bloom-Stufe 1+2)
  - gefüllt mit Fragen
- **Vorlesungsmatrix**: Matrixartige Darstellung der Inhalte
  - auf die Vorlesungslandkarte aufbauend
  - Kreuzen mit zweiter Dimension: Querschneidende Aspekte wie Analyse, Design, Entwurfsmuster in die zweite Dimension eintragen
  - Damit die Vorlesungslandkarte in einen zweiten Zusammenhang bringen (Bloom-Stufe 3+4)
- **Übung:** Erstellen Sie eine Vorlesungslandkarte von Vorlesung 10, “Objekte und Klassen”
  - Erstellen Sie eine Vorlesungslandkarte von Vorlesung 11, “Vererbung und Polymorphie”
  - Ermitteln sie querscheidende Aspekte wie Objektallokation, Speicherrepräsentation
  - Entwickeln Sie eine Vorlesungsmatrix

Exkursion: Safari Books Online

- Die SLUB hat für sie eine Menge von Büchern online
- Von innerhalb der TU Dresden **kostenlos lesbar**
- http://proquest.tech.safaribooksonline.de/

Sehr empfohlen für die Technik des Lernens und wiss. Arbeitens:

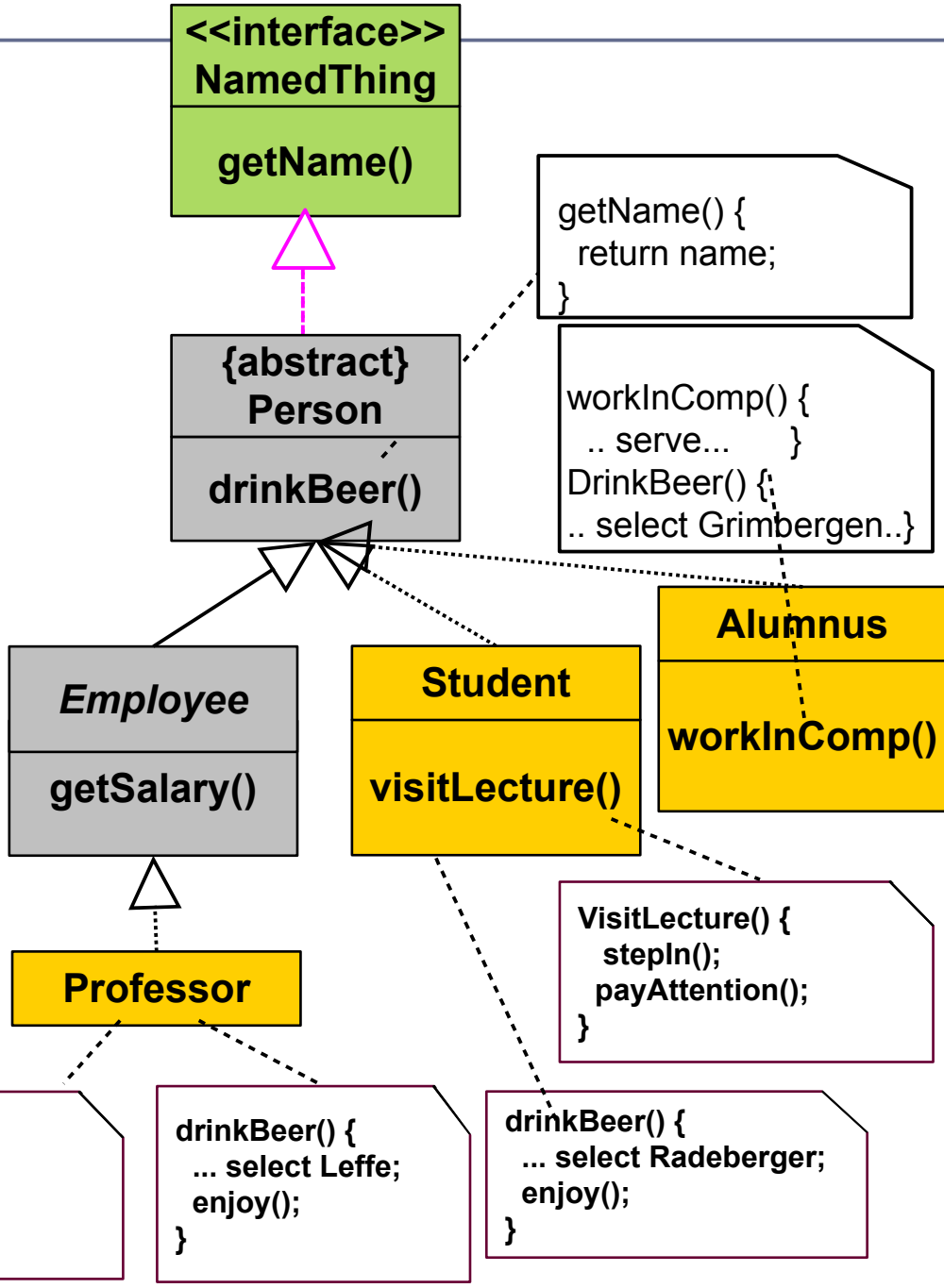
- Stickel-Wolf, Wolf. Wissenschaftliches Arbeiten und Lerntechniken. Gabler. Blau. Sehr gutes Überblicksbuch für Anfänger.
- **Kurs "Academic Skills for Computer Scientists" (3/1/0)**
  - Wintersemester
  - http://st.inf.tu-dresden.de/teaching/asics

11.2 Schnittstellen und Abstrakte Klassen für das Programmieren von Löchern

Typen können verschiedene Formen annehmen.
Eine partiell spezifizierte Klasse (Schnittstelle, abstrakte Klasse, generische Klasse) macht Vorgaben für Anwendungsentwickler

Schnittstellen und Abstrakte Klassen bilden “Haken”, an die man Klassenimplementierungen anhängen kann

- Beim Entwurf von Bibliotheken soll für Anwendungen eine Struktur vorgegeben werden, an die sich alle Anwendungsprogrammierer halten müssen
- **Vorsehen von “Haken” (hooks)** in der Vererbungshierarchie:
- **Abstrakte Klassen** werden mit einem speziellen Markierer (tagged value) gekennzeichnet ({abstract}) oder *kursiv* gemalt
- **Schnittstellen** beschreiben einen Teil der Funktionalität eines Objekts
  - In UML werden sog. Stereotypen vergeben, um Schnittstellen zu kennzeichnen (<<interface>>)
  - Sie dienen dazu, Verhalt Objektes in einem be Kontext festzulegen



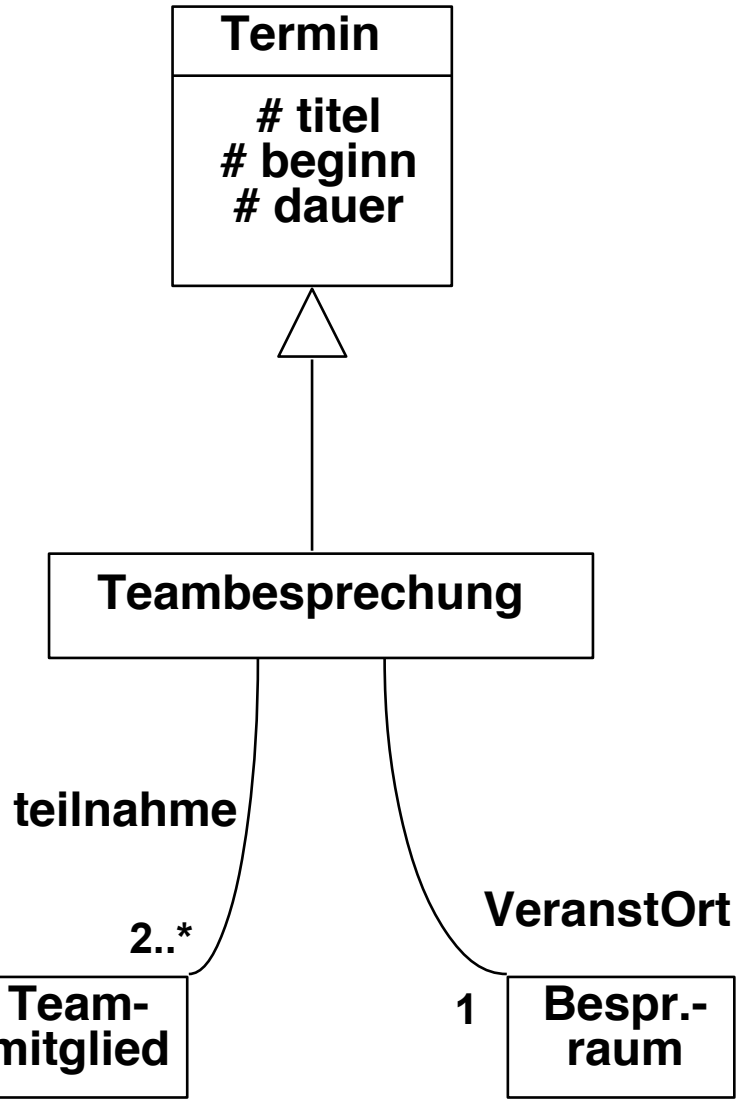
Schnittstellen und Klassen in Java geben “Hooks” vor (“abstract”)


```
interface NamedThing {
  String getName();  // no implementation
}
abstract class Person implements NamedThing {
  String name;
  String getName() { return name; } // implementation exists
  abstract void drinkBeer();        // no implementation
}
abstract class Employee extends Person {
  abstract void getSalary();        // no implementation
}
class Professor extends Employee {  // concrete class
  void getSalary() { goToBank(); withDraw(); }
  void drinkBeer() { .. select Leffe(); enjoy(); }
}
class Student extends Person {      // concrete class
  void visitLecture() { stepIn(); payAttention(); }
  void drinkBeer() { .. select Radeberger(); enjoy(); }
}
```

Schnittstellen und Klassen in Java geben “Hooks” vor


```
interface NamedThing {
  String getName();  // no implementation
}
abstract class Person implements NamedThing {
  String name;
  String getName() { return name; } // implementation exists
  abstract void drinkBeer();        // no implementation
}
abstract class Employee extends Person {
  abstract void getSalary();        // no implementation
}
class Professor extends Employee {  // concrete class
  void getSalary() { goToBank(); withDraw(); }
  void drinkBeer() { .. select Leffe(); enjoy(); }
}
class Student extends Person {      // concrete class
  void visitLecture() { stepIn(); payAttention(); }
  void drinkBeer() { .. select Radeberger(); enjoy(); }
}
class Alumnus extends Person {
   // new concrete class must fit to Person and NamedThing
   void workInComp() { .. serve... }
   void drinkBeer() { ...select Grimbergen... }
}
```

Laufendes Beispiel Terminverwaltung




```
class Termin {
  ...
  protected String titel;
  protected Hour beginn;
  protected int dauer;
  ...
};
```

```
class Teambesprechung
  extends Termin {

  private Teammitglied[] teilnahme;
  private BesprRaum veranstOrt;
  ...
};
```

Beispiel (2): Abstrakte Klassen und Abstrakte Operationen

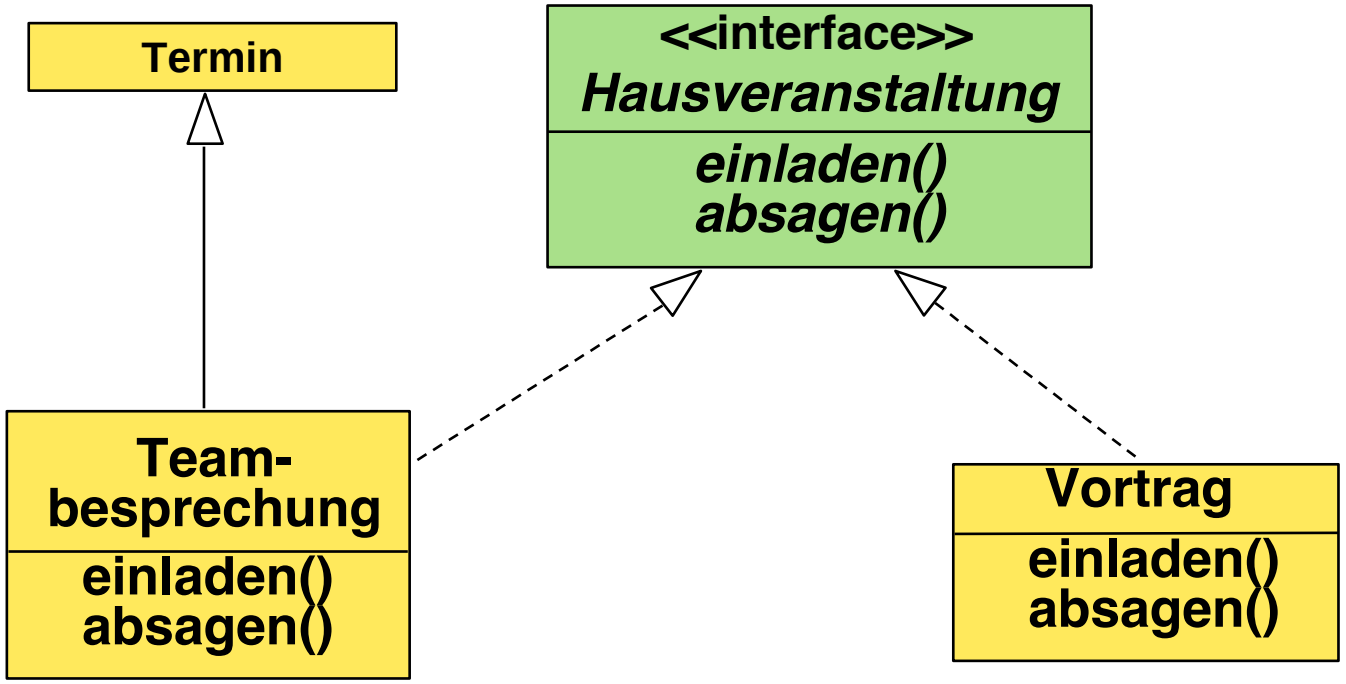

```
abstract class Termin {
  ...
  String titel;
  Hour beginn;
  int dauer;
  ...
  abstract boolean verschieben (Hour neu);
};
```

Jede abstrakt deklarierte Methode muß in einer Unterklasse realisiert werden - sonst können keine Objekte der Unterklasse erzeugt werden!

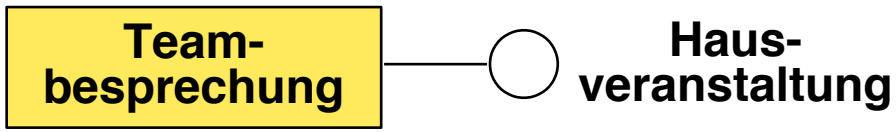
```
class Teambesprechung
  extends Termin {
  ...
  boolean verschieben (Hour neu) {
    boolean ok =
      abstimmen(neu, dauer);
    if (ok) {
      beginn = neu;
      raumFestlegen();
    };
    return ok;
  };
};
```

```
class Betriebsversammlung
  extends Termin {
  ...
  boolean verschieben (Hour neu) {
    beginn = neu;
    raumFestlegen();
    return ok;
  };
};
```

Einfache Vererbung von Typen durch Schnittstellen



Hinweis: “Lutscher”-Notation *(lollipop)* für Schnittstellen „Klasse bietet Schnittstelle an“:

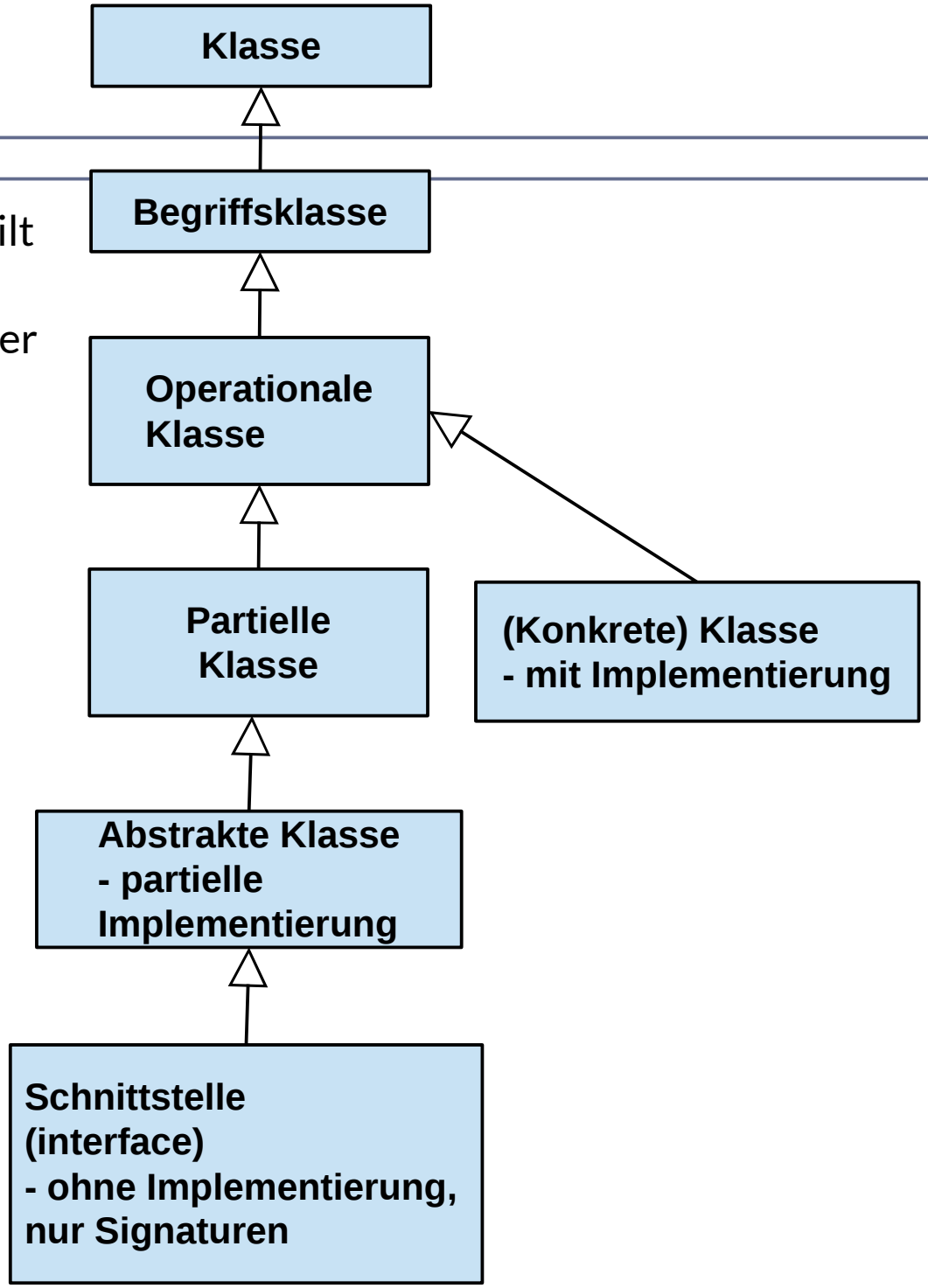


Vergleich von Schnittstellen und abstrakte Klassen

Abstrakte Klasse	Schnittstelle
Enthält Attribute und Operationen	Enthält nur Operationen (und ggf. Konstante)
Kann Default-Verhalten festlegen	Kann kein Default-Verhalten festlegen
Wiederverwendung von Schnittstellen und Code, aber keine Instanzbildung	Redefinition unsinnig
Default-Verhalten kann in Unterklassen überdefiniert werden	Java und UML: Eine Klasse kann mehrere Schnittstellen implementieren
Java: Unterklasse kann nur von einer Klasse erben	Schnittstelle ist eine spezielle Sicht auf eine Klasse

Q2: Begriffshierarchie von Klassen (Erw.)

- Operationale Klassen werden unterteilt in Klassen mit, ohne, und mit Implementierung einer Untermenge der Operationen
- Schnittstellen und Abstrakte Klassen dienen dem Teilen von Typen und partiellem Klassen-Code



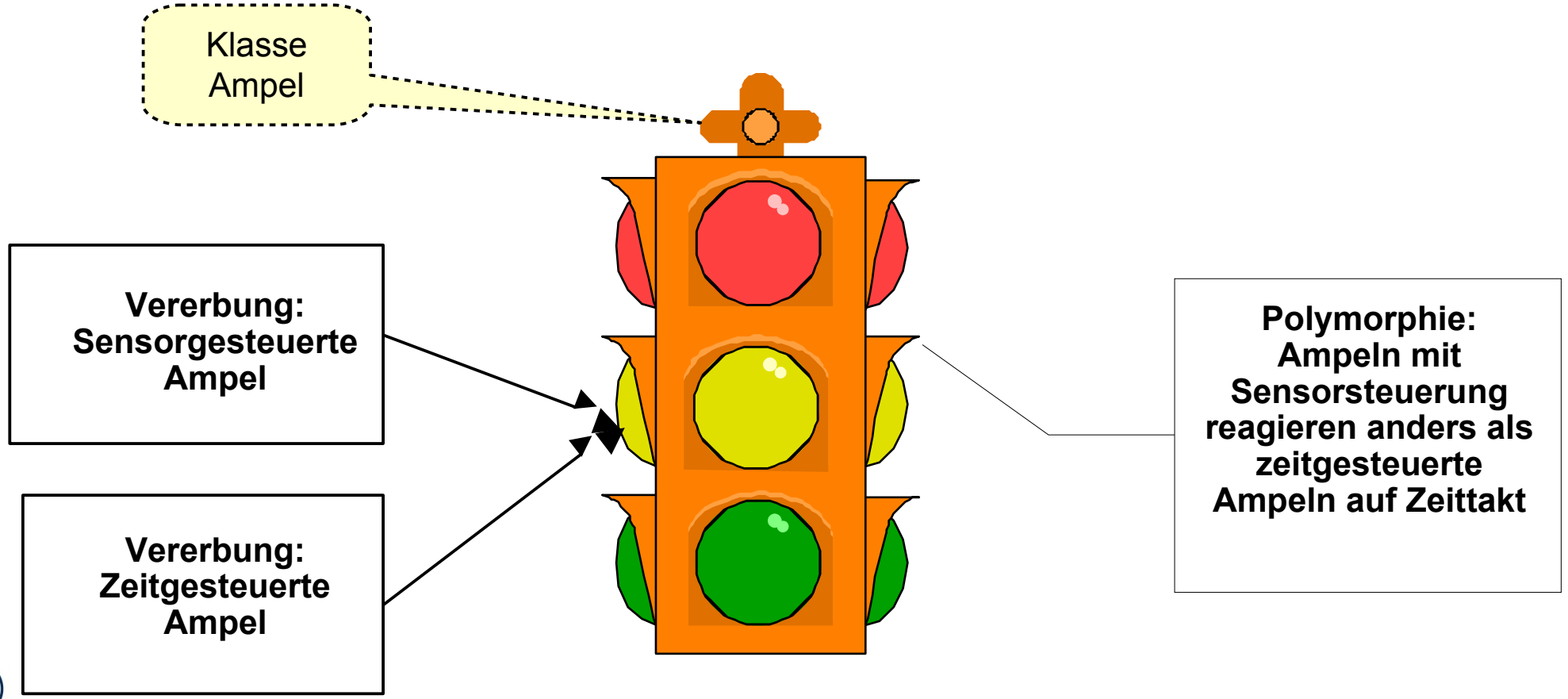
11.3. Polymorphie

.. verändert das Verhalten einer Anwendung, ohne den Code zu verändern

- Polymorphie erlaubt *dynamische Architekturen*
  - Dynamisch wechselnd
  - Unbegrenzt viele Objekte
- Zentraler Fortschritt gegenüber einfachem imperativen Programmieren

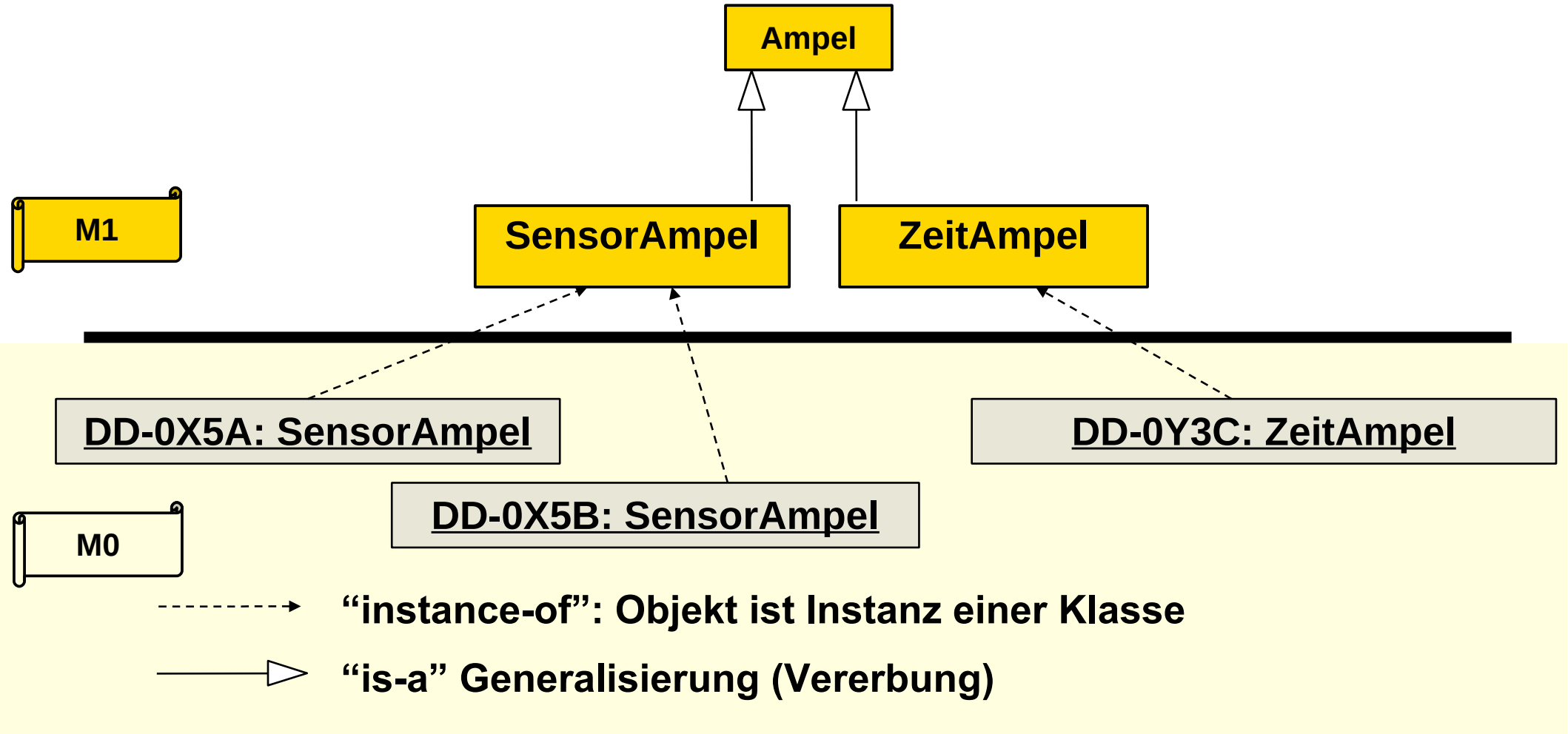
Vererbung und Polymorphie

- Welcher Begriff einer Begriffshierarchie wird verwendet (Oberklassen/ Unterklassen)?
- Wie hängt das Verhalten des Objektes von der Hierarchie ab (spezieller vs allgemeiner)?



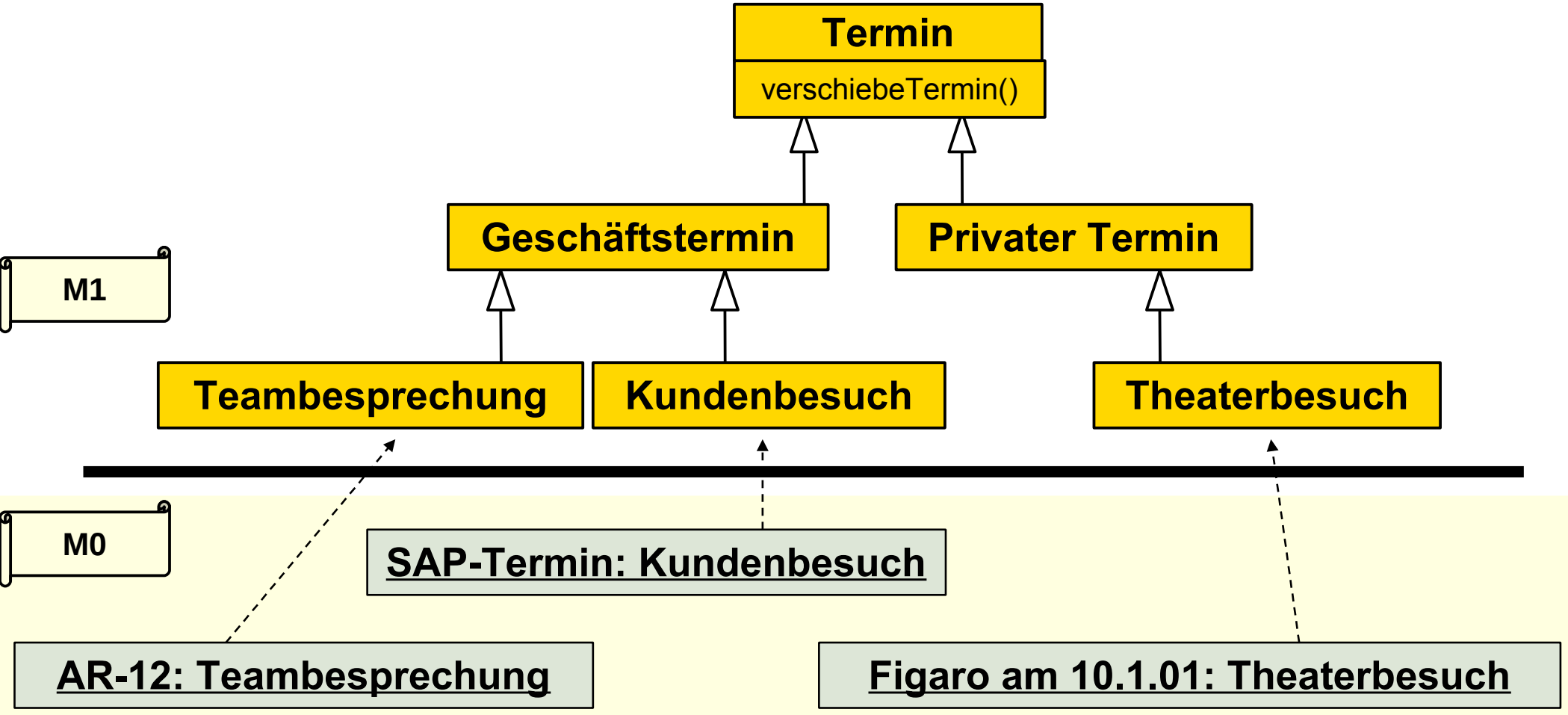
Beispiel: Ampel-Klasse und Ampel-Objekte

- Jede Ampel reagiert auf den Zeittakt.
  - Die Klasse **Ampel** schreibt vor, daß auf die Nachricht „Zeittakt“ reagiert werden muß.
  - Verschiedene Reaktionen der Unterklassen



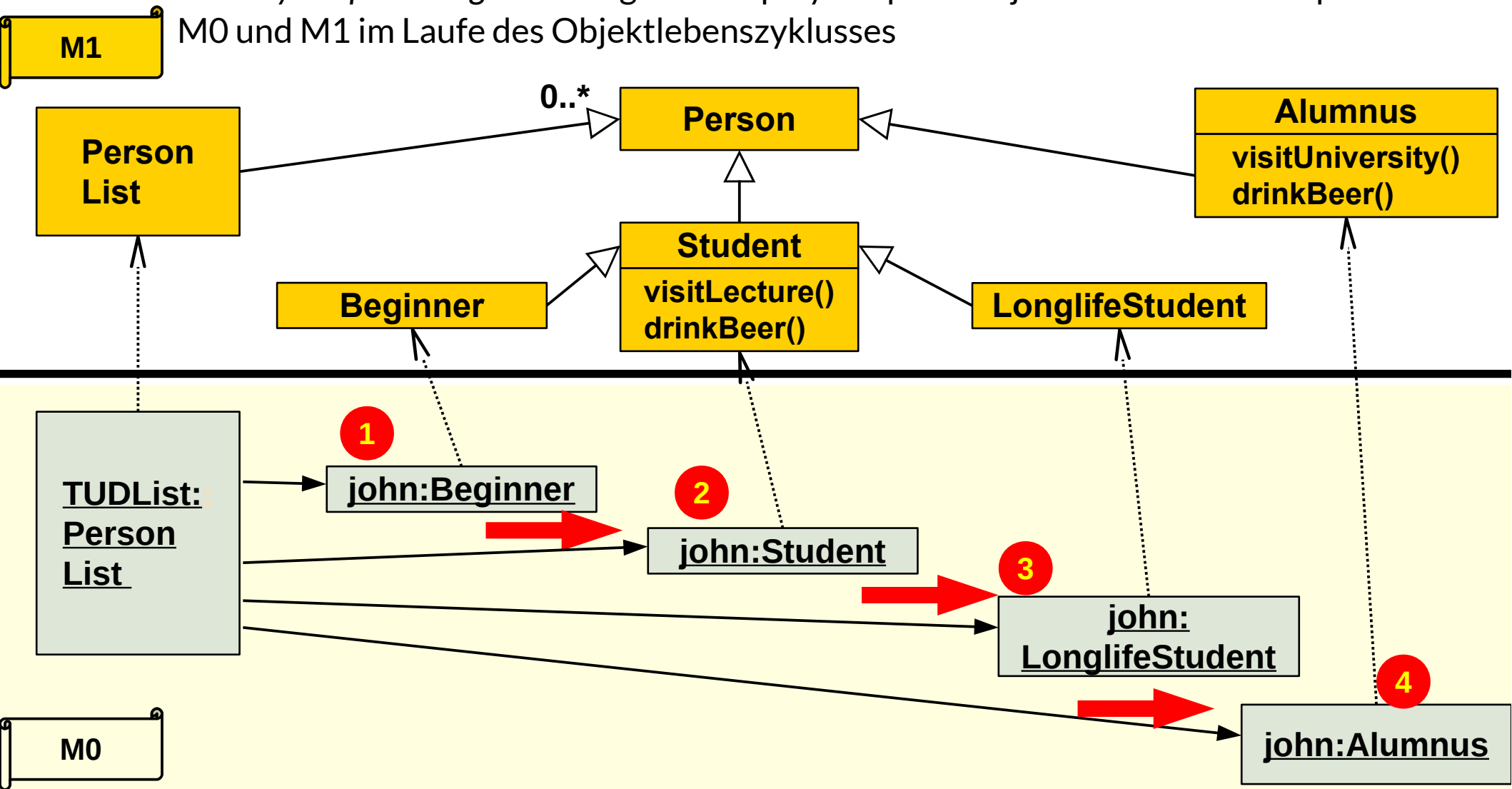
Beispiel: Termin-Klasse und Termin-Objekte

- Allgemeines Merkmal: Jeder Termin kann verschoben werden.
  - Daher schreibt die Klasse **Termin** vor, daß auf die Nachricht „verschiebeTermin“ reagiert werden muß.
- Unterklassen *spezialisieren* Oberklassen; Oberklassen *generalisieren* Unterklassen



Polymorphie (polymorphism) (im Polymorphie-Diagramm)

- Zur Laufzeit kann jedes Objekt einer Unterklasse ein Objekt einer Oberklasse *vertreten*. Das Objekt der Oberklasse ist damit *vielgestaltig (polymorphic)*.
- Ein *Polymorphie-Diagramm* zeigt für ein polymorphes Objekt das Zusammenspiel von M0 und M1 im Laufe des Objektlebenszyklusses



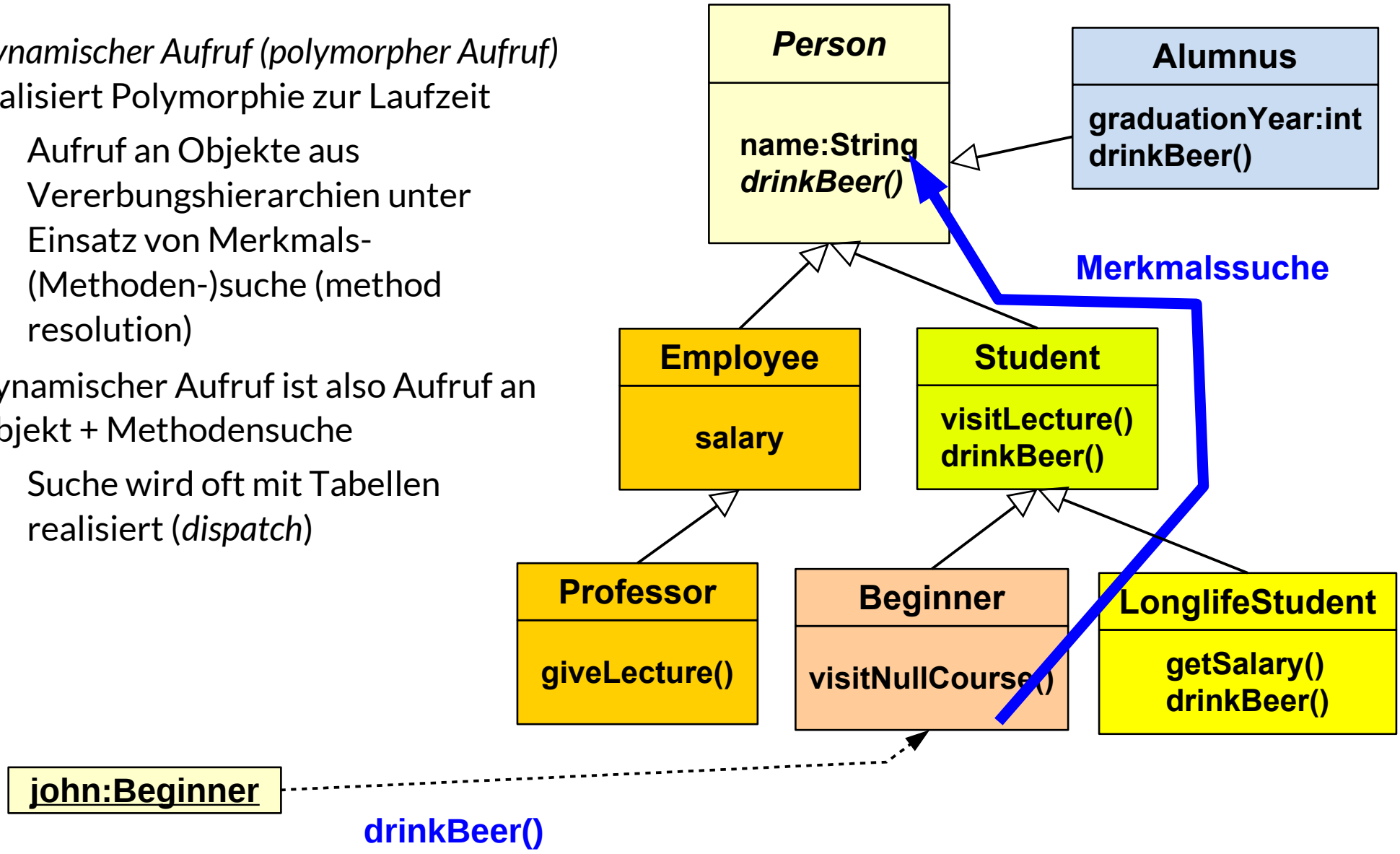
Wechsel der Gestalt (Objektevolution und Polymorphie)

- Die genaue Unterklasse einer Variablen wird festgestellt
  - Beim **Erzeugen** (der Allokation) des Objekts (Allokationszeit, oft in der Aufbauphase des Objektnetzes), oft in einem alternativen Zweig des Programms alternativ festgelegt
  - Bei einer **neuen Zuweisung** (oft in einer Umbauphase des Objektnetzes)

```
Person john;

if (hasLeftUniversity)
  john = new Alumnus();
else
  john = new Student();

// which type has Person john here?
....

if (hasWorkedHardLongEnough)
  john = new Professor();
// which type has person here?
// how will the person act?
john.visitLecture();
john.drinkBeer();
```

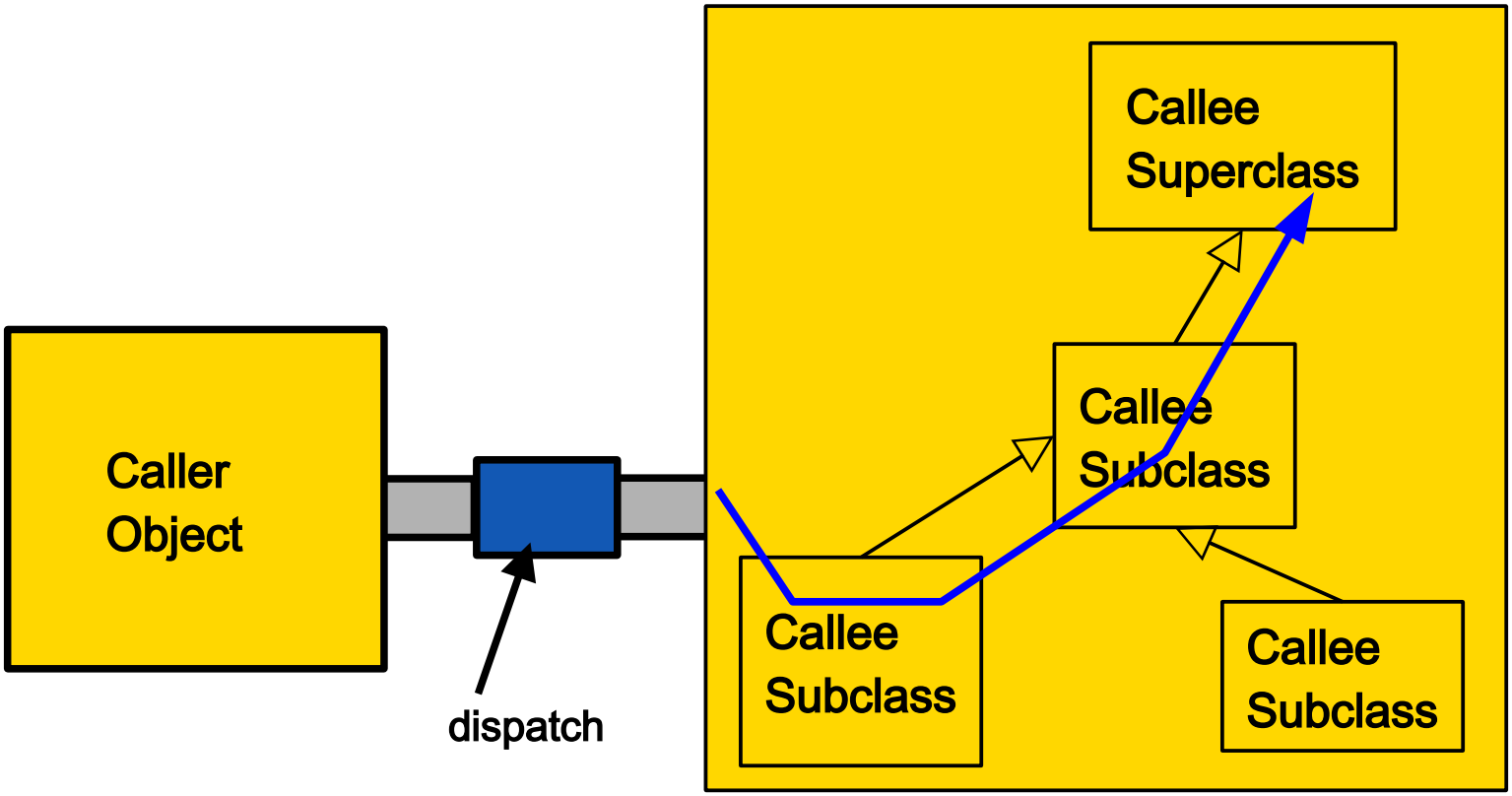
Dynamischer Aufruf (Dynamic dispatch)

- *Dynamischer Aufruf (polymorpher Aufruf)* realisiert Polymorphie zur Laufzeit
  - Aufruf an Objekte aus Vererbungshierarchien unter Einsatz von Merkmals-(Methoden-)suche (method resolution)
- Dynamischer Aufruf ist also Aufruf an Objekt + Methodensuche
  - Suche wird oft mit Tabellen realisiert (*dispatch*)



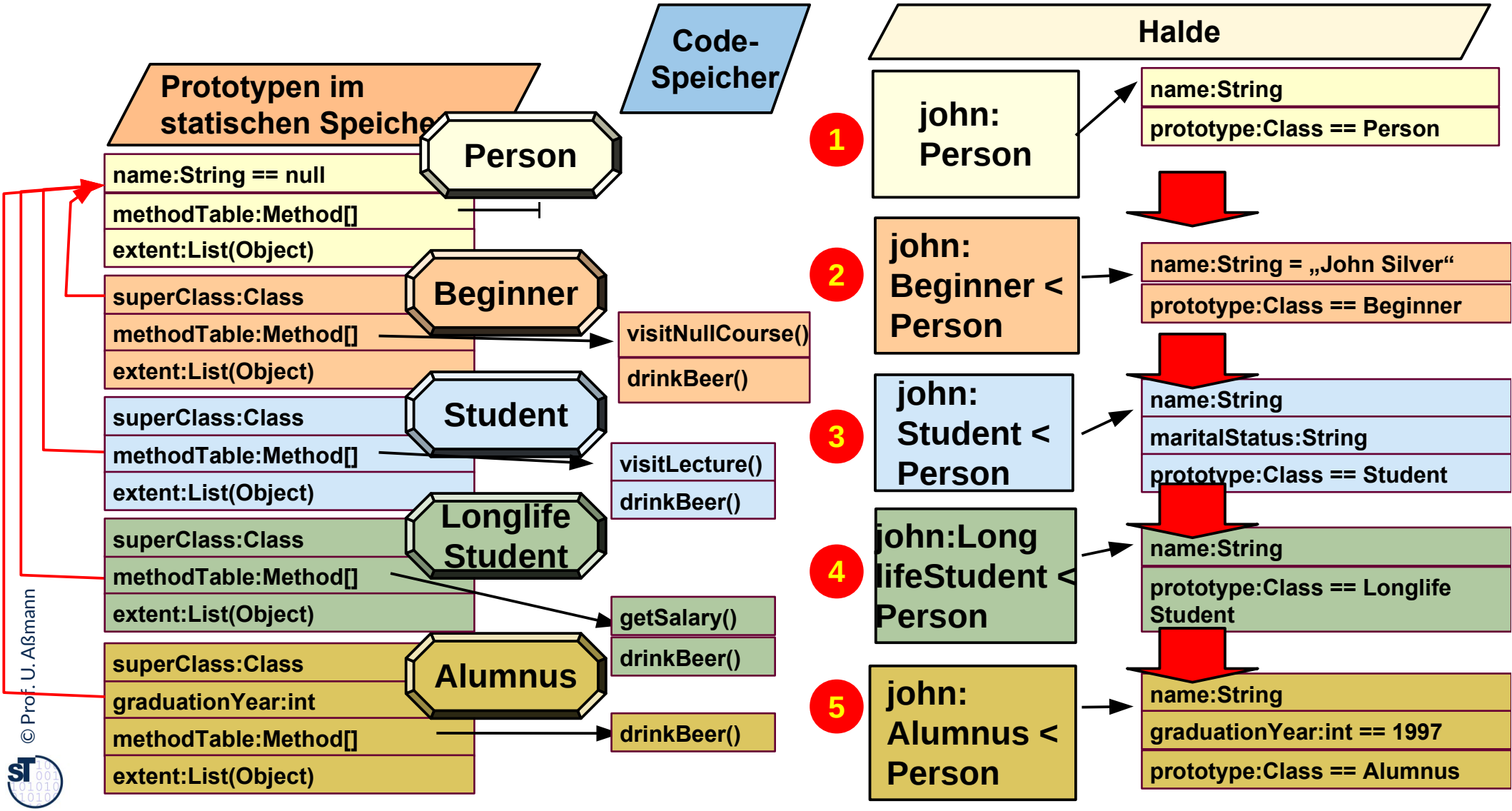
Dynamischer Aufruf (Dynamic Dispatch)

- Vom Aufrufer aus wird ein Suchalgorithmus gestartet, der die Vererbungshierarchie aufwärts läuft, um die rechte Methode zu finden
  - Die Suche läuft tatsächlich über die Klassenprototypen
  - Diese Suche kann teuer sein und muß vom Übersetzer optimiert werden (dispatch optimization)



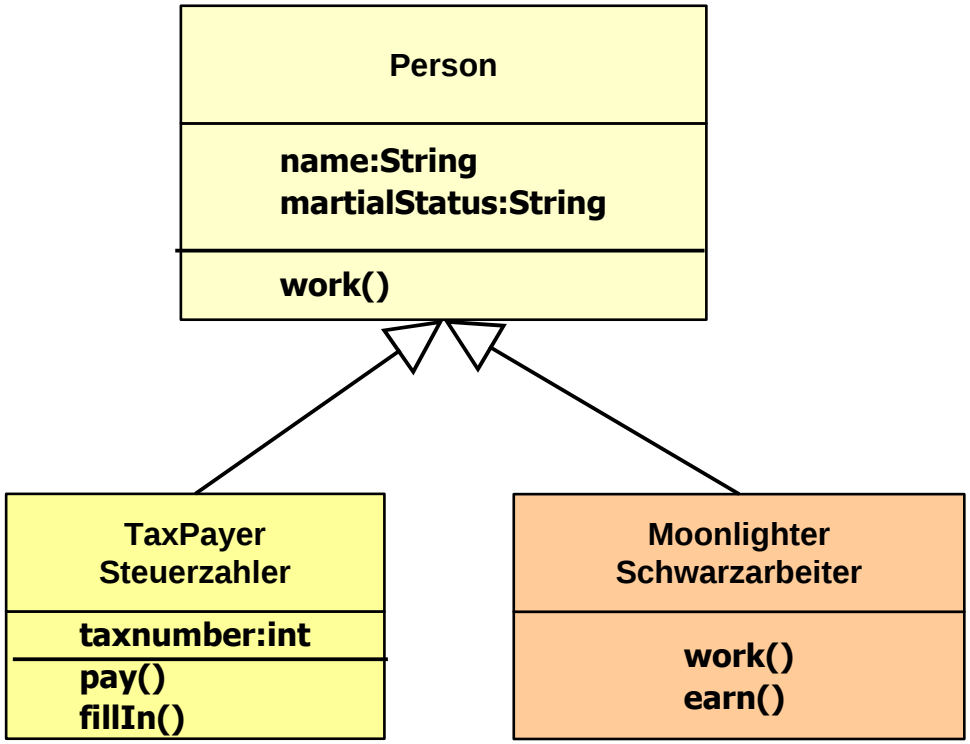
Was passiert beim polymorphen Aufruf im Speicher?

- Frage: Welche Inkarnation der Methode *drinkBeer()* wird zu den verschiedenen Zeitpunkten im Leben johns aufgerufen?



Polymorphe und monomorphe Methoden

- Methoden, die nicht mit einer Oberklasse geteilt werden, können nicht polymorph sein
- Die Adresse einer solchen *monomorphen* Methode im Speicher kann statisch, d.h., vom Übersetzer ermittelt werden. Eine Merkmalssuche ist dann zur Laufzeit nicht nötig
- Frage: Welche der folgenden Methoden sind poly-, welche monomorph?



11.4. Generische Klassen (Klassenschablonen, Template-Klassen, Parametrische Klassen)

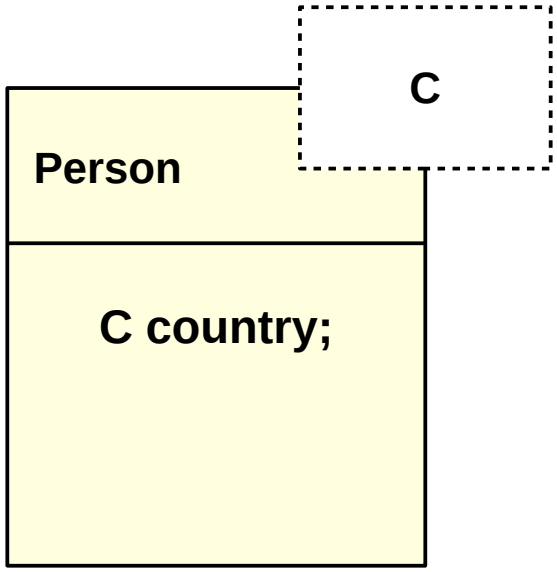
… bieten eine weitere Art, mit Löchern zu programmieren, um Vorgaben zu machen

- Generische Klassen lassen den Typ von einigen Attributen und Referenzen offen (“generisch”)
- Sie ermöglichen typisierte Wiederverwendung von Code

Generische Klassen

Eine generische (parametrische, Template-) Klasse ist eine Klassenschablone, die mit einem oder mehreren Typparametern (für Attribute, Referenzen, Methodenparameter) versehen ist.

- In UML



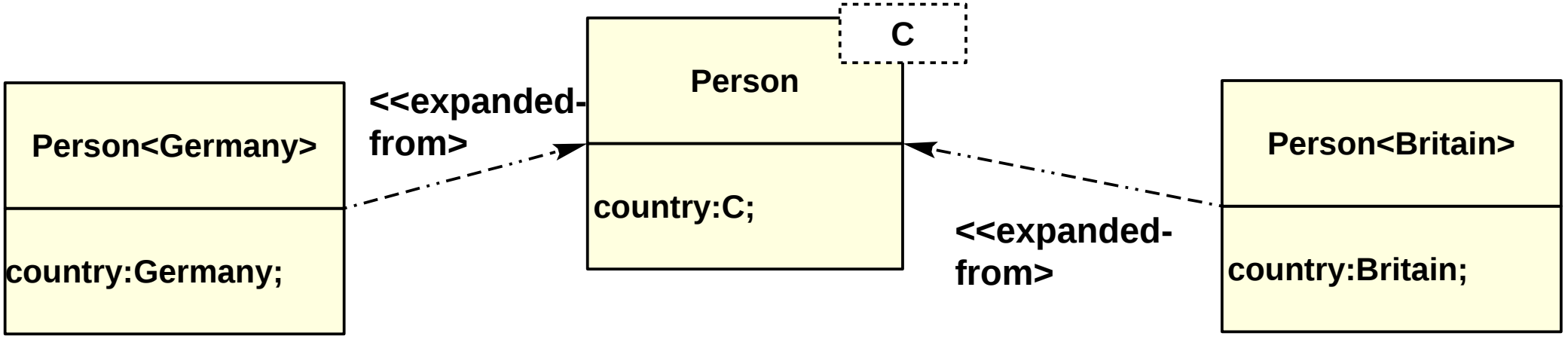
- In Java
  - Sprachregelung: “Person of C”

```
// Definition of a generic class
class Person<C> {
  C country;
}
```

```
/* Type definition using a generic type */
Person<Germany> egon;
Person<Britain> john;
```

Feinere statische Typüberprüfung für Attribute

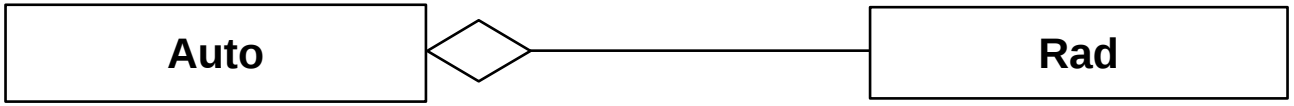
- Zwei Attributtypen, die durch Parameterisierung aus einer generischen Klassenschablone entstanden sind, sind nicht miteinander kompatibel
- Der Übersetzer entdeckt den Fehler (**statische Typprüfung**)




```
/* Type definition and initialization with object */
Person<Germany> egon = new Person<Germany>;
Person<Britain> john = new Person<Britain>;

/* Checks of assignments can use the improved typing */
john = egon;
```

Einsatzzweck: Typsichere Aggregation (has-a)

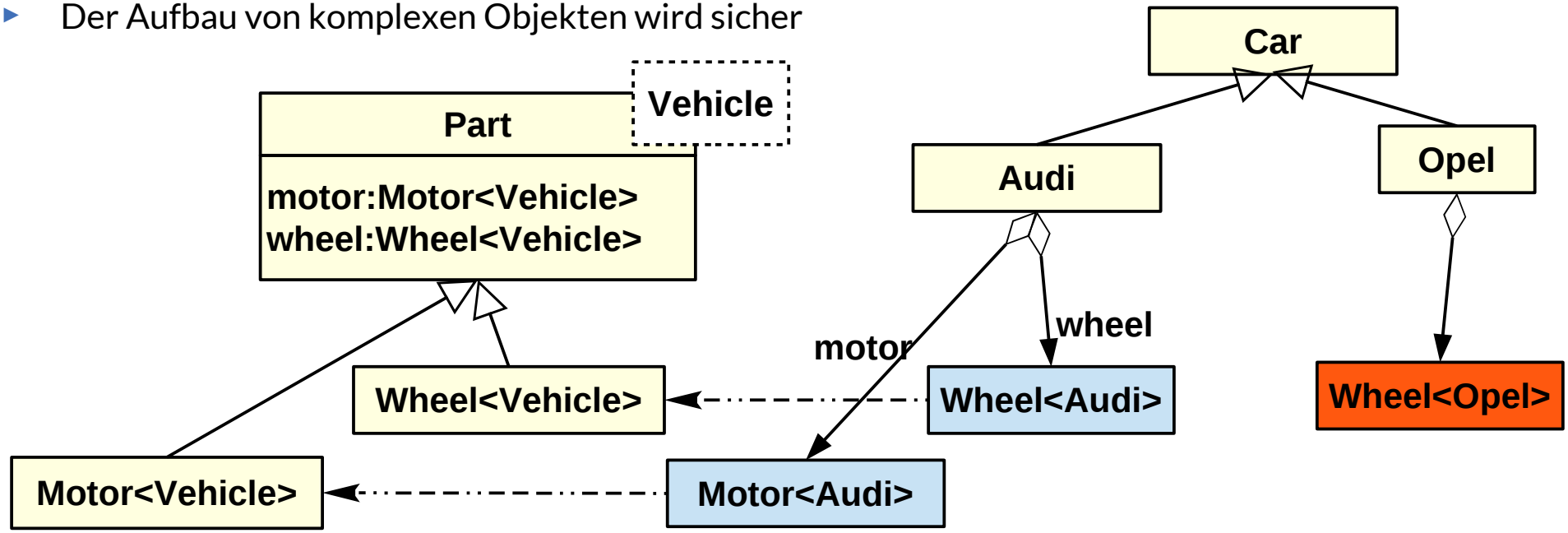
- **Definition:** Wenn eine Assoziation den Namen „hat-ein“ oder "besteht-aus" tragen könnte, handelt es sich um eine **Aggregation** (Ganzes/Teile-Relation).
  - Eine Aggregation besteht zwischen einem *Aggregat*, dem *Ganzen*, und seinen *Teilen*.
  - Die auftretenden Aggregationen bilden auf den Objekten immer eine transitive, antisymmetrische Relation (einen gerichteten zyklenfreien Graphen, *dag*).
  - Ein Teil kann zu mehreren Ganzen gehören (*shared*), zu einem Ganzen (*owns-a*) und exklusiv zu einem Ganzen (*exclusively-owns-a*)



Lies: „Auto *hat ein* Rad“

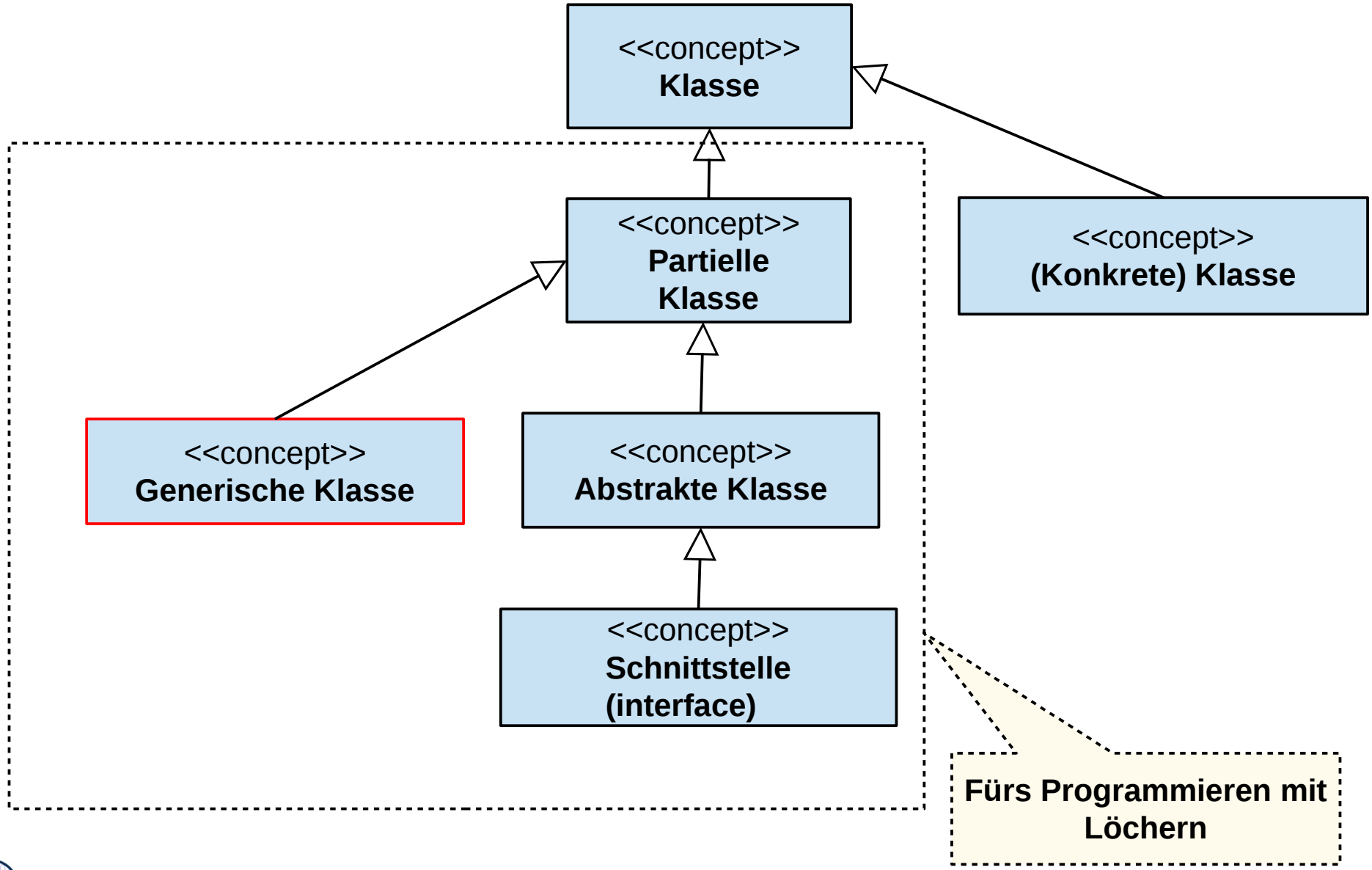
Einsatzzweck: Typsichere Aggregation (has-a)

- Generische Klassen können Typen für Ganz-Teile-Beziehungen vorgeben
- Der Aufbau von komplexen Objekten wird sicher




```
/* Type definition and initialization with object */
Motor<Audi> motorOfAudi = new Motor<Audi>;
Wheel<Audi> wheelOfAudi = new Wheel<Audi>;
Wheel<Opel> wheelOfOpel = new Wheel<Opel>;

/* Checks of assignments can use the improved typing */
audi = new Audi();
audi.motor = motorOfAudi;
audi.wheel = wheelOfOpel;
```

Q2: Begriffshierarchie von Klassen (Erweiterung)



Was haben wir gelernt?

- Codeverschmutzung wird vermieden durch Vererbung
  - Vererbung erlaubt die *Wiederverwendung* von Merkmalen aus Oberklassen,
  - Einfache Vererbung führt zu Vererbungshierarchien
- Schnittstellen als auch abstrakte Klassen erlauben es, Anwendungsprogrammierern Struktur vorzugeben
  - Sie definieren “Haken”, in die Unterklassen konkrete Implementierungen schieben
  - Schnittstellen sind vollständig abstrakte Klassen
- Polymorphie erlaubt dynamische Architekturen
  - Merkmalssuche (dynamic dispatch) löst die Bedeutung von Merkmalsnamen auf, in dem von den gegebenen Unterklassen aus aufwärts gesucht wird
  - Polymorphie benutzt Merkmalssuche, um die Mehrdeutigkeit von Namen in einer Vererbungshierarchie aufzulösen
  - Monomorphe Aufrufe sind schneller, weil Merkmalssuche eingespart werden kann
- Die Klasse `Object` enthält als implizite Oberklasse der Java-Bibliothek gemeinsam nutzbare Funktionalität für alle Java-Klassen
- Generische Klassen ermöglichen typsichere Wiederverwendung von Code über Typ-Parameter der Compiler meldet mehr Fehler

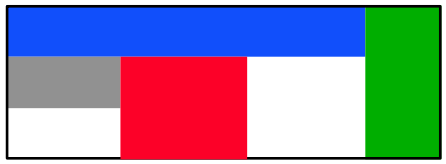
Warum ist das wichtig?

- Wiederverwendung ist eines der Hauptprobleme des Software Engineering
  - In einem Programm
  - Von Projekt zu Projekt
  - Von Produkt zu Produkt (Produktfamilien, Produktlinien)
- Vererbung, generische Klassen und Rollenklassen können Code-Replikate und Code-Explosion weitgehend vermeiden
  - Der Test von Software wird systematisiert
- Wiederverwendung ist das Hauptmittel der Softwarefirmen, um profitabel arbeiten zu können:
  - Schreibe und teste einmal und wiederverwende oft
  - Alle erfolgreichen Geschäftsmodelle von Softwarefirmen basieren auf Wiederverwendung (□ Produktlinien, □ Produktmatrix)
  - Ohne Wiederverwendung kein Verdienst und Überleben als Softwarefirma
- Firmen, die Wiederverwendung beherrschen, können neue Produkte sehr schnell erzeugen (reduction of time-to-market)
  - und sich an wechselnde Märkte gut anpassen
- Firmen mit guter Wiederverwendungstechnologie leben länger

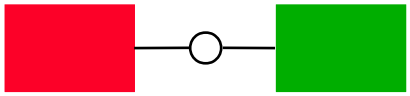
Verständnisfragen

- Geben Sie eine Begriffshierarchie der Methodenarten an. Welche könnten Sie sich noch denken?
- Geben Sie eine Begriffshierarchie des Klassenbegriffs an. Welche Klassenarten kennen Sie? Wie spezialisieren sie sich?
- Erweitern Sie die Vererbungshierarchie der Universitätsangehörigen um den Rektor und den Pedell (s. Wikipedia). Wo müssen sie eingeordnet werden?
- Stellen Sie ein Polymorphie-Diagramm überdie Phasen Ihres Lebens auf.
- Welchen Polymorphie-Zyklus durchläuft der Steuerzahler unseres Beispiels?
- Kann eine Steuererklärung polymorph sein?
- Wie würden Sie ein Testprogramm für ein polymorphes Objekt aus einem Polymorphie-Diagramm heraus entwickeln?

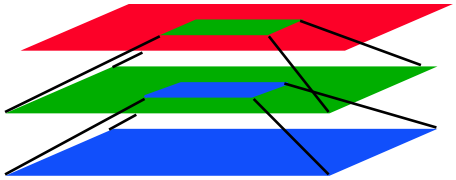
Prinzipielle Vorteile von Objektorientierung



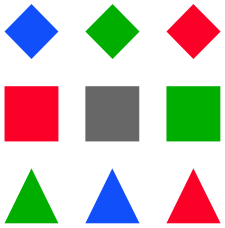
Zuständigkeitsbereiche



Klare Schnittstellen



Hierarchie



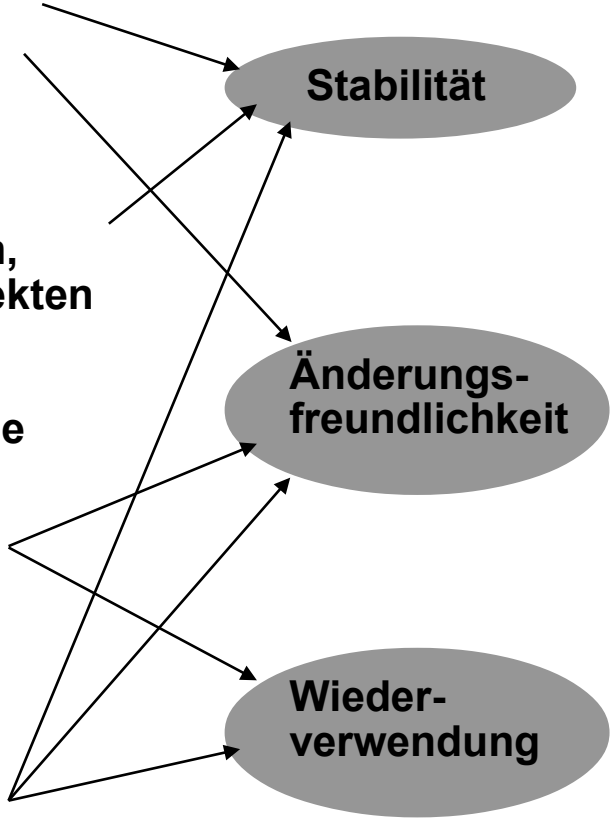
Baukastenprinzip

Lokalität: Lokale Kapselung von Daten und Operationen, gekapselter Zustand

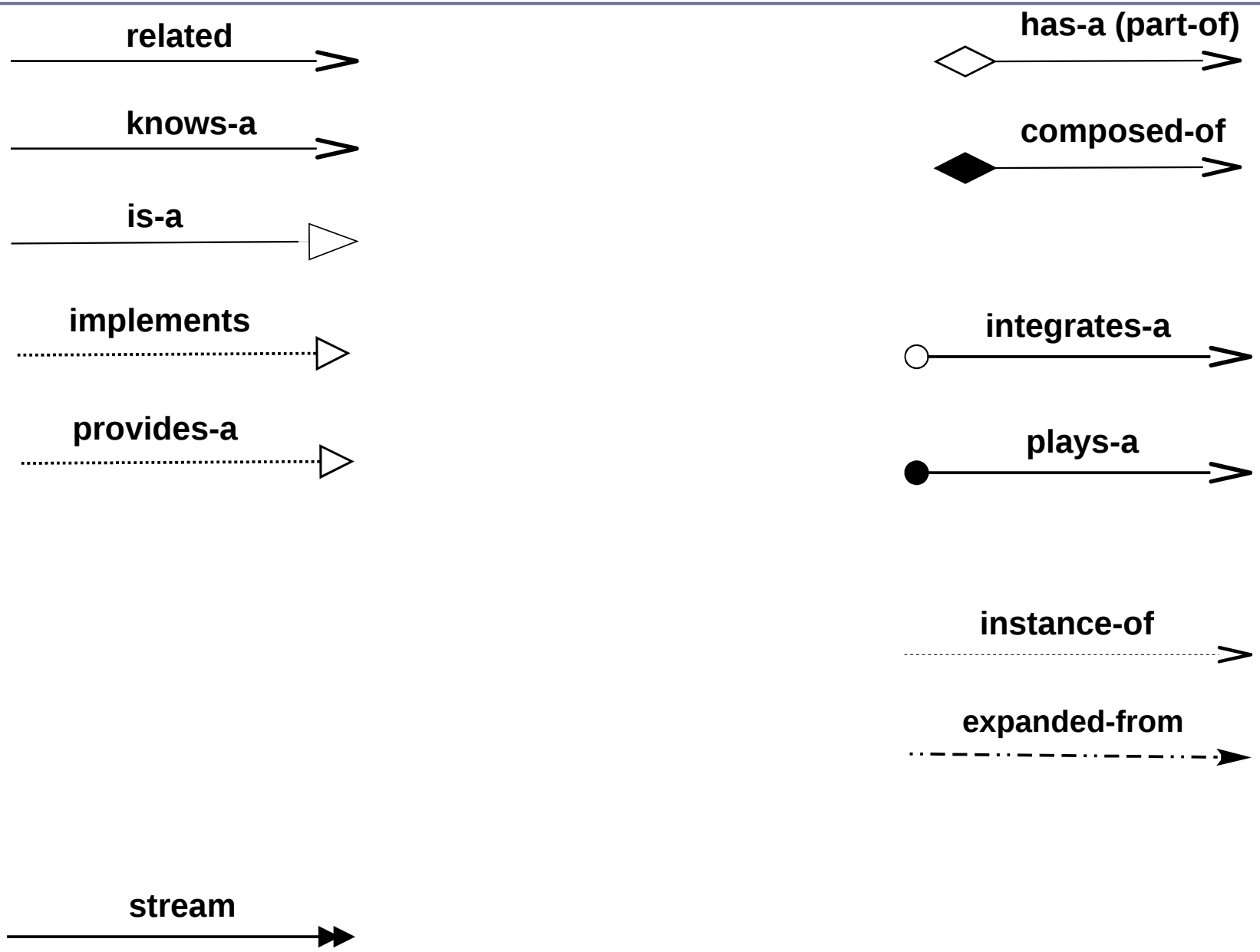
Typen und Typsicherheit Definiertes Objektverhalten, Nachrichten zwischen Objekten

Vererbung und Polymorphie (Spezialisierung), Wiederverwendung Klassenschachtelung

Benutzung vorgefertigter Klassenbibliotheken (Frameworks), Anpassung durch Spezialisierung (Vererbung)

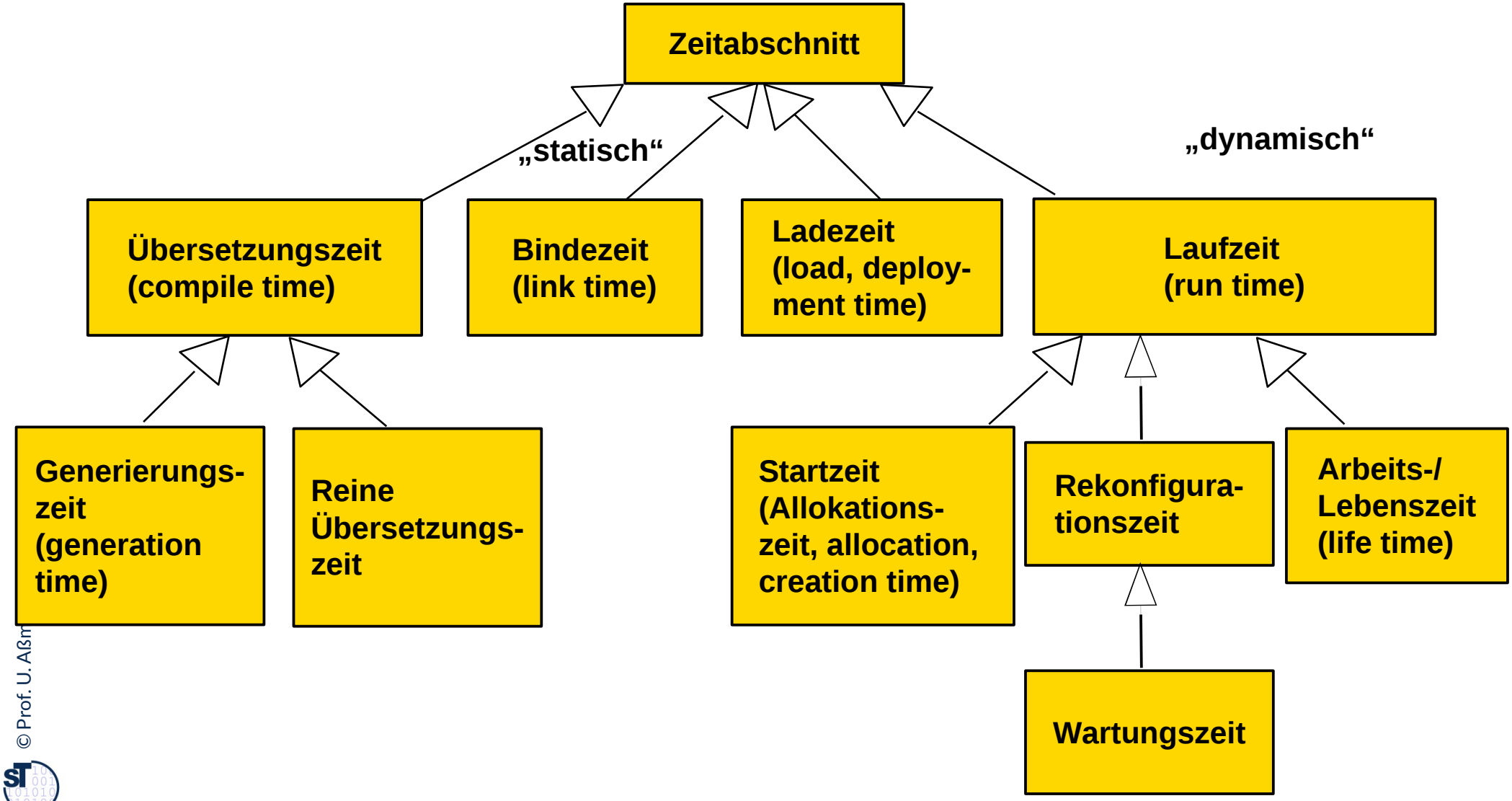


Q10: Relationen



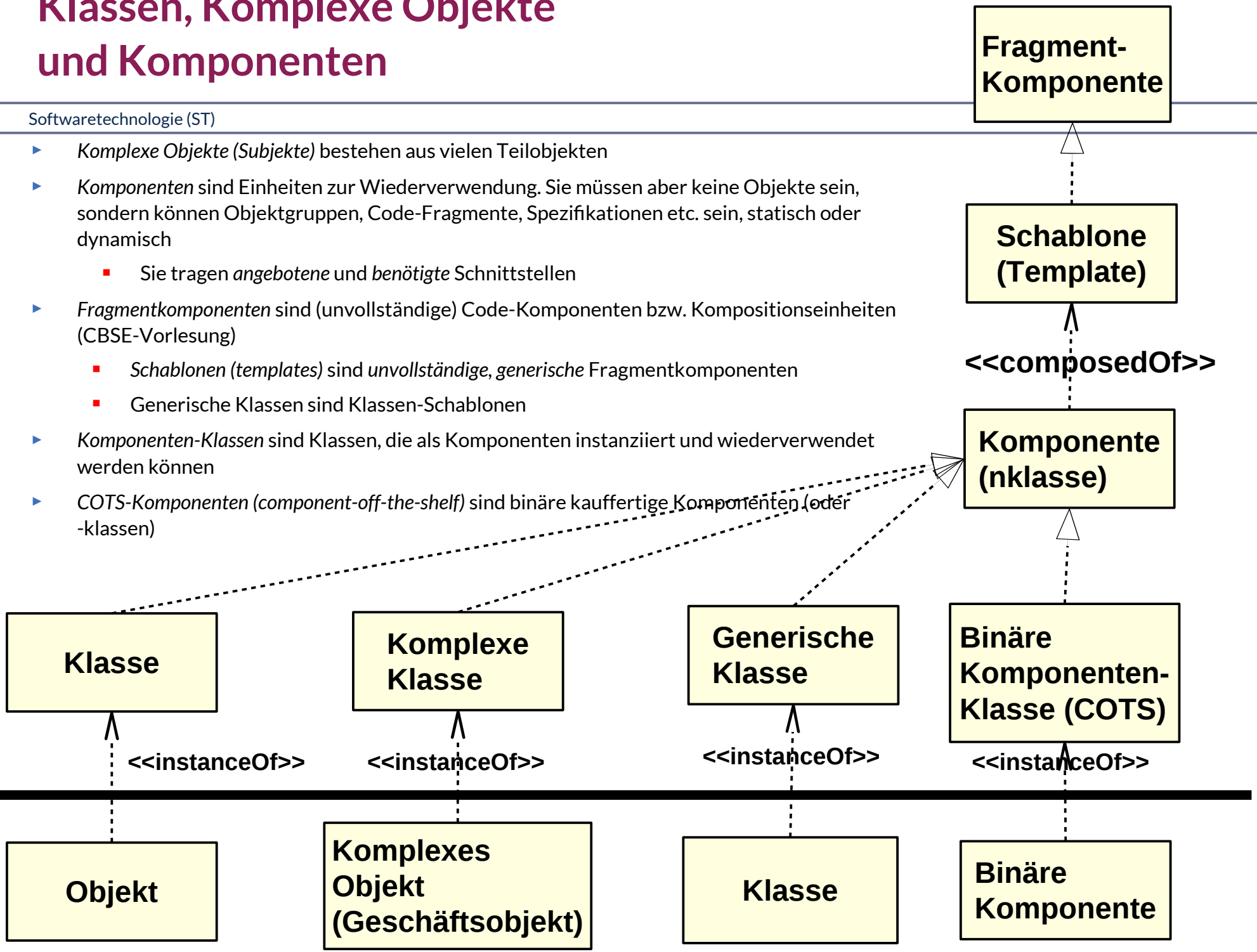
Bsp. Begriffshierarchie: Verschiedene Zeiten im Lebenszyklus einer Software

- Übersetzungszeit kennt hauptsächlich *Klassen*; Laufzeit kennt hauptsächlich *Objekte*



Klassen, Komplexe Objekte und Komponenten

- *Komplexe Objekte (Subjekte)* bestehen aus vielen Teilobjekten
- *Komponenten* sind Einheiten zur Wiederverwendung. Sie müssen aber keine Objekte sein, sondern können Objektgruppen, Code-Fragmente, Spezifikationen etc. sein, statisch oder dynamisch
  - Sie tragen *angebotene* und *benötigte* Schnittstellen
- *Fragmentkomponenten* sind (unvollständige) Code-Komponenten bzw. Kompositionseinheiten (CBSE-Vorlesung)
  - *Schablonen (templates)* sind *unvollständige, generische* Fragmentkomponenten
  - Generische Klassen sind Klassen-Schablonen
- *Komponenten-Klassen* sind Klassen, die als Komponenten instanziiert und wiederverwendet werden können
- *COTS-Komponenten (component-off-the-shelf)* sind binäre kauffertige Komponenten (oder -klassen)

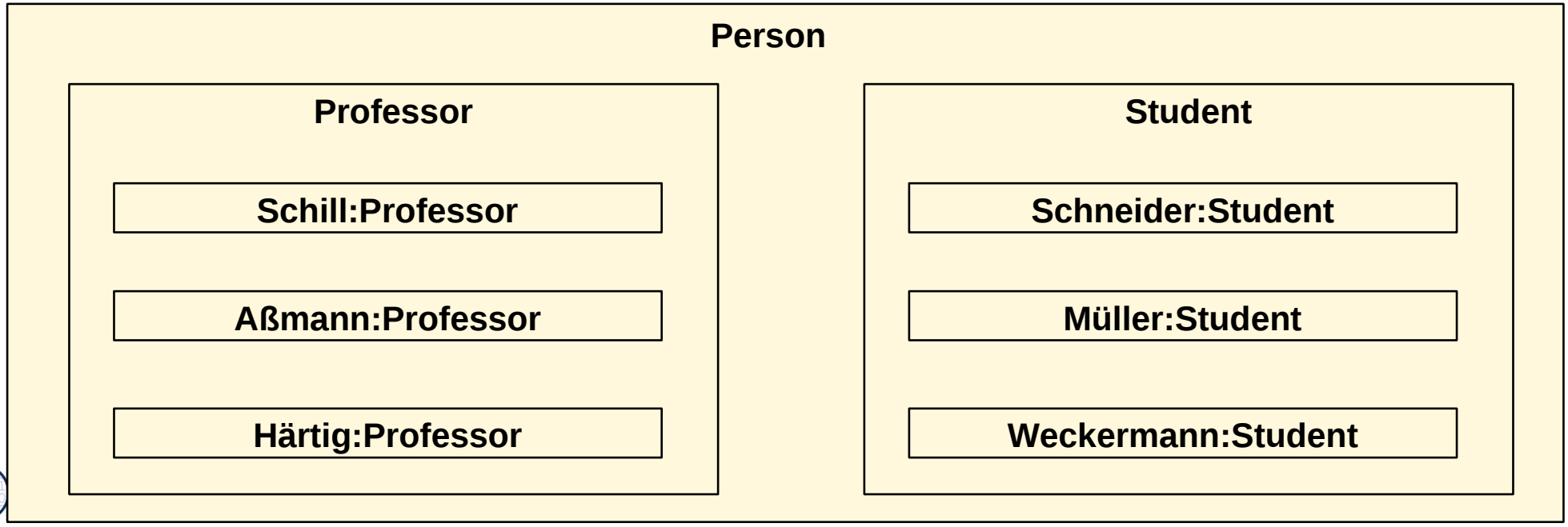


Bemerkung

- Wir benutzen i.A. mehrere Darstellungen für Klassen- und Objektdiagramme
  - UML-Strukturdiagramme
  - Tripel-Sätze: SPO mit infix-Prädikaten
  - Venn-Diagramme (mengenorientierte Sicht)
- Venn-Diagramme
  - Sie können sowohl die Zugehörigkeit eines Objekts zu einer Klasse als auch Vererbung zwischen Klassen mit einem *Einkapselung* einheitlich beschreiben (mengentheoretischer und Ähnlichkeitsaspekt)
  - Sie können sowohl die statischen Beziehungen von Klassen, als auch die dynamischen Beziehungen von Objekten und Klassen beschreiben
  - Sie werden oft für die Spezifikation von Begriffshierarchien (Taxonomien, Ontologien) verwendet

Beispiel als Venn-Diagramm

- Vererbung kann auch durch Enthaltensein von Rechtecken ausgedrückt werden
  - Einfach-Vererbung ergibt ein Diagramm ohne Überschneidungen
- Für Objekt-Extents einer Klassenhierarchie gilt, dass jede Klasse die eigenen Objekte verwaltet
  - Der Objekt-Extent einer Oberklasse ergibt sich aus der Vereinigung der Extents aller Unterklassen (Person aus Professor und Student). Genau das drückt das Venn-Diagramm aus



TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik - Institut Software- und Multimediatechnik - Softwaretechnologie

11. Vererbung und Polymorphie
Die Filter gegen Codeverschmutzung

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 17-0.3, 07.04.17

1) Vererbung zwischen Klassen
2) Abstrakte Klassen und Schnittstellen
3) Polymorphie
4) Generische Klassen

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Keep your code clean, take your stance:
Use single inheritance.
Instead copy, move it up
Superclass will be its hub.
Breathe into your objects life -
Let them polymorphy drive.
Abstract classes, interface
Shape their childrens' pretty face.

Begleitende Literatur

- Das **Vorlesungsbuch** von Pearson: **Softwaretechnologie für Einsteiger.** Vorlesungsunterlage für die Veranstaltungen an der TU Dresden. Pearson Studium, 2014. Enthält ausgewählte Kapitel aus:
  - UML: Harald Störrle. UML für Studenten. Pearson 2005. Kompakte Einführung in UML 2.0.
  - Softwaretechnologie allgemein: W. Zuser, T. Grechenig, M. Köhle. Software Engineering mit UML und dem Unified Process. Pearson.
  - Bernd Brügge, Alan H. Dutoit. Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java. Pearson Studium/Prentice Hall.
  - Erhältlich bei Thalia (Nürnberger Platz), Thalia (Haus des Buches)
- Noch ein sehr gutes, umfassend mit Beispielen ausgestattetes Java-Buch:
  - C. Heinisch, F. Müller, J. Goll. Java als erste Programmiersprache. Vom Einsteiger zum Profi. Teubner.
- Für alle, die sich nicht durch Englisch abschrecken lassen:
- Safari Books, von unserer Bibliothek SLUB gemietet:
  - http://proquest.tech.safaribooksonline.de/
- Free Books: http://it-ebooks.info/
  - Kathy Sierra, Bert Bates: Head-First Java http://it-ebooks.info/book/255/

Safari Tech Books Online verfügbar.

•Katalog der SLUB (www.slub-dresden.de) verzeichnet ca. 30.000 elektronische Bücher mit Bezug zur Informatik, die online gelesen werden können

•http://proquest.tech.safaribooksonline.de/

Obligatorische Literatur

- ST für Einsteiger Kap. 9, Teil II (Störrle, Kap. 5.2.6, 5.6)
- Zuser Kap 7, Anhang A
- Java
  - http://docs.oracle.com/javase/tutorial/java/index.html is the official Oracle tutorial on Java classes
  - Balzert LE 9-10
  - Boles Kap. 7, 9, 11, 12

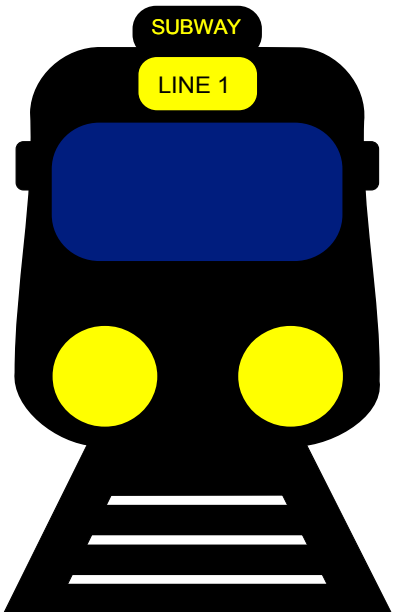
"Objektorientierte Softwareentwicklung"
Hörsaalübung
Fr, 13:00 HSZ 03, Dr. Demuth

Ziele

- Elementare Techniken der Wiederverwendung von objektorientierten Programmen kennen
  - Einfache Vererbung zwischen Klassen, konzeptuell und im Speicher
  - Abstrakte Klassen und Schnittstellen verstehen
  - Merkmalssuche in einer Klasse und in der Vererbungshierarchie aufwärts nachvollziehen können
  - Überschreiben von Merkmalen verstehen
  - Generische Typen zur Vermeidung von Fehlern
- Dynamische Architektur eines objektorientierten Programms verstehen
  - Polymorphie im Speicher verstehen
  - Objekte vs. Rollen verstehen

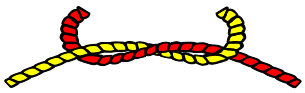
Java Herunterladen

- Das Java Development Kit (JDK) 8
- www.javasoft.com
- http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html



If you have not yet downloaded Java, and started the compiler and the VM, you are already rather late and in danger to miss the train.

Problem: Was tut man gegen Codeverschmutzung bzw. Copy-And-Paste-Programming (CAPP)?



Codeverschmutzung durch CAPP: Nach einer Weile entdeckt man in einem gewachsenen System, dass jede Menge Code repliziert wurde
Große Software kann 10-20% an *Replikaten (code clones)* enthalten *(Code-Explosion, code bloat)*

Plagiat
Ignoranz
Aufwandsreduktion
Mangelnde Anforderungsanalyse der Anwendungsdomäne

Aufwandsreduktion:
Wiederverwendung von Tests

- http://c2.com/cgi/wiki?CopyAndPasteProgramming
- http://en.wikipedia.org/wiki/Copy_and_paste_programming

•Gründe:

•**Plagiat:** Code-Diebstahl führt oft zu Prozessen

•**Ignoranz**: Code wird nicht verstanden, sondern aus einem funktionierenden Modul eines Dritten kopiert wie stabil ist der Code wirklich?

•**Aufwandsreduktion** für den einzelnen Programmierer, um den Unterschied Programm-Software zu nutzen:

- *Getesteter Code (Software)* wird wiederverwendet, weil es zu aufwändig wäre, neue Tests für eigengeschriebenen Code zu entwickeln
- Problem: man vergisst, was Replikate waren und muss sie alle separat testen

•**Mangelnde Anforderungsanalyse** der Anwendungsdomäne: Die gemeinsamen Eigenschaften von Domänenklassen wurden nicht herausgefunden

- und nicht in gemeinsam genutzte Klassen ausfaktorisiert

Linking Replicates

- Interessante Technik, Code-Replikate zu finden und dauerhaft zu verlinken:
  - Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In VL/HCC, pages 173-180. IEEE Computer Society, 2004.
  - http://harmonia.cs.berkeley.edu/papers/toomim-linked-editing.pdf
- Optional, mit vielen schönen Visualisierungen von Code Clones:
  - Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In WCRE, pages 100-109. IEEE Computer Society, 2004.
  - http://rmod.lille.inria.fr/archives/papers/Rieg04b-WCRE2004-ClonesVisualizationSCG.pdf

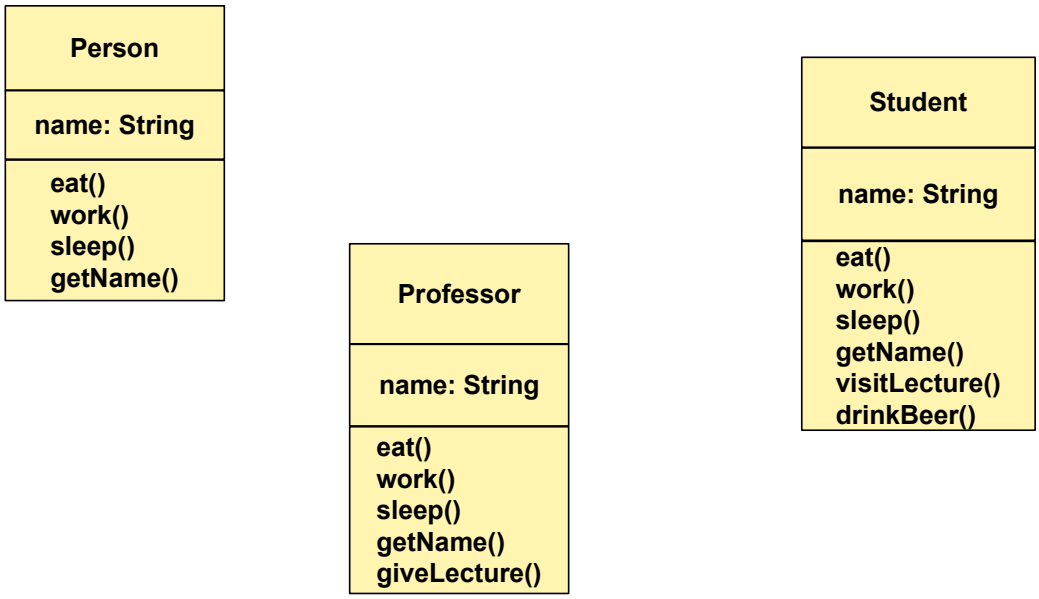
In der Vorlesung werden viele wissenschaftliche Papiere zur zusätzlichen Lektüre empfohlen. Sie sind dank der SLUB mit Lizenzen für viele elektronische Bibliotheken dieser Welt ausgestattet. Nutzen Sie sie!

11.1 Vererbung zwischen Klassen
beseitigt Codereplikate

Ähnlichkeit von Klassen sollten in Oberklassen ausfaktorisiert werden

Codeverschmutzung am Beispiel

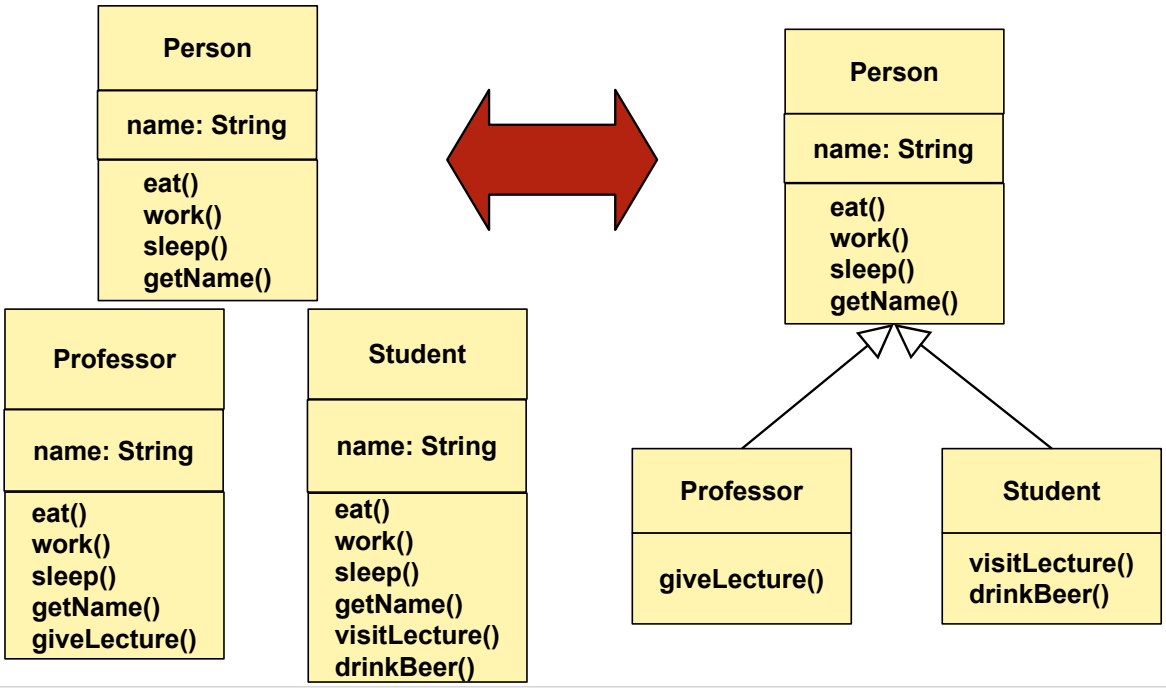
- **Hier:** Person wurde zu Professor und Student kopiert und danach erweitert



Wahlloses Hinzufügen von Attributen udn Methoden führt in eine Sackgasse, denn meist werden die “features” repliziert und Codeverschmutzung entsteht.

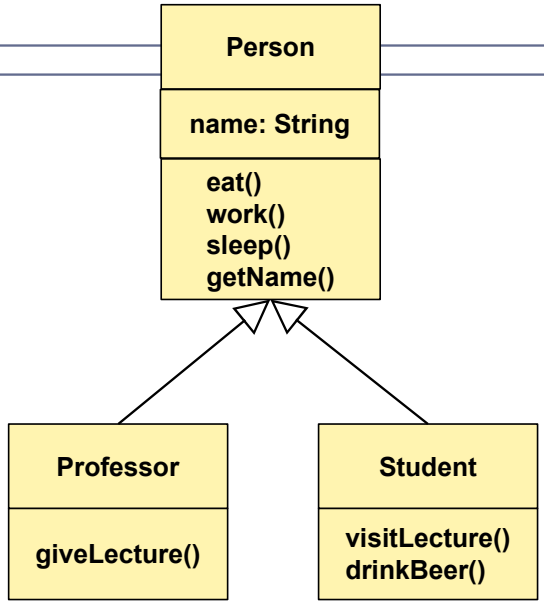
Einfache Vererbung

- **Vererbung**: Eine Klasse kann Merkmale von einer Oberklasse **erben**
- **Hier:** Oberklasse Person enthält alle gemeinsamen Merkmale der Unterklasse



Einfache Vererbung

- **Vorteil:** Vererbung drückt Gemeinsamkeiten aus
  - Die Unterklasse ist damit ähnlich zu dem Elter und den Geschwistern
  - Vererbung stellt *is-a*-Beziehung her (<)
- **Hier**: Oberklasse Person enthält alle gemeinsamen Merkmale der Unterklasse
- Bei **einfacher Vererbung** hat jede Klasse nur *eine* Oberklasse
  - Dann ist die Vererbungsrelation ein Baum




```
// Java
Professor extends Person {}; Student extends Person{};
```

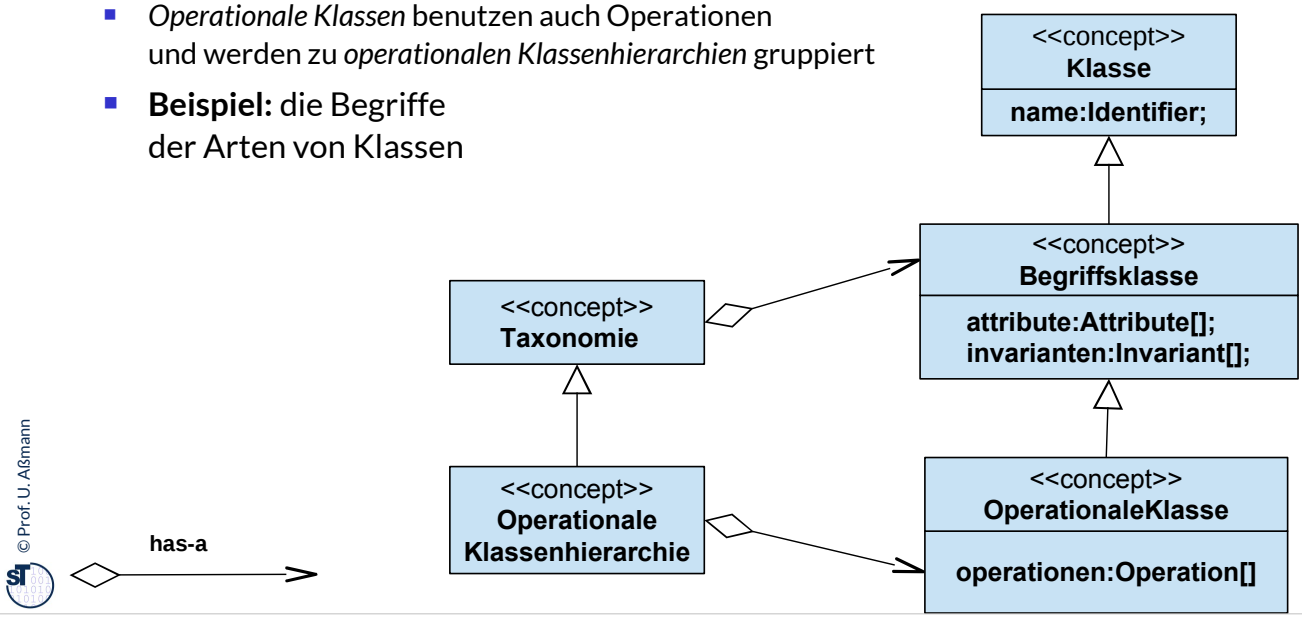

Einfache Vererbung ist relativ einfach zu verstehen, weil man nur eine Oberklasse hat, aus der Merkmale stammen können, also einem einzigen Pfad zur Wurzel der Vererbungshierarchie folgen muss.

In anderen Programmiersprachen wird u.U. andere Syntax für Vererbungsoperatoren benutzt:

```
// F-Prolog
Professor < Person. Student < Person.
// OWL
Professor is-a Person. Student is-a Person.
```

Q1: Begriffshierarchien (Taxonomien) nutzen einfache Vererbung

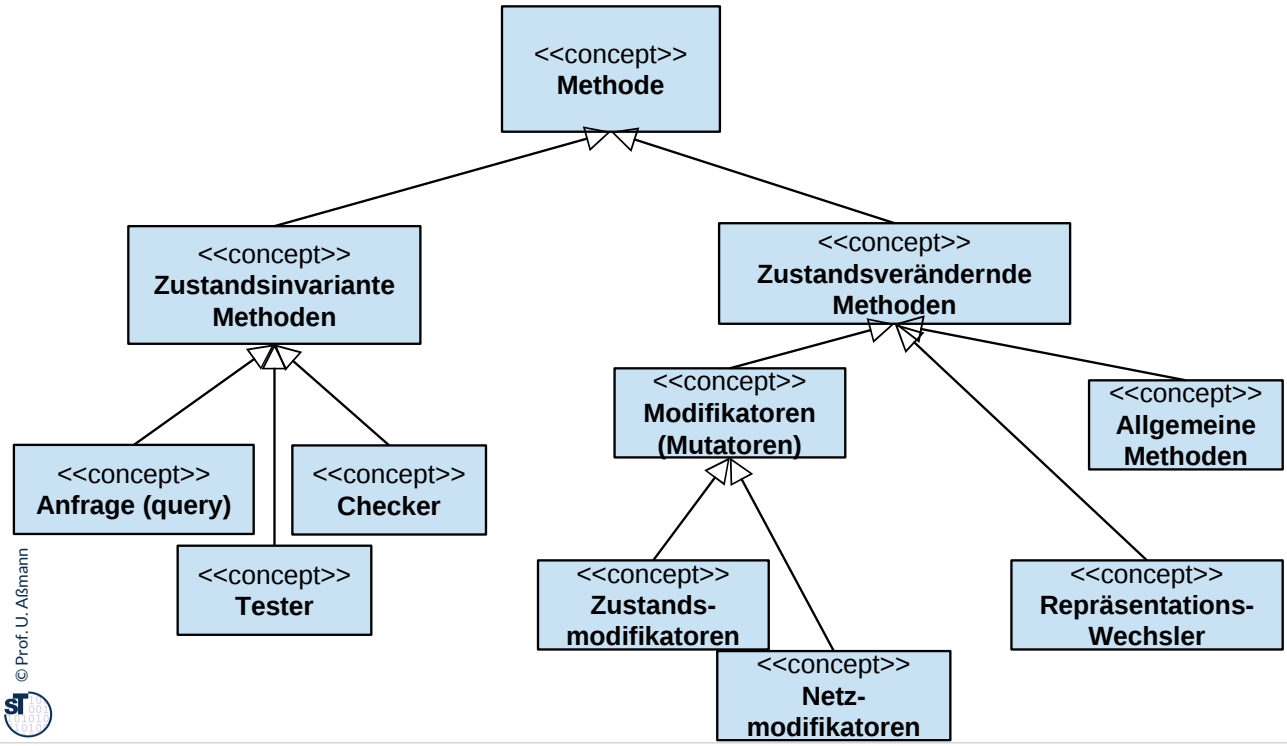
- Domänenmodelle werden durch *Klassifikation* der Domänenobjekte und Domänenkonzepte ermittelt
- Klassifikationen führen zu ***Begriffshierarchien (Taxonomien)***
  - *Begriffsklassen* besitzen nur Attribute und Invarianten (leicht blau)
- *Operationale Klassen* benutzen auch Operationen und werden zu *operationalen Klassenhierarchien* gruppiert
- **Beispiel:** die Begriffe der Arten von Klassen



Hier sehen wir eine Begriffshierarchie (Taxonomie) von Begriffen (Klassen ohne Instanzen).

Bsp: Begriffshierarchie (Taxonomie) der Methodenarten

- **Wiederholung**: Welche Arten von Methoden gibt es in einer Klasse?

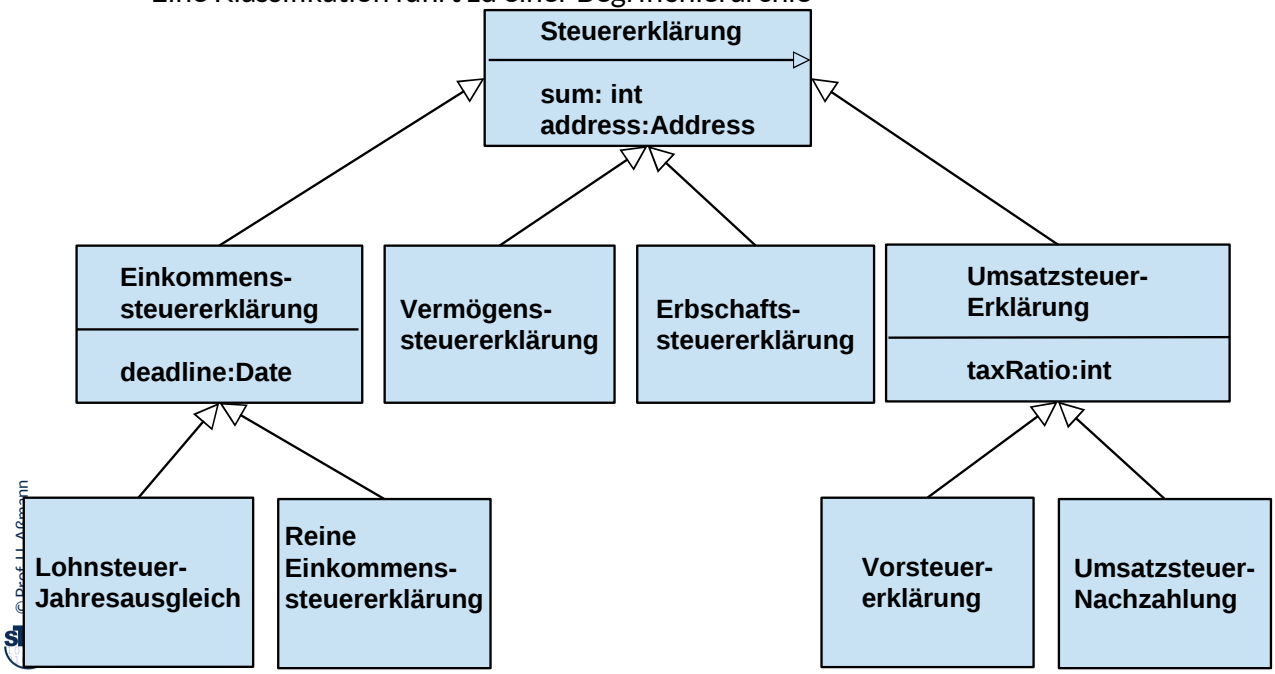


Dieser Vererbungsbaum zeigt eine Begriffshierarchie der Methoden, die wir in Java finden.

Die Klassifikation ist wichtig, weil die verschiedenen Methoden den Objektzustand ihres Objekts unterschiedlich beeinflussen.

Bsp. Taxonomie der Steuererklärungen

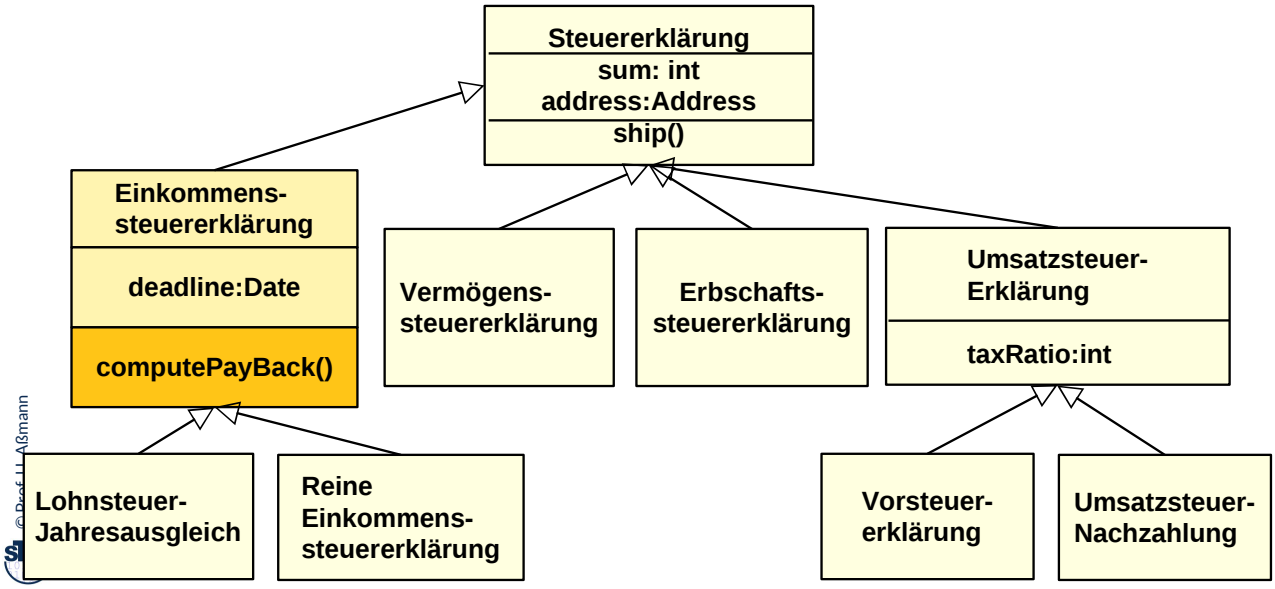
- Domäne: Finanzbuchhaltung
- Das deutsche Steuerrecht kennt viele Arten von Steuererklärungen
- Eine Klassifikation führt zu einer Begriffshierarchie



Deutsche Finanzämter erheben die Einkommenssteuer mit des ELSTER-programms (www.elster.de). Allerdings ist die Einkommenssteuererklärung einer Einzelperson nur eine von vielen Steuererklärungsarten. Freiberufler und Unternehmer müssen ggf. andere Arten ausfüllen.

Bsp. Erweiterung einer Begriffshierarchie hin zu operationalen Klassenhierarchie

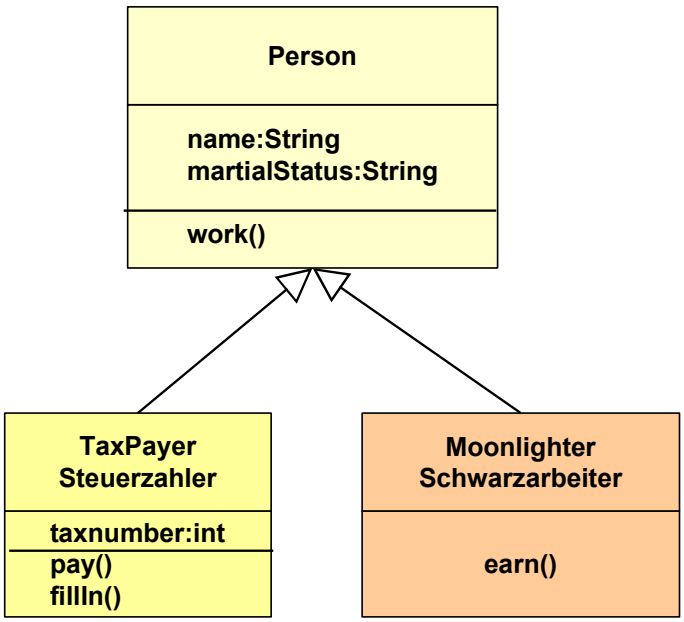
- Programmiert man eine Steuerberater-Software, muss man die Begriffshierarchie der Steuererklärungen als Klassen einsetzen.
- Daneben sind aber die Klassen um eine neue **Abteilung (compartment)** mit **Operationen** zu erweitern, denn innerhalb der Software müssen sie ja etwas tun.



Man kann eine Taxonomie in eine operationale Klassenhierarchie verwandeln, indem man Methoden-Signaturen hinzufügt.

11.1.2 Vererbung im Speicher

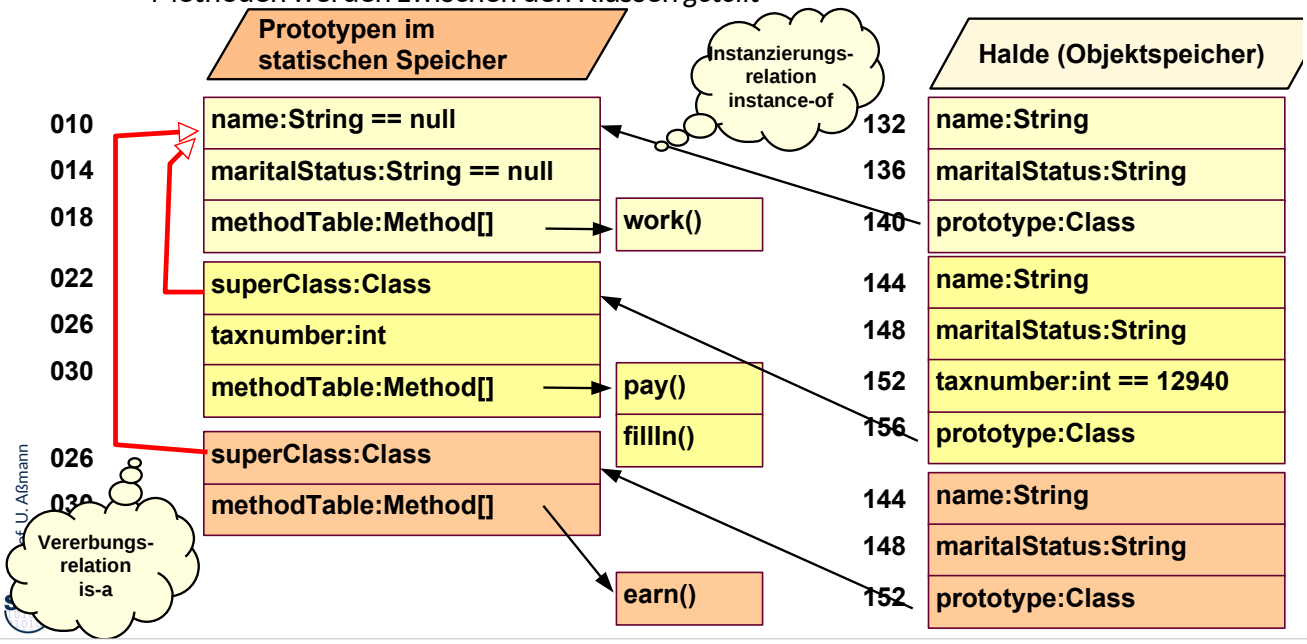
- ... am Beispiel Steuerzahler



Manche Leute zahlen Steuern, während andere schwarz arbeiten, d.h. nur Geld verdienen. Die Vererbungshierarchie wird auch in den Speicher eines Programms abgebildet. Ein laufendes Java-Programm verwaltet einen Laufzeit-Baum seiner Vererbungshierarchie.

Vererbung im Speicher

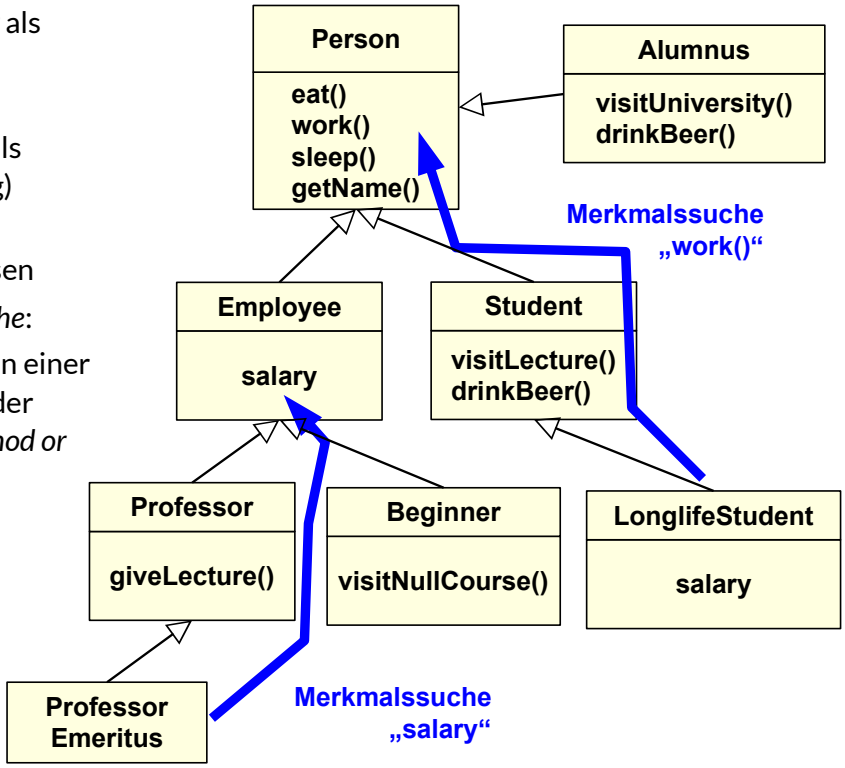
- Die Vererbungsrelation wird im Speicher als *Baum* zwischen den Prototypen der Ober- und Unterklassen dargestellt (Verzeigerung von unten nach oben)
  - Unterscheide davon die Objekt-Prototyp-Relation instance-of!
- Methoden werden zwischen den Klassen *geteilt*



Für die Suche nach Merkmalen (“features”) werden sich im Speicher die Links zu den Superklassen gemerkt (siehe rote Zeiger).

Merkmalssuche im Vererbungsbaum

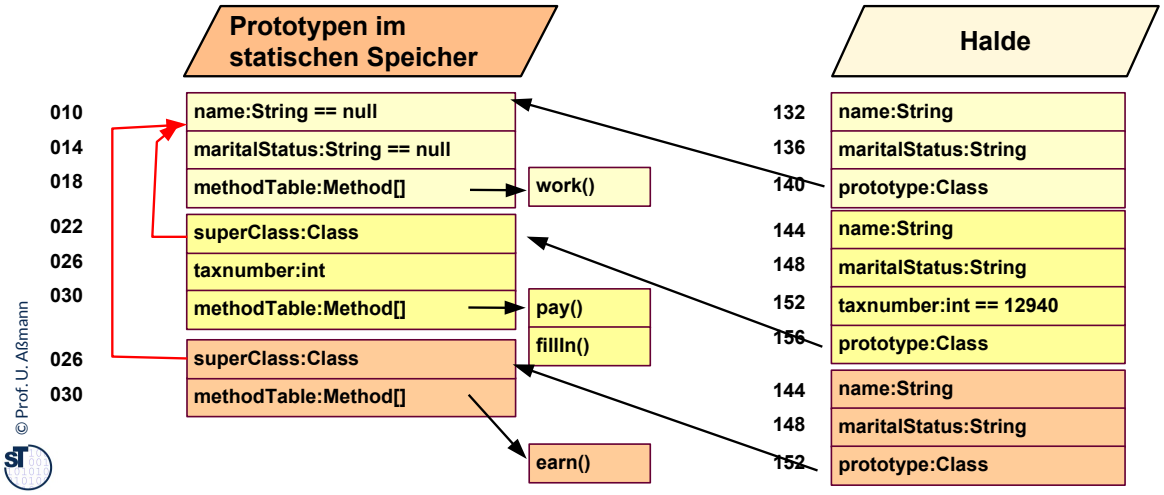
- Oberklassen sind *allgemeiner* als Unterklassen (Prinzip der Generalisierung)
- Unterklassen sind *spezieller* als Oberklassen (Spezialisierung)
  - Unterklassen *erben* alle Merkmale der Oberklassen
- *Methoden-* bzw. *Merkmalssuche*:
  - Wird ein Merkmal nicht in einer Klasse definiert, wird in der Oberklasse *gesucht (method or feature resolution)*



Die Suche nach einem Merkmal (Attribut oder Methode) beginnt mit dem konkreten Typ des Objekts. Wird nichts gefunden, setzt man die Suche nach oben in den Superklassen fort. Dazu benötigt man den Vererbungsbaum.

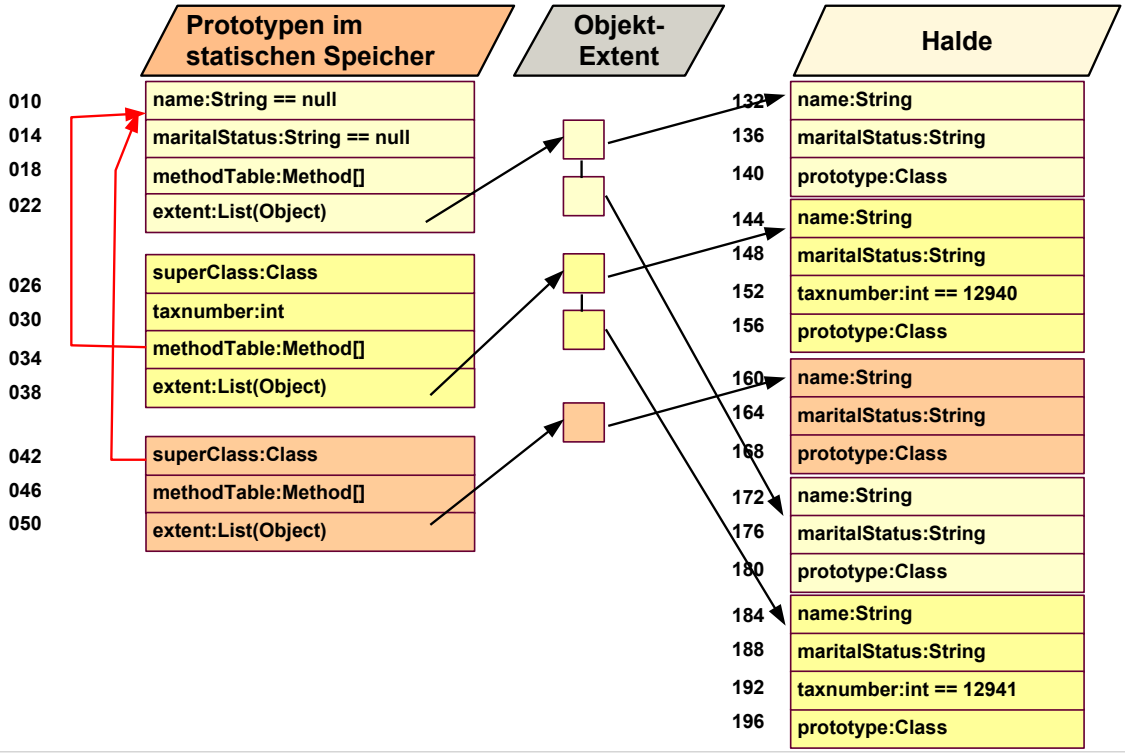
Merkmalssuche im Speicher - Beispiele

1) Suche Attribut *name* in Steuerzahler: direkt vorhanden
2) Suche Methode *pay()* in Steuerzahler: Schlage Prototyp nach, finde in Methodentabelle des Prototyps
3) Suche Methode *work()* in Steuerzahler: Schlage Prototyp nach, Schlage Oberklasse nach (Person), finde in Methodentabelle von Person
4) Suche Methode payback() in Steuerzahler: Schlage Prototyp nach, Schlage Oberklasse nach (Person); exisitert nicht in Methodentabelle von Person. Da keine weitere Oberklasse existiert, wird ein Fehler ausgelöst *"method not found" "message not understood"*



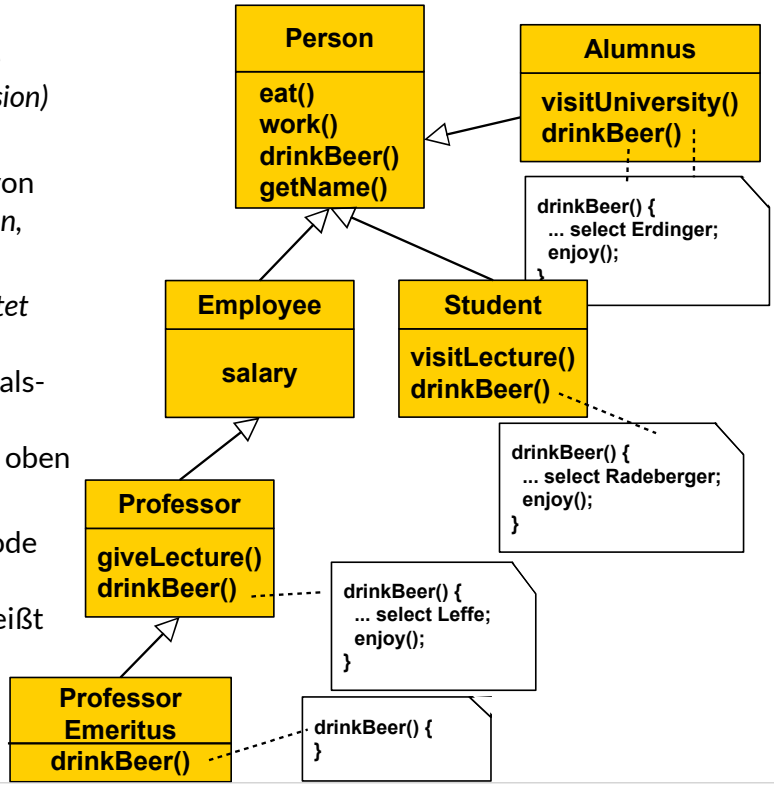
Objekt-Extent im Speicher, mit Vererbung

- Zu einer Klasse vereinige man alle Extents aller Unterklassen



Erweitern und Überschreiben von Merkmalen

- Eine Unterklasse kann neue Merkmale zu einer Oberklasse hinzufügen *(Erweiterung, extension)*
- Definiert eine Unterklasse ein Merkmal erneut, spricht man von einer *Redefinition (Überschreiben, overriding)*
  - Dieses Merkmal *überschattet (verbirgt)* das Merkmal der Oberklasse, da der Merkmals-suchalgorithmus in der Hierarchie von unten nach oben sucht.
  - Die überschriebene Methode hat mehrere Implementierungen und heißt *polymorph* oder *virtual*



Es ist möglich, Merkmale einer Oberklasse beim Bilden von Unterklassen zu redefinieren (“überschreiben”). Die Merkmalssuche findet dann die “feineren” Merkmale in den Unterklassen zuerst und ignoriert die Merkmale der Oberklassen.

11.1.3. Die oberste Klasse von Java: "Object"

- **java.lang.Object:** allgemeine Eigenschaften aller Objekte und Klassen
  - Jede Klasse ist Unterklasse von Object ("extends Object").
  - Diese Vererbung ist *implizit* (d.h. man kann "extends Object" weglassen).
  - Wiederverwendung in der gesamten JDK-Bibliothek!
- Jede Klasse kann die Standard-Operationen überdefinieren:
  - equals: Objektgleichheit (Standard: Referenzgleichheit)
  - hashCode: Zahlcodierung
  - toString: Textdarstellung, z.B. für println()

```
class Object {
      protected Object clone (); // kopiert das Objekt
      public boolean equals (Object obj);
          // prüft auf Gleichheit zweier Objekte
      public int hashCode();      // produce a unique identifier
      public String toString();  // produce string representation
      protected void finalize(); // lets GC run
      Class getClass();          // gets prototype object
}
```

Die clone()-Methode kopiert den Zustand eines Objekts (alle Attribute) in einen neuen Speicherbereich und erzeugt ein identisches, aber verschiedenes Objekt.

Equals() wird jedesmal aufgerufen, wenn der Gleichheitsoperator "==" benutzt wird.

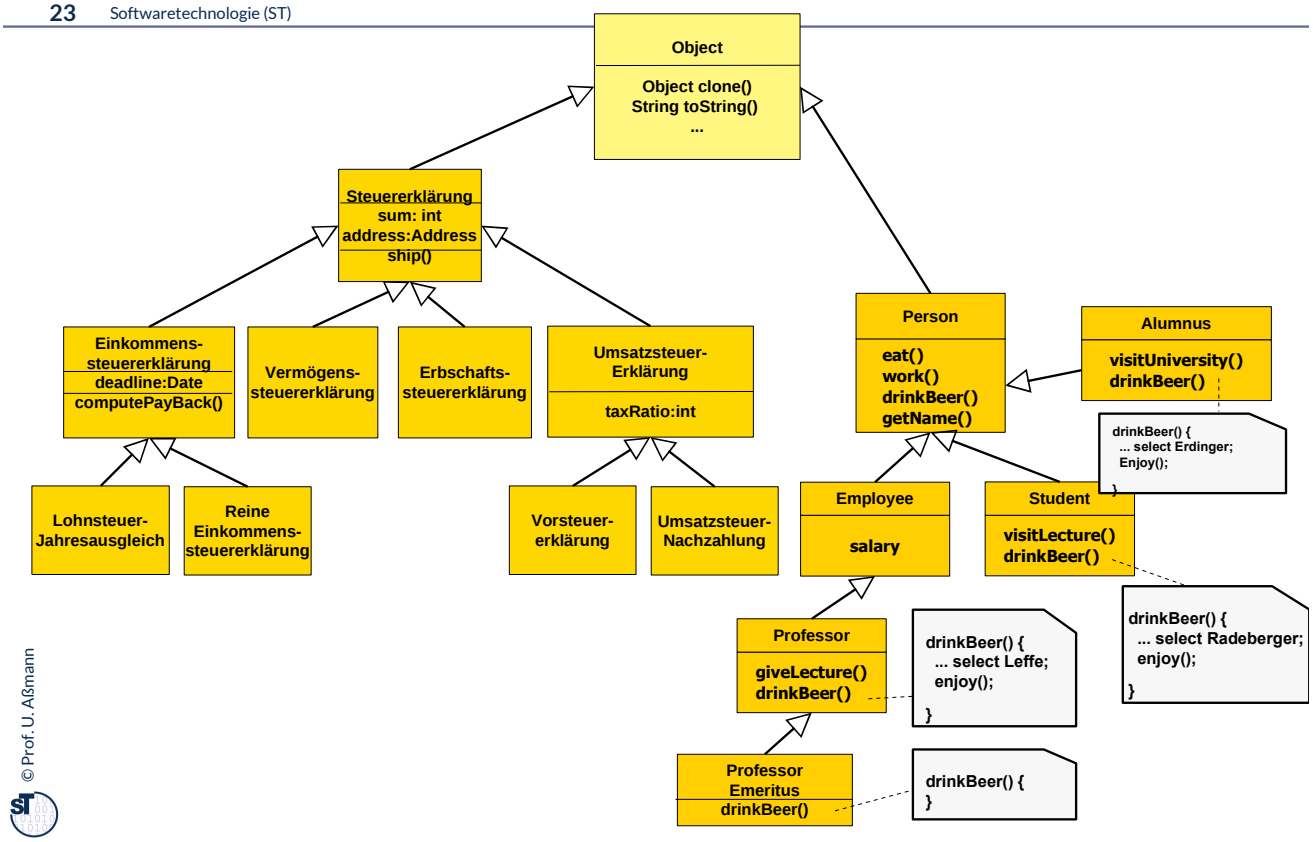
HashCode() ermittelt einen eindeutigen Identifikator des Objekts (große Zahl), die mit hoher Wahrscheinlichkeit eindeutig ist. Zwei Objekte mit verschiedenem Hashcode sind unterschiedlich.

Die toString()-Methode wird praktischerweise automatisch aufgerufen, wenn ein Objekt mit Konkatenationsoperatoren in einen Stringausdruck eingefügt wird.

Die finalize()-Methode wird aufgerufen, bevor der Abfallsammler das Objekt auf die freie Halde legt. Man kann also "aufräumen".

Schließlich liefert die getClass()-Methode den Klassenprototypen eines Java-Objekts.

Vererbung von Object auf Anwendungsklassen



Die Klasse Object wird implizit an alle Java-Klassen vererbt.

11.E1 Exkurs: Lernen mit Begriffshierarchien

Begriffshierarchien können zum Lernen eingesetzt werden

Der einzige Weg, auf welchem wahre Kenntnis erreicht werden kann, ist durch liebesvolles Studium.

Carl Hilty (1831 - 1909), Schweizer Staatsrechtler und Laientheologe

http://www.aphorismen.de/

Blooms Taxonomie des Lernens

- [Wikipedia, Lernziele] Die 6 Stufen im kognitiven Bereich lauten:
- **Lehrlingsschaft**
  - **Stufe 1) Kenntnisse / Wissen:** Kenntnisse konkreter Einzelheiten wie Begriffe, Definitionen, Fakten, Daten, Regeln, Gesetzmäßigkeiten, Theorien, Merkmalen, Kriterien, Abläufen; Lernende können Wissen abrufen und wiedergeben.
  - **Stufe 2) Verstehen:** *Lernende können Sachverhalt mit eigenen Worten erklären oder zusammenfassen; können Beispiele anführen, Zusammenhänge verstehen; können Aufgabenstellungen interpretieren.*
- **Gesellenschaft**
  - **Stufe 3) Anwenden:** Transfer des Wissens, problemlösend; Lernende können das Gelernte in neuen Situationen anwenden und unaufgefordert Abstraktionen verwenden oder abstrahieren.
  - **Stufe 4) Analyse:** Lernende können ein Problem in einzelne Teile zerlegen und so die Struktur des Problems verstehen; sie können Widersprüche aufdecken, Zusammenhänge erkennen und Folgerungen ableiten, und zwischen Fakten und Interpretationen unterscheiden.
  - **Stufe 5) Synthese:** Lernende können aus mehreren Elementen eine neue Struktur aufbauen oder eine neue Bedeutung erschaffen, können neue Lösungswege vorschlagen, neue Schemata entwerfen oder begründete Hypothesen entwerfen.
- **Meisterschaft**
  - **Stufe 6) Beurteilung:** Lernende können den Wert von Ideen und Materialien beurteilen und können damit Alternativen gegeneinander abwägen, auswählen, Entschlüsse fassen und begründen, und bewusst Wissen zu anderen transferieren, z. B. durch Arbeitspläne.

Warnung: Gelesen ist noch nicht verstanden.

Verstanden ist noch nicht behalten.

Behalten ist noch nicht kapiert.

Lernen braucht Zeit. Im Laufe des Studiums merkt man, dass man einen Stoff versteht und kapiert hat, wenn man ihn einem anderen Menschen flüssig erklären kann (aktives Wissen). Das testen mündliche Prüfungen: Kann man den Stoff dem Professor erklären?

Noch weiter geht Stufe 3, das Anwenden von Wissen auf unbekannte Probleme – das wird von Ihnen in der industriellen Praxis immer verlangt.

Lernlandkarten und Lernmatrizen als Hilfsmittel

- Erstellen Sie eine Strukturkarte (concept map) der Vorlesung zur Vorbereitung für die Klausur
- **Vorlesungslandkarte**: Quasi-hierarchische Darstellung der Inhalte der Vorlesung
  - gegliedert wie die Vorlesung
  - gefüllt mit Begriffen, die Sie erklären können (Bloom-Stufe 1+2)
  - gefüllt mit Fragen
- **Vorlesungsmatrix**: Matrixartige Darstellung der Inhalte
  - auf die Vorlesungslandkarte aufbauend
  - Kreuzen mit zweiter Dimension: Querschneidende Aspekte wie Analyse, Design, Entwurfsmuster in die zweite Dimension eintragen
  - Damit die Vorlesungslandkarte in einen zweiten Zusammenhang bringen (Bloom-Stufe 3+4)
- **Übung:** Erstellen Sie eine Vorlesungslandkarte von Vorlesung 10, “Objekte und Klassen”
  - Erstellen Sie eine Vorlesungslandkarte von Vorlesung 11, “Vererbung und Polymorphie”
  - Ermitteln sie querscheidende Aspekte wie Objektallokation, Speicherrepräsentation
  - Entwickeln Sie eine Vorlesungsmatrix

Exkursion: Safari Books Online

- Die SLUB hat für sie eine Menge von Büchern online
- Von innerhalb der TU Dresden **kostenlos lesbar**
- http://proquest.tech.safaribooksonline.de/

Sehr empfohlen für die Technik des Lernens und wiss. Arbeitens:

- Stickel-Wolf, Wolf. Wissenschaftliches Arbeiten und Lerntechniken. Gabler. Blau. Sehr gutes Überblicksbuch für Anfänger.
- **Kurs "Academic Skills for Computer Scientists" (3/1/0)**
  - Wintersemester
  - http://st.inf.tu-dresden.de/teaching/asics

11.2 Schnittstellen und Abstrakte Klassen für das Programmieren von Löchern

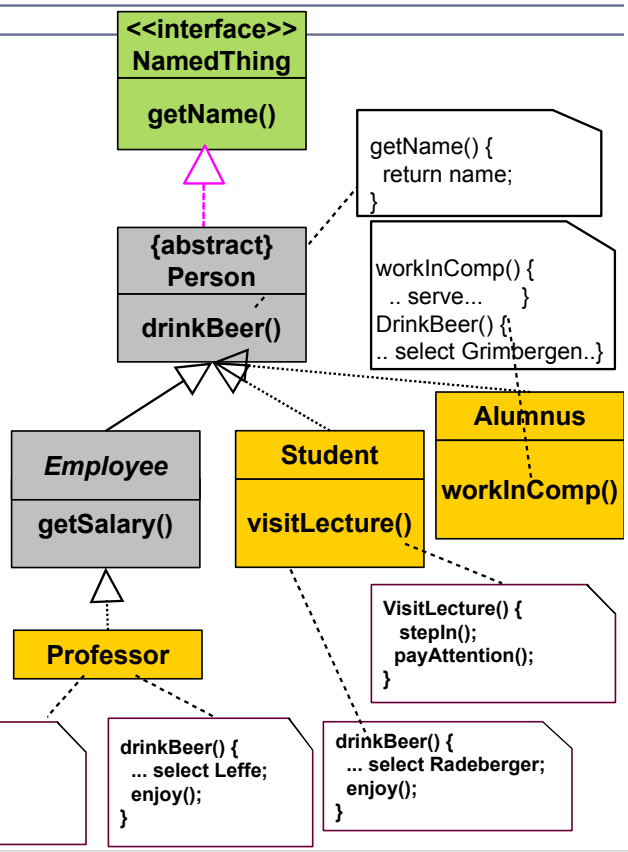
Typen können verschiedene Formen annehmen.
Eine partiell spezifizierte Klasse (Schnittstelle, abstrakte Klasse, generische Klasse) macht Vorgaben für Anwendungsentwickler

Schnittstelle:

Loch an Loch – und sie hält doch.

Schnittstellen und Abstrakte Klassen bilden “Haken”, an die man Klassenimplementierungen anhängen kann

- Beim Entwurf von Bibliotheken soll für Anwendungen eine Struktur vorgegeben werden, an die sich alle Anwendungsprogrammierer halten müssen
- **Vorsehen von “Haken” (hooks)** in der Vererbungshierarchie:
- **Abstrakte Klassen** werden mit einem speziellen Markierer (tagged value) gekennzeichnet ({abstract}) oder *kursiv* gemalt
- **Schnittstellen** beschreiben einen Teil der Funktionalität eines Objekts
  - In UML werden sog. Stereotypen vergeben, um Schnittstellen zu kennzeichnen (<<interface>>)
  - Sie dienen dazu, Verhalt Objektes in einem be Kontext festzulegen



In Java programmiert man mit Schablonen:

• Schnittstellen geben Methodensignaturen vor, die von Unterklassen implementiert werden müssen

• Abstrakte Klassen ebenfalls

• Generische Klassen geben Typen für Attribute und Methodenparameter vor, die zu verwenden sind.

Damit schafft es ein Chef-Entwickler, anderen Entwicklern Vorgaben zu machen:

• Der Programmierer einer Bibliothek kann vorgeben, wie man sie benutzt

• Die Entwickler des “Java Development Kits (JDK)” geben uns allen vor, wie wir Anwendungen zu schreiben haben (“Frameworks”, Rahmenwerke)

Das ist in nicht-objektorientierten Sprachen bei Weitem nicht so gut möglich. Daher unterstützen OO-Programme Wiederverwendung.

Schnittstellen und Klassen in Java geben “Hooks” vor (“abstract”)


```
interface NamedThing {
  String getName();  // no implementation
}
abstract class Person implements NamedThing {
  String name;
  String getName() { return name; } // implementation exists
  abstract void drinkBeer();        // no implementation
}
abstract class Employee extends Person {
  abstract void getSalary();        // no implementation
}
class Professor extends Employee {  // concrete class
  void getSalary() { goToBank(); withDraw(); }
  void drinkBeer() { .. select Leffe(); enjoy(); }
}
class Student extends Person {      // concrete class
  void visitLecture() { stepIn(); payAttention(); }
  void drinkBeer() { .. select Radeberger(); enjoy(); }
}
```


Alle in einer Schnittstelle definierten Methodensignaturen bilden “Löcher”, “Haken”, “hooks”.

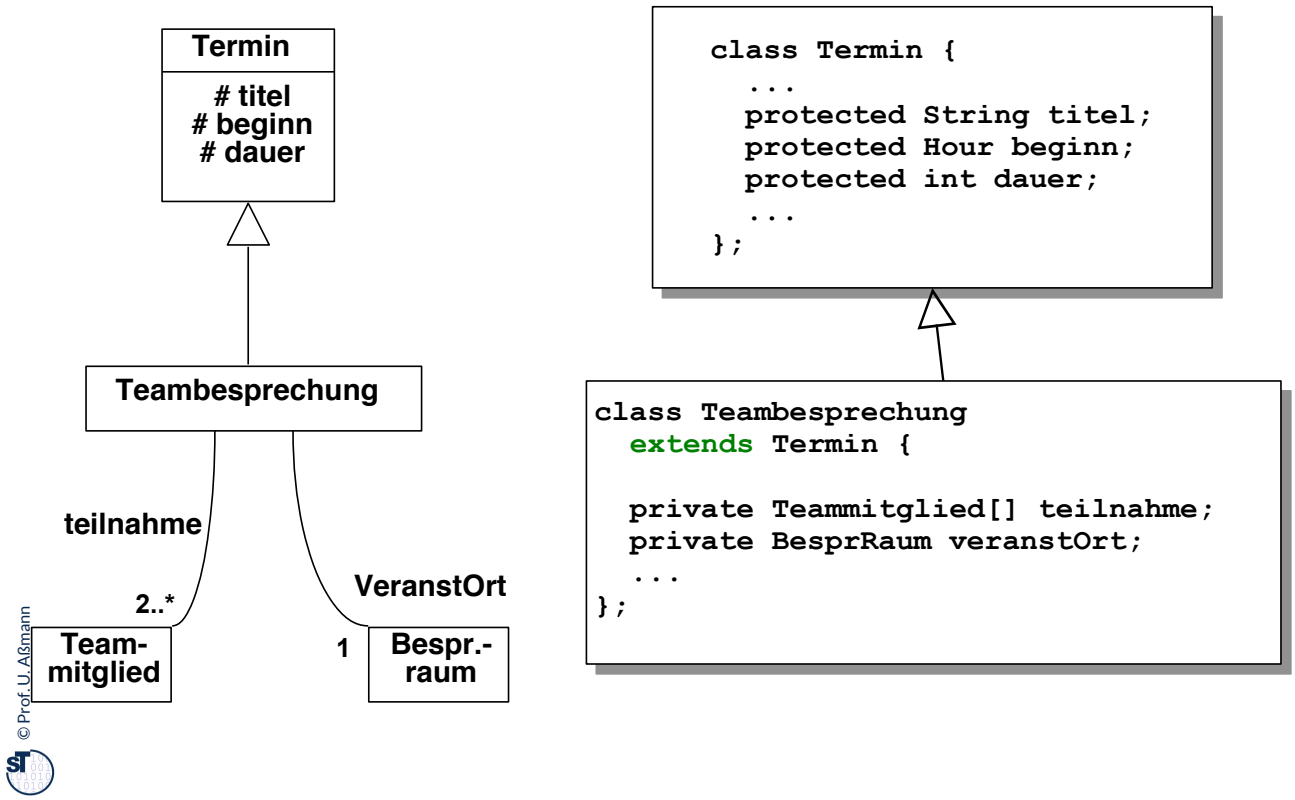
Eine konkrete Unterklasse muss sie alle füllen, also “Methoden einhängen”.

Schnittstellen und Klassen in Java geben “Hooks” vor


```
interface NamedThing {
  String getName();  // no implementation
}
abstract class Person implements NamedThing {
  String name;
  String getName() { return name; } // implementation exists
  abstract void drinkBeer();        // no implementation
}
abstract class Employee extends Person {
  abstract void getSalary();        // no implementation
}
class Professor extends Employee {  // concrete class
  void getSalary() { goToBank(); withDraw(); }
  void drinkBeer() { .. select Leffe(); enjoy(); }
}
class Student extends Person {      // concrete class
  void visitLecture() { stepIn(); payAttention(); }
  void drinkBeer() { .. select Radeberger(); enjoy(); }
}
class Alumnus extends Person {
   // new concrete class must fit to Person and NamedThing
   void workInComp() { .. serve... }
   void drinkBeer() { ...select Grimbergen... }
}
```


Unterklassen dürfen neue Methoden definieren. Wenn sie aber eine Methodensignatur einer Schnittstelle füllen wollen, müssen sie dieser genau entsprechen.

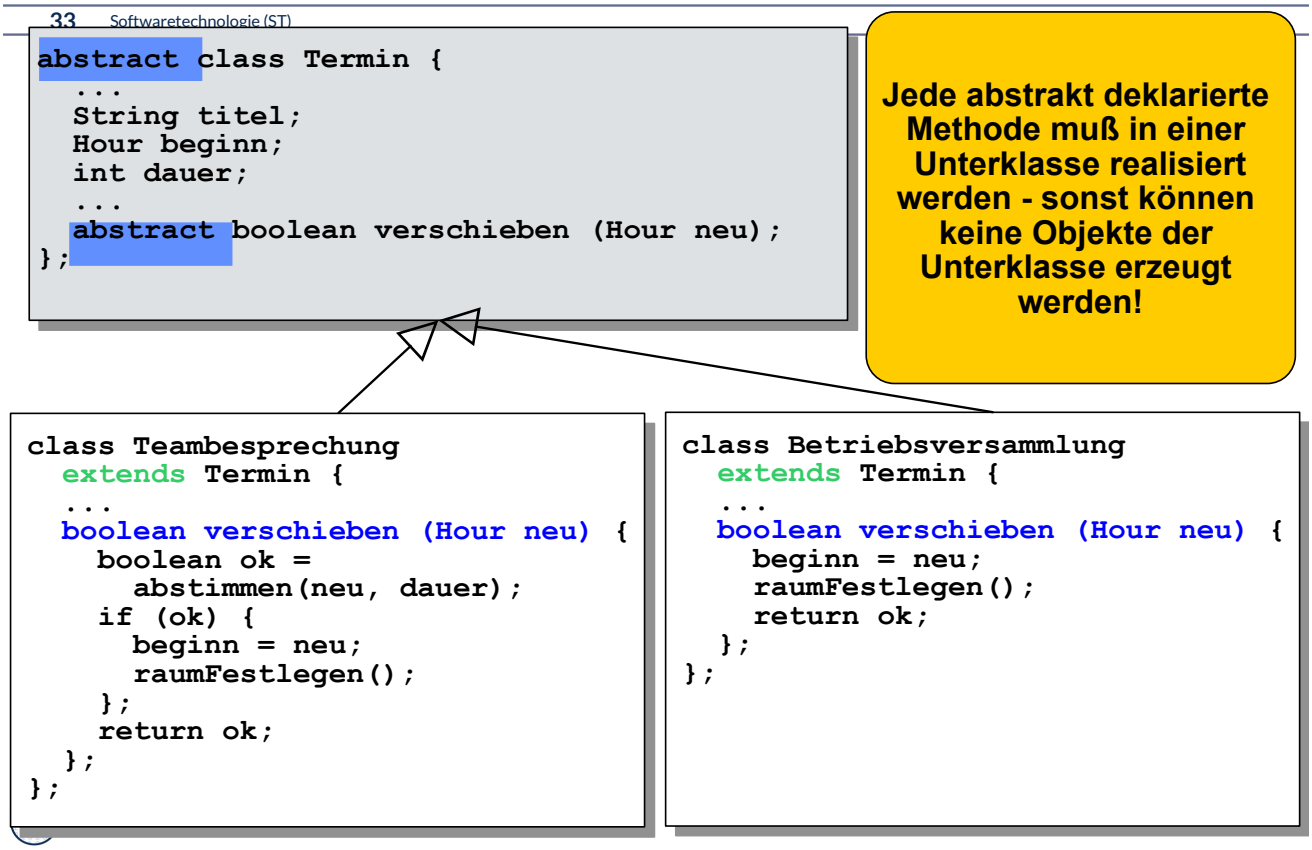
Laufendes Beispiel Terminverwaltung



Die Klasse Termin definiert keine Methodensignaturen, aber geschützte Attribute.

Auch hier können Unterklassen weitere Attribute hinzufügen.

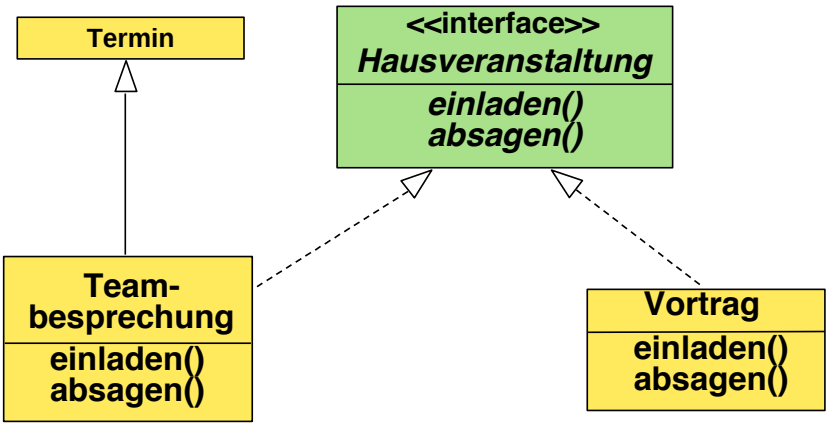
Beispiel (2): Abstrakte Klassen und Abstrakte Operationen



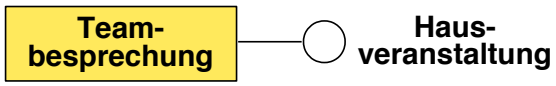
Fügt man eine abstrakte Methode hinzu, muss man auch die Klasse als abstrakt deklarieren.

Diese können dann in konkreten Unterklassen gefüllt werden.

Einfache Vererbung von Typen durch Schnittstellen



Hinweis: "Lutscher"-Notation *(lollipop)* für Schnittstellen „Klasse bietet Schnittstelle an":



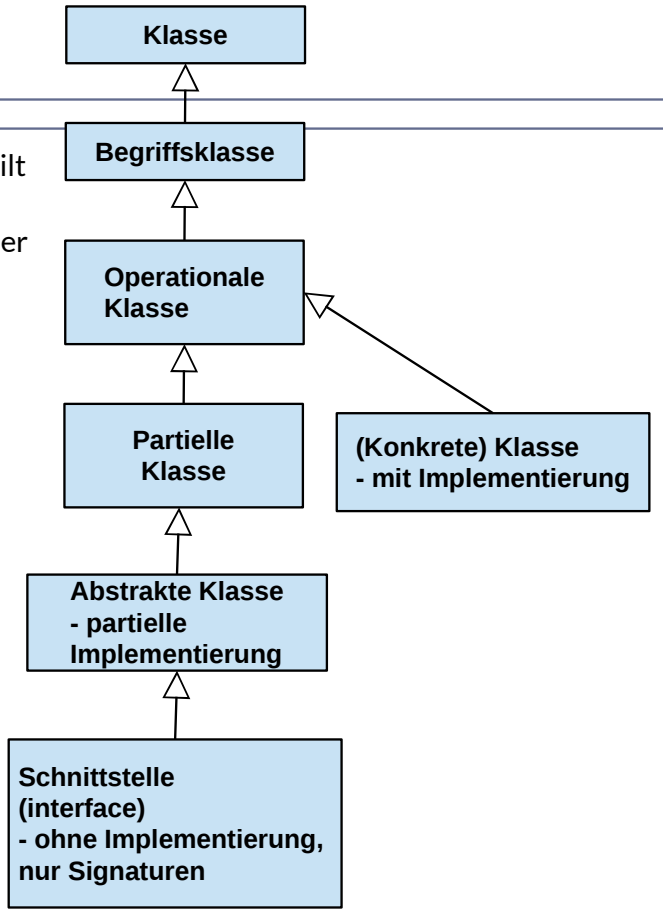
Das Abteil "angebotene Funktionen" kann in UML durch einen Lollipop benamt und abgekürtzt werden.

Vergleich von Schnittstellen und abstrakte Klassen

Abstrakte Klasse	Schnittstelle
Enthält Attribute und Operationen	Enthält nur Operationen (und ggf. Konstante)
Kann Default-Verhalten festlegen	Kann kein Default-Verhalten festlegen
Wiederverwendung von Schnittstellen und Code, aber keine Instanzbildung	Redefinition unsinnig
Default-Verhalten kann in Unterklassen überdefiniert werden	Java und UML: Eine Klasse kann mehrere Schnittstellen implementieren
Java: Unterklasse kann nur von einer Klasse erben	Schnittstelle ist eine spezielle Sicht auf eine Klasse

Q2: Begriffshierarchie von Klassen (Erw.)

- Operationale Klassen werden unterteilt in Klassen mit, ohne, und mit Implementierung einer Untermenge der Operationen
- Schnittstellen und Abstrakte Klassen dienen dem Teilen von Typen und partiellem Klassen-Code



Wieder eine Taxonomie, diesmal von Klassen. Partielle Klassen geben vor, wie man Unterklassen bzw. Typen bildet.

11.3. Polymorphie

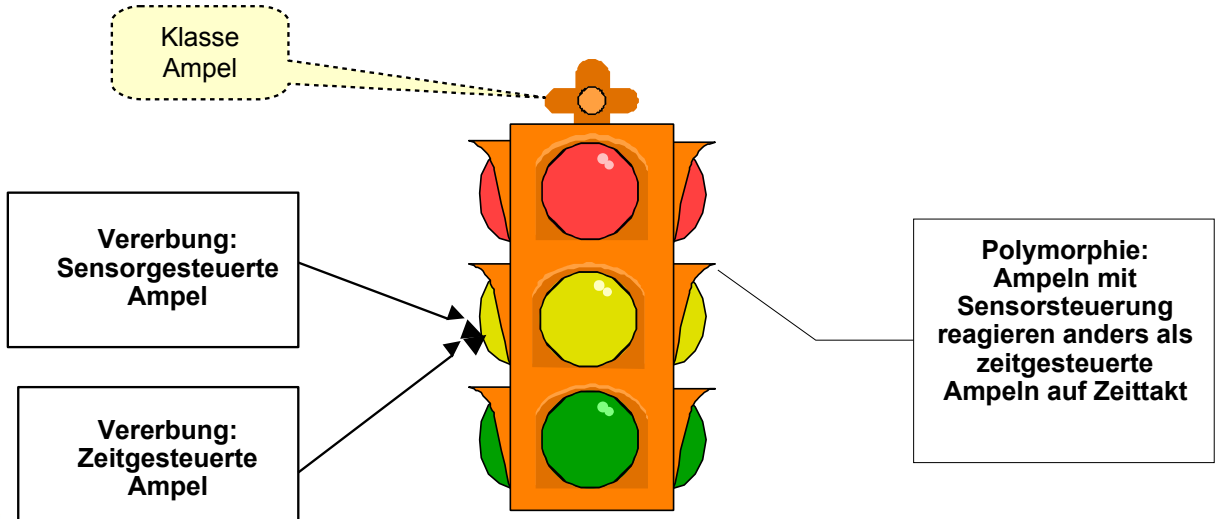
.. verändert das Verhalten einer Anwendung, ohne den Code zu verändern

- Polymorphie erlaubt *dynamische Architekturen*
  - Dynamisch wechselnd
  - Unbegrenzt viele Objekte
- Zentraler Fortschritt gegenüber einfachem imperativen Programmieren

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

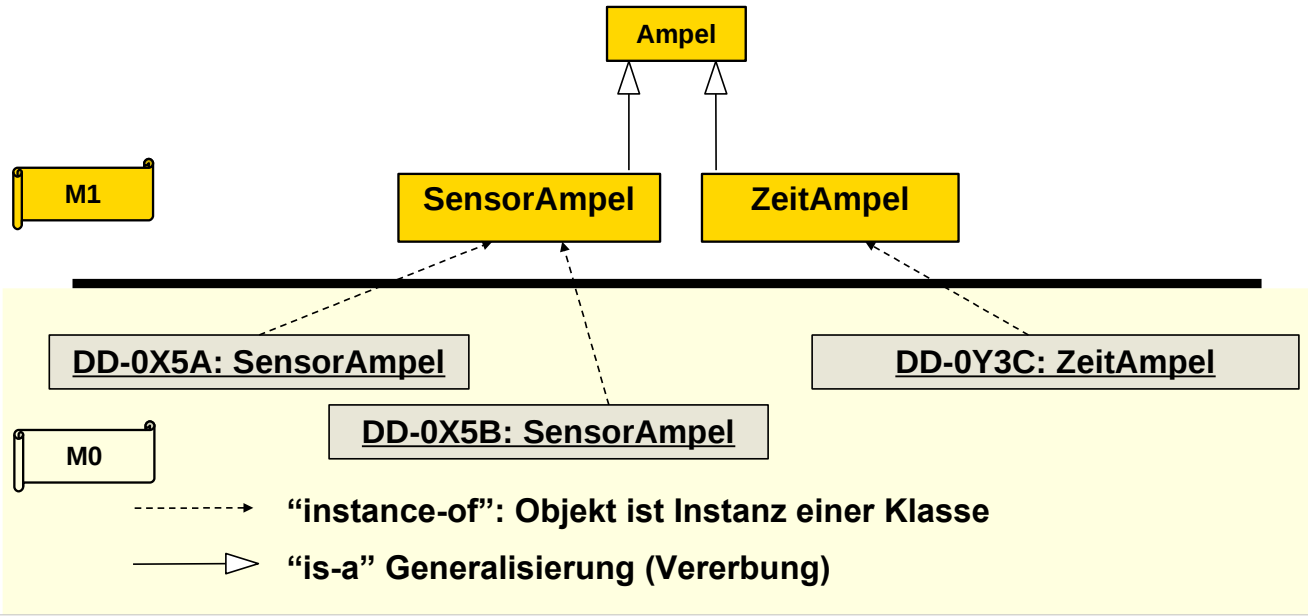
Vererbung und Polymorphie

- Welcher Begriff einer Begriffshierarchie wird verwendet (Oberklassen/ Unterklassen)?
- Wie hängt das Verhalten des Objektes von der Hierarchie ab (spezieller vs allgemeiner)?



Beispiel: Ampel-Klasse und Ampel-Objekte

- Jede Ampel reagiert auf den Zeittakt.
  - Die Klasse **Ampel** schreibt vor, daß auf die Nachricht „Zeittakt“ reagiert werden muß.
  - Verschiedene Reaktionen der Unterklassen

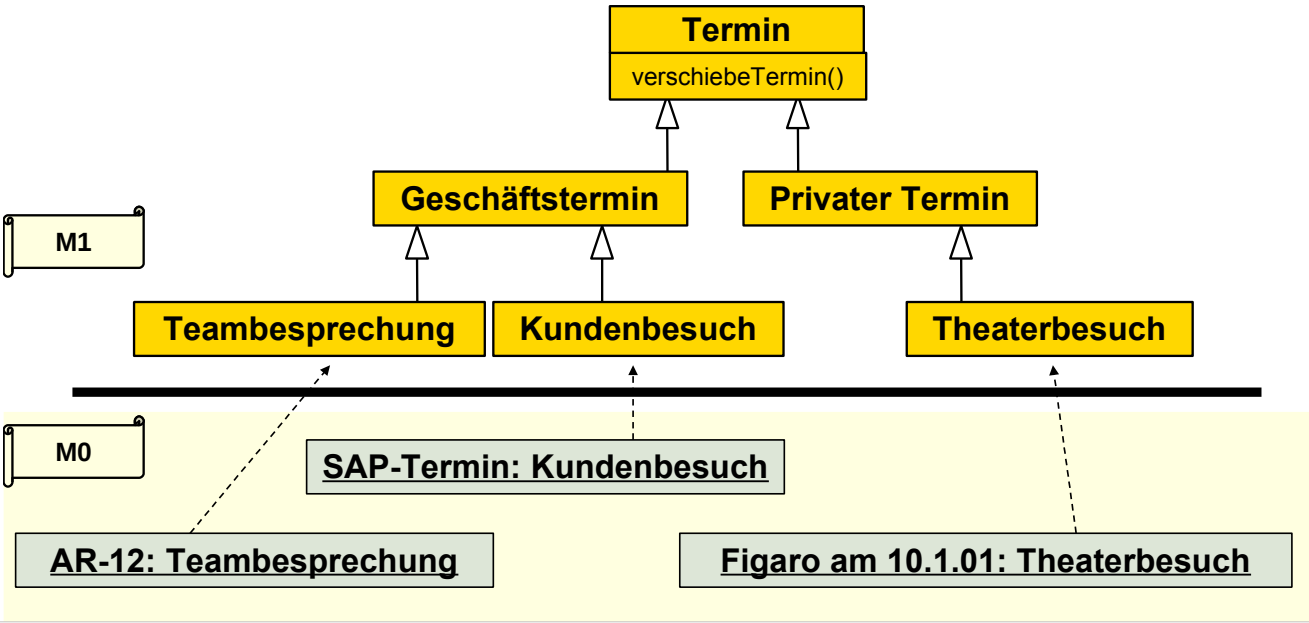


Man sieht hier das erste Mal das Zusammenspiel eines Objekt- mit einem Klassendiagramm. Während M1, die Klassenebene während der Ausführung eines Programms gleich bleibt, ändert sich der Snapshot des Objektnetzes auf M0 ständig (Lebensphase).

Beide Ebenen (M0 und M1) sind durch die “instance-of”-Relation enthalten: Objekt ist Element einer Klasse.

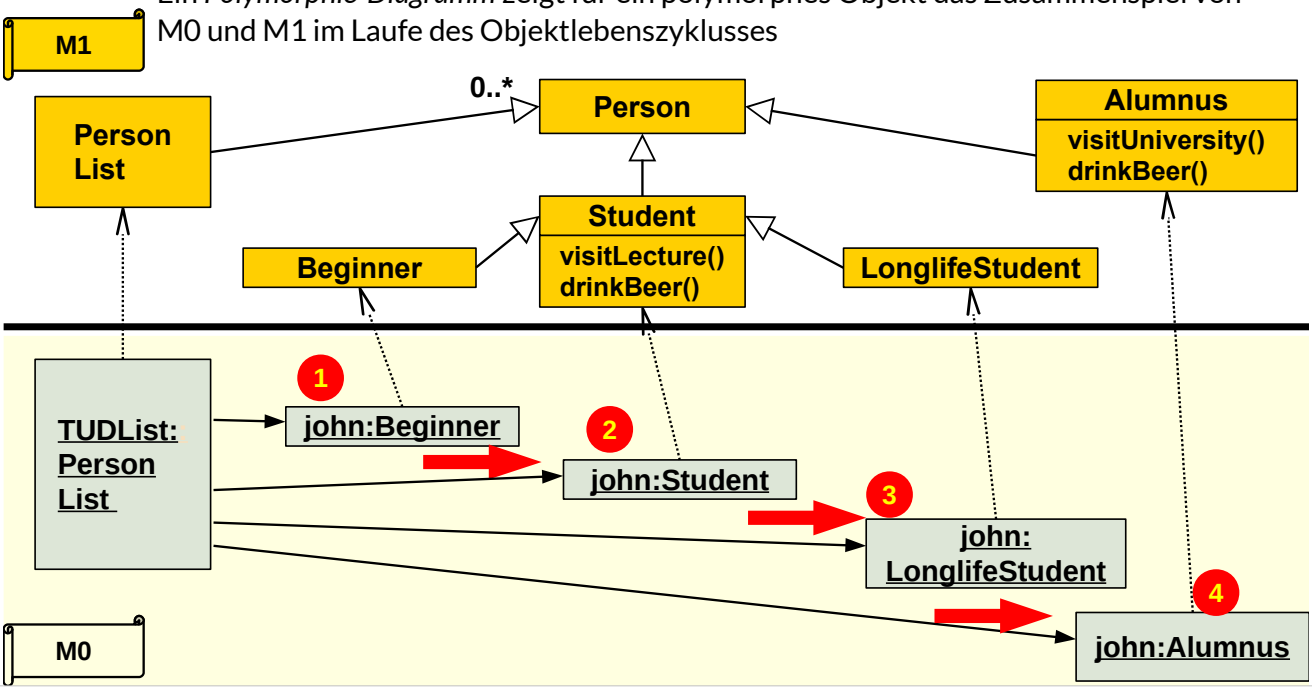
Beispiel: Termin-Klasse und Termin-Objekte

- Allgemeines Merkmal: Jeder Termin kann verschoben werden.
  - Daher schreibt die Klasse **Termin** vor, daß auf die Nachricht „verschiebeTermin“ reagiert werden muß.
- Unterklassen *spezialisieren* Oberklassen; Oberklassen *generalisieren* Unterklassen



Polymorphie (polymorphism) (im Polymorphie-Diagramm)

- Zur Laufzeit kann jedes Objekt einer Unterklasse ein Objekt einer Oberklasse *vertreten*. Das Objekt der Oberklasse ist damit *vielgestaltig (polymorphic)*.
- Ein *Polymorphie-Diagramm* zeigt für ein polymorphes Objekt das Zusammenspiel von M0 und M1 im Laufe des Objektlebenszyklusses



Im Laufe des Lebens eines Objekt kann es seinen Typ wechseln (Polymorphie). Der Wechsel ist eingeschränkt auf die Typen der Vererbungshierarchie.

Polymorphie ist typisch bei Lebensphasen eines Objekts (hier Student).

Wechsel der Gestalt (Objektevolution und Polymorphie)

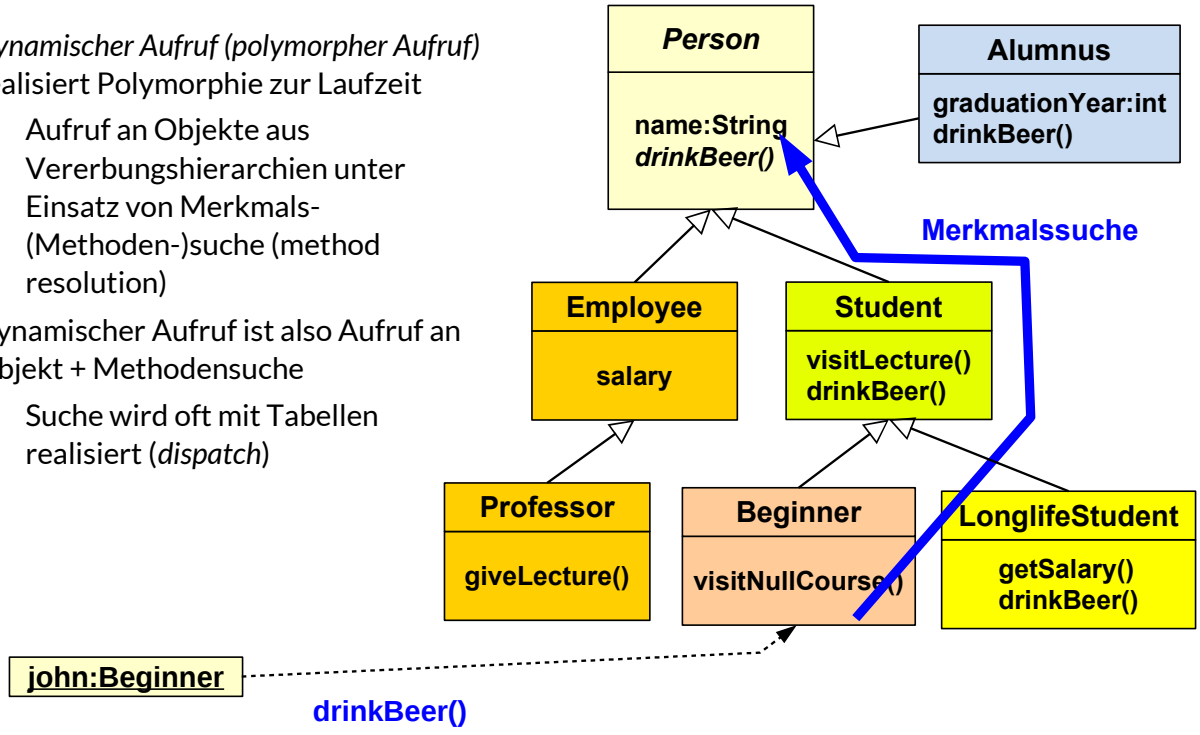
- Die genaue Unterklasse einer Variablen wird festgestellt
  - Beim **Erzeugen** (der Allokation) des Objekts (Allokationszeit, oft in der Aufbauphase des Objektnetzes), oft in einem alternativen Zweig des Programms alternativ festgelegt
  - Bei einer **neuen Zuweisung** (oft in einer Umbauphase des Objektnetzes)

```
Person john;

if (hasLeftUniversity)
  john = new Alumnus();
else
  john = new Student();

// which type has Person john here?
....

if (hasWorkedHardLongEnough)
  john = new Professor();
// which type has person here?
// how will the person act?
john.visitLecture();
john.drinkBeer();
```


Jeder Übergang einer Lebensphase wird durch die Allokation eines Objekts des neuen Typs eingeleitet. Oft kann man nur durch genaue Verfolgung des Programmablaufs (“Kontrollflussanalyse”) verstehen, welcher Typ genau an einem Programmpunkt vorliegt.

Dynamischer Aufruf (Dynamic dispatch)

- *Dynamischer Aufruf (polymorpher Aufruf)* realisiert Polymorphie zur Laufzeit
  - Aufruf an Objekte aus Vererbungshierarchien unter Einsatz von Merkmals-(Methoden-)suche (method resolution)
- Dynamischer Aufruf ist also Aufruf an Objekt + Methodensuche
  - Suche wird oft mit Tabellen realisiert (*dispatch*)

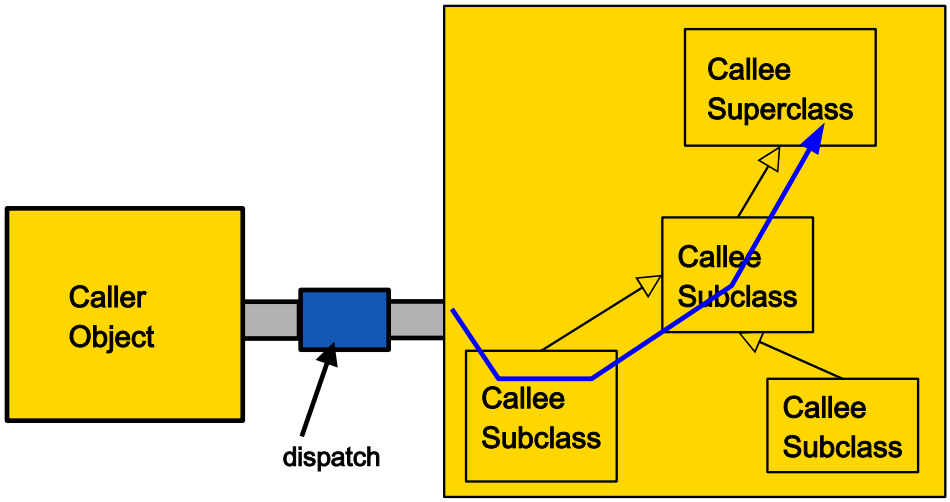


Ruft man in einem objektorientierten Programm eine Methode auf, muss diese sich nicht in dem lokalen Objekt befinden, sondern kann in einer Oberklasse stecken. Daher muss man nach Merkmalen dynamisch im Vererbungsbaum suchen.

Man sucht von der Klasse eines Objekts aus nach oben. Die erste Methode, die man findet, wird ausgewählt, egal, ob sich noch weitere Methoden *oberhalb* des Fundortes liegen.

Dynamischer Aufruf (Dynamic Dispatch)

- Vom Aufrufer aus wird ein Suchalgorithmus gestartet, der die Vererbungshierarchie aufwärts läuft, um die rechte Methode zu finden
  - Die Suche läuft tatsächlich über die Klassenprototypen
  - Diese Suche kann teuer sein und muß vom Übersetzer optimiert werden (dispatch optimization)

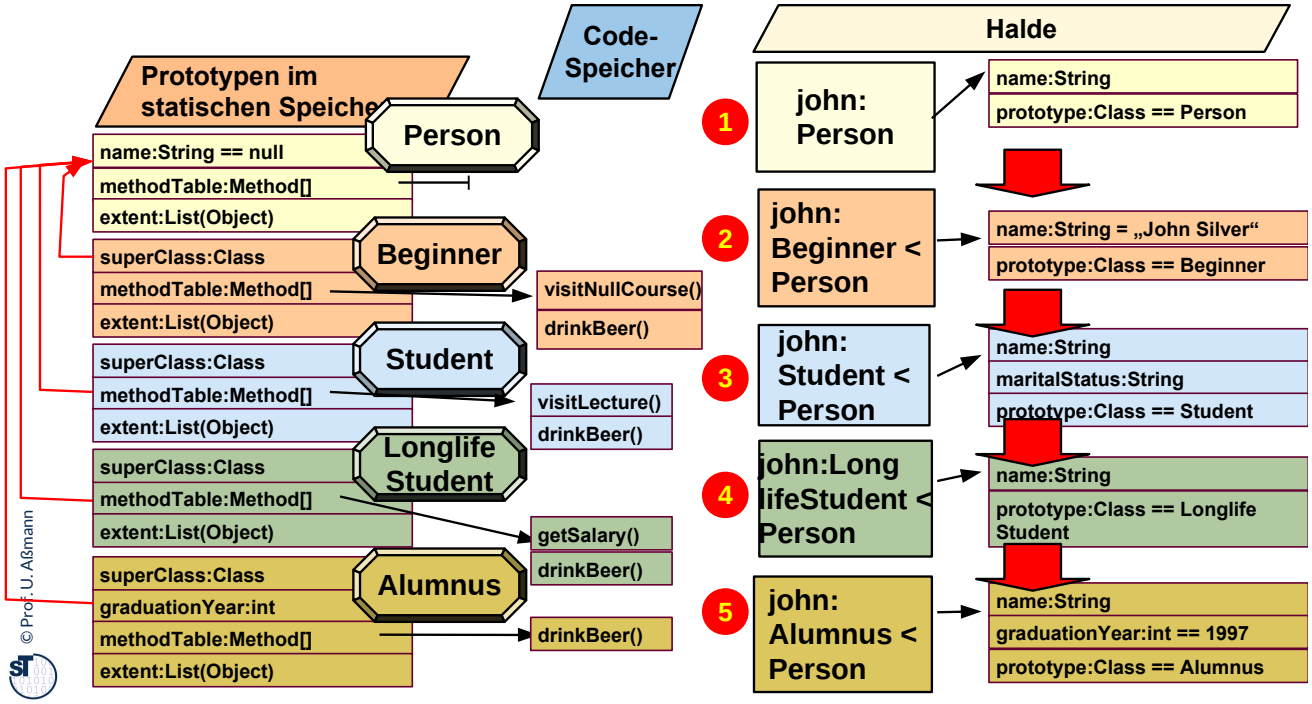


Der Typ eines Aufgerufenen (bzw. des Empfängers einer Botschaft) kann innerhalb des Vererbungsbaumes variieren. Der Typ des Aufgerufenen ist *vielgestaltig* (Polymorphie).

Zur Laufzeit muss nach dem konkret vorliegenden Typ im Vererbungsbaum gesucht werden, ähnlich, wie bei der Methodensuche. Diesen Suchprozess nennt man *dynamischen Aufruf (dynamic dispatch)*.

Was passiert beim polymorphen Aufruf im Speicher?

- Frage: Welche Inkarnation der Methode *drinkBeer()* wird zu den verschiedenen Zeitpunkten im Leben johns aufgerufen?



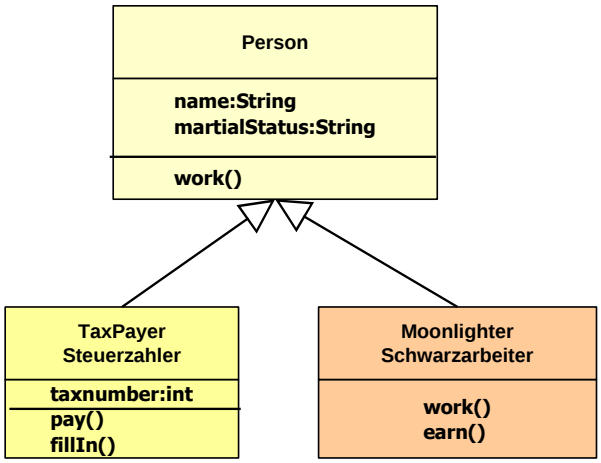
Über der Zeit können Objekte ihren Typ innerhalb des Vererbungsbaumes *ändern (Objektevolution)*, hier durch Farben ausgedrückt. Daher ändern u.U. die aufgerufenen Methoden ihre Natur (Polymorphie).

Jeder Typwechsel während der Objektevolution kann dazu führen, dass die Methodensuche ein anderes Ergebnis liefert. Daher ist der Code der Anwendung zwar dergleiche, aber seine Bedeutung ändert sich (Polymorphie).

Objektevolution und Polymorphie sind, sozusagen, die zwei Seiten einer Medaille.

Polymorphe und monomorphe Methoden

- Methoden, die nicht mit einer Oberklasse geteilt werden, können nicht polymorph sein
- Die Adresse einer solchen *monomorphen* Methode im Speicher kann statisch, d.h., vom Übersetzer ermittelt werden. Eine Merkmalssuche ist dann zur Laufzeit nicht nötig
- Frage: Welche der folgenden Methoden sind poly-, welche monomorph?



Objektevolution und Polymorphie sind flexible Mechanismen, die aber zu vielen Suchvorgängen zur Laufzeit führen und damit sich als teuer herausstellen. In einem Programm kann man daher oft wählen, ob man Polymorphie für eine Methode möchte:

• C++: Methode muss als *virtual* definiert werden

• Java: Hier hängt es davon ab, ob Unterklassen mit überschreibenden Methoden definiert sind oder nicht

• Python, etc: In Skriptsprachen spielt Geschwindigkeit nicht die Hauptrolle, sodass diese meistens nur Polymorphie bieten

11.4. Generische Klassen (Klassenschablonen, Template-Klassen, Parametrische Klassen)

… bieten eine weitere Art, mit Löchern zu programmieren, um Vorgaben zu machen

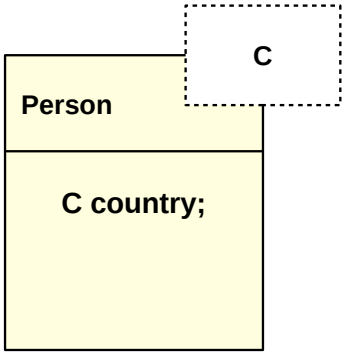
- Generische Klassen lassen den Typ von einigen Attributen und Referenzen offen (“generisch”)
- Sie ermöglichen typisierte Wiederverwendung von Code

Generische Klassen bieten eine weitere Möglichkeit des “Programmierens mit Löchern”. Während abstrakte Klassen einzelne Methoden “offen” lassen, lässt eine generische Klasse einen ganzen Typ offen, i.d.R. den Typ eines Attributs seiner Objekte.

Generische Klassen

Eine generische (parametrische, Template-) Klasse ist eine Klassenschablone, die mit einem oder mehreren Typparametern (für Attribute, Referenzen, Methodenparameter) versehen ist.

- In UML



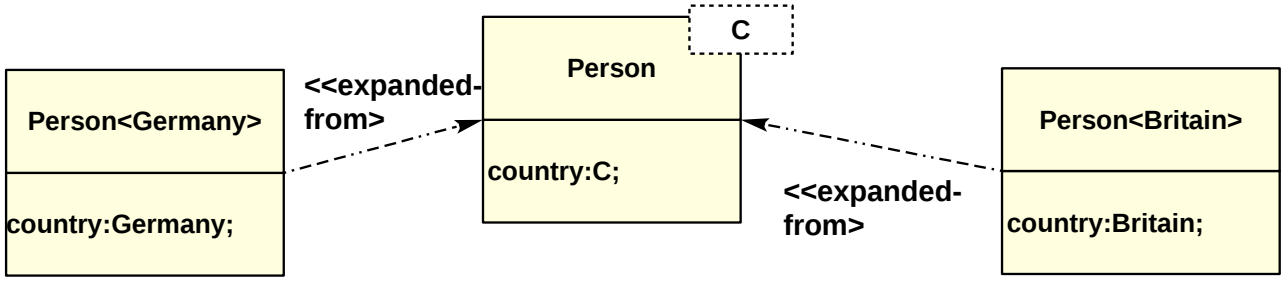
- In Java
  - Sprachregelung: “Person of C”

```
// Definition of a generic class
class Person<C> {
  C country;
}
```

```
/* Type definition using a generic type */
Person<Germany> egon;
Person<Britain> john;
```

Feinere statische Typüberprüfung für Attribute

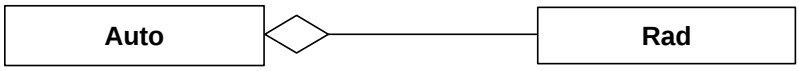
- Zwei Attributtypen, die durch Parameterisierung aus einer generischen Klassenschablone entstanden sind, sind nicht miteinander kompatibel
- Der Übersetzer entdeckt den Fehler (**statische Typprüfung**)




```
/* Type definition and initialization with object */
Person<Germany> egon = new Person<Germany>;
Person<Britain> john = new Person<Britain>;

/* Checks of assignments can use the improved typing */
john = egon;
```

Einsatzzweck: Typsichere Aggregation (has-a)

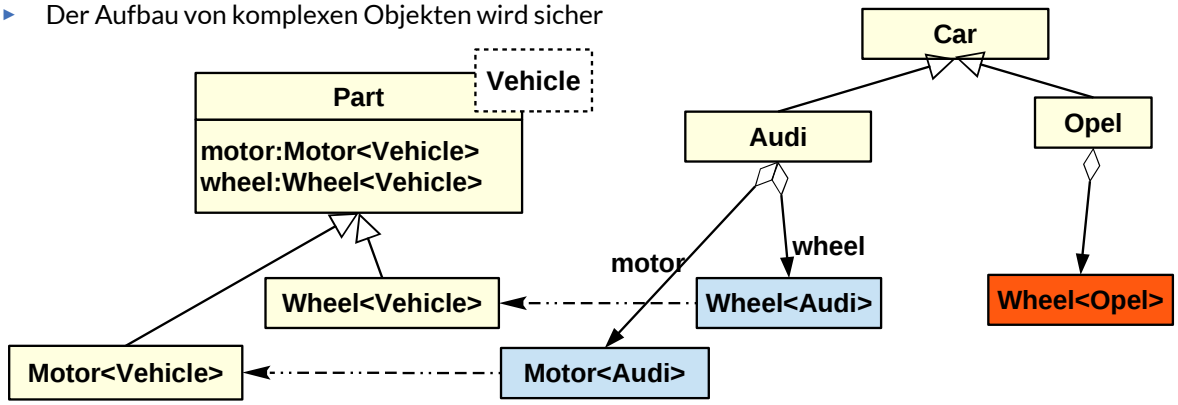
- **Definition:** Wenn eine Assoziation den Namen „hat-ein“ oder "besteht-aus" tragen könnte, handelt es sich um eine **Aggregation** (Ganzes/Teile-Relation).
  - Eine Aggregation besteht zwischen einem *Aggregat*, dem *Ganzen*, und seinen *Teilen*.
  - Die auftretenden Aggregationen bilden auf den Objekten immer eine transitive, antisymmetrische Relation (einen gerichteten zyklenfreien Graphen, *dag*).
  - Ein Teil kann zu mehreren Ganzen gehören (*shared*), zu einem Ganzen (*owns-a*) und exklusiv zu einem Ganzen (*exclusively-owns-a*)



Lies: „Auto *hat ein* Rad“

Einsatzzweck: Typsichere Aggregation (has-a)

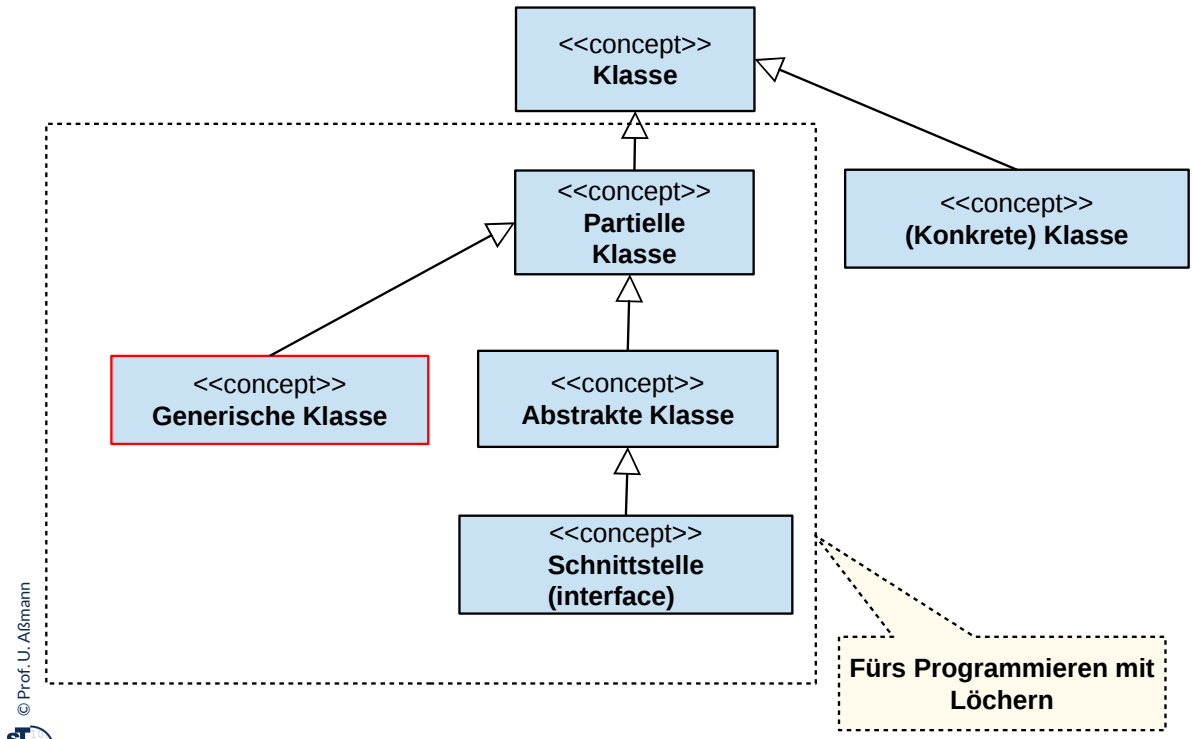
- Generische Klassen können Typen für Ganz-Teile-Beziehungen vorgeben
- Der Aufbau von komplexen Objekten wird sicher




```
/* Type definition and initialization with object */
Motor<Audi> motorOfAudi = new Motor<Audi>;
Wheel<Audi> wheelOfAudi = new Wheel<Audi>;
Wheel<Opel> wheelOfOpel = new Wheel<Opel>;

/* Checks of assignments can use the improved typing */
audi = new Audi();
audi.motor = motorOfAudi;
audi.wheel = wheelOfOpel;
```


Typparameter einer generischen Klasse werden zum feinen Typisieren von Attributtypen eingesetzt (Zeile 1-3). Die feinen Unterscheidungen der konkreten Typen helfen, Äpfel von Bananen zu unterscheiden und früh Fehler zu entdecken.

Q2: Begriffshierarchie von Klassen (Erweiterung)



Jetzt können wir verstehen, welche Unterkonzepte der “Klasse” zum Programmieren mit Löchern vorgesehen sind.

Was haben wir gelernt?

- Codeverschmutzung wird vermieden durch Vererbung
  - Vererbung erlaubt die *Wiederverwendung* von Merkmalen aus Oberklassen,
  - Einfache Vererbung führt zu Vererbungshierarchien
- Schnittstellen als auch abstrakte Klassen erlauben es, Anwendungsprogrammierern Struktur vorzugeben
  - Sie definieren “Haken”, in die Unterklassen konkrete Implementierungen schieben
  - Schnittstellen sind vollständig abstrakte Klassen
- Polymorphie erlaubt dynamische Architekturen
  - Merkmalssuche (dynamic dispatch) löst die Bedeutung von Merkmalsnamen auf, in dem von den gegebenen Unterklassen aus aufwärts gesucht wird
  - Polymorphie benutzt Merkmalssuche, um die Mehrdeutigkeit von Namen in einer Vererbungshierarchie aufzulösen
  - Monomorphe Aufrufe sind schneller, weil Merkmalssuche eingespart werden kann
- Die Klasse `Object` enthält als implizite Oberklasse der Java-Bibliothek gemeinsam nutzbare Funktionalität für alle Java-Klassen
- Generische Klassen ermöglichen typsichere Wiederverwendung von Code über Typ-Parameter ⇒ der Compiler meldet mehr Fehler

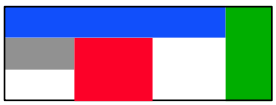
Warum ist das wichtig?

- Wiederverwendung ist eines der Hauptprobleme des Software Engineering
  - In einem Programm
  - Von Projekt zu Projekt
  - Von Produkt zu Produkt (Produktfamilien, Produktlinien)
- Vererbung, generische Klassen und Rollenklassen können Code-Replikate und Code-Explosion weitgehend vermeiden
  - Der Test von Software wird systematisiert
- Wiederverwendung ist das Hauptmittel der Softwarefirmen, um profitabel arbeiten zu können:
  - Schreibe und teste einmal und wiederverwende oft
  - Alle erfolgreichen Geschäftsmodelle von Softwarefirmen basieren auf Wiederverwendung (Produktlinien, Produktmatrix)
  - Ohne Wiederverwendung kein Verdienst und Überleben als Softwarefirma
- Firmen, die Wiederverwendung beherrschen, können neue Produkte sehr schnell erzeugen (reduction of time-to-market)
  - und sich an wechselnde Märkte gut anpassen
- Firmen mit guter Wiederverwendungstechnologie leben länger

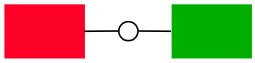
Verständnisfragen

- Geben Sie eine Begriffshierarchie der Methodenarten an. Welche könnten Sie sich noch denken?
- Geben Sie eine Begriffshierarchie des Klassenbegriffs an. Welche Klassenarten kennen Sie? Wie spezialisieren sie sich?
- Erweitern Sie die Vererbungshierarchie der Universitätsangehörigen um den Rektor und den Pedell (s. Wikipedia). Wo müssen sie eingeordnet werden?
- Stellen Sie ein Polymorphie-Diagramm überdie Phasen Ihres Lebens auf.
- Welchen Polymorphie-Zyklus durchläuft der Steuerzahler unseres Beispiels?
- Kann eine Steuererklärung polymorph sein?
- Wie würden Sie ein Testprogramm für ein polymorphes Objekt aus einem Polymorphie-Diagramm heraus entwickeln?

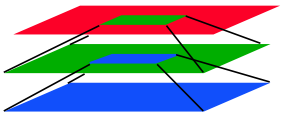
Prinzipielle Vorteile von Objektorientierung



Zuständigkeitsbereiche



Klare Schnittstellen



Hierarchie

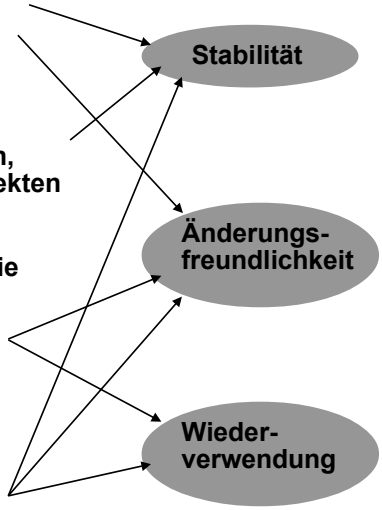
Baukastenprinzip

Lokalität: Lokale Kapselung von Daten und Operationen, gekapselter Zustand

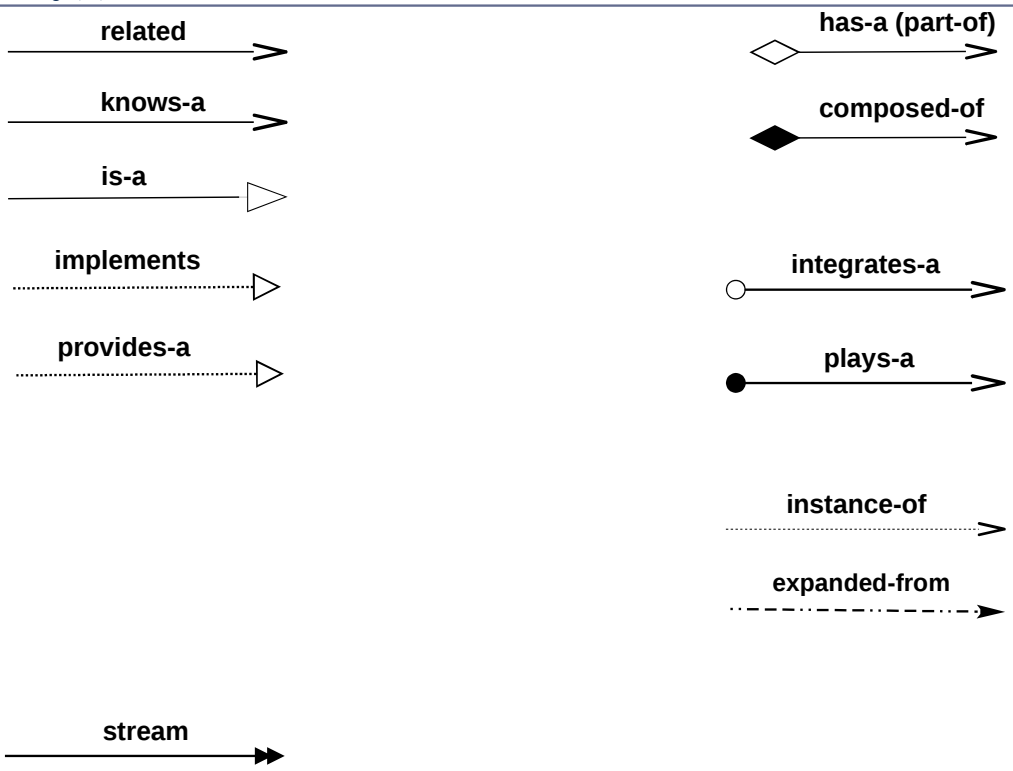
Typen und Typsicherheit
Definiertes Objektverhalten, Nachrichten zwischen Objekten

Vererbung und Polymorphie (Spezialisierung), Wiederverwendung
Klassenschachtelung

Benutzung vorgefertigter Klassenbibliotheken (Frameworks), Anpassung durch Spezialisierung (Vererbung)

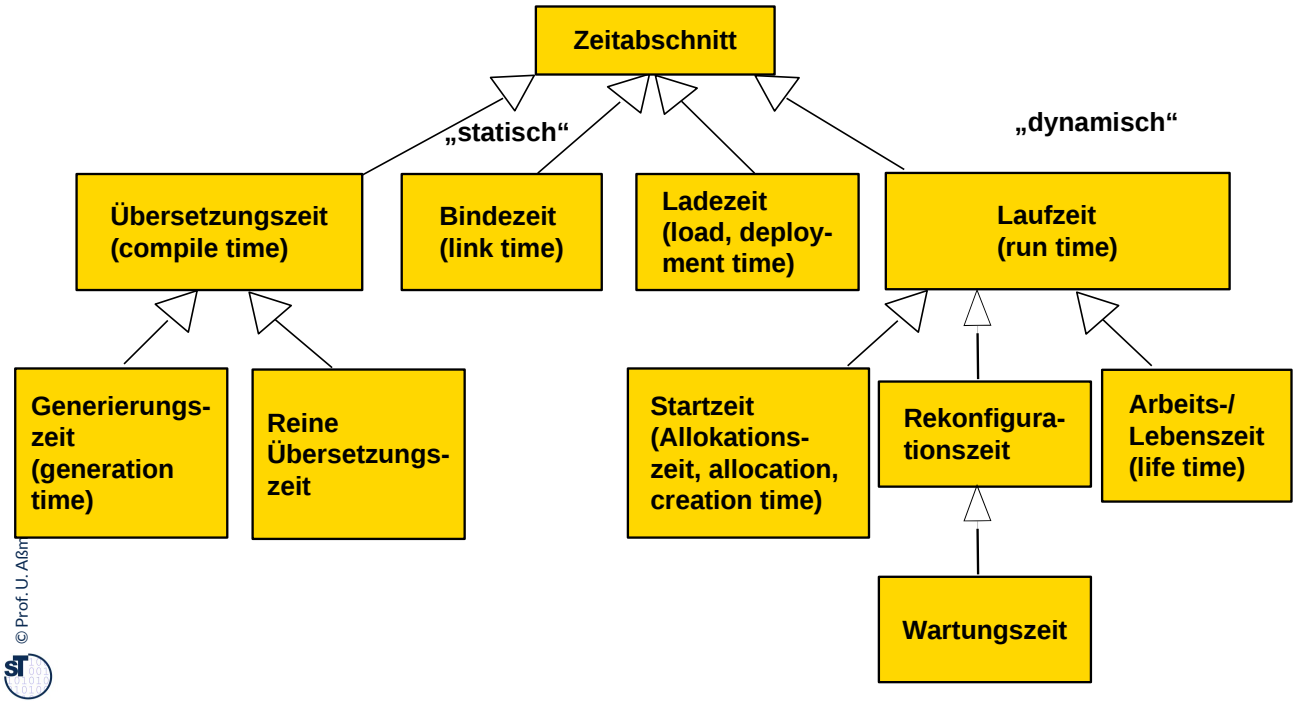


Q10: Relationen



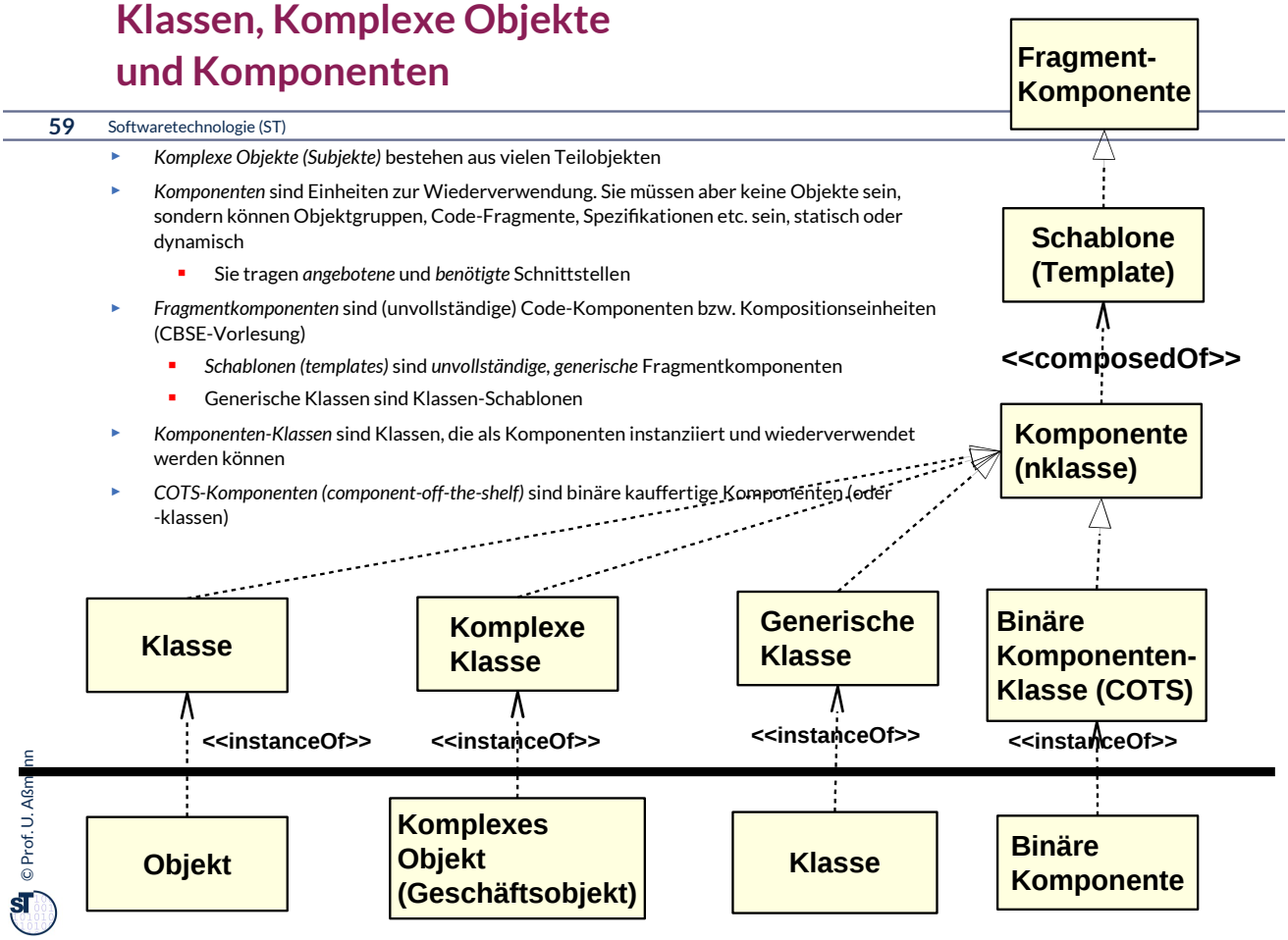
Bsp. Begriffshierarchie: Verschiedene Zeiten im Lebenszyklus einer Software

- Übersetzungszeit kennt hauptsächlich *Klassen*; Laufzeit kennt hauptsächlich *Objekte*



Klassen, Komplexe Objekte und Komponenten

- *Komplexe Objekte (Subjekte)* bestehen aus vielen Teilobjekten
- *Komponenten* sind Einheiten zur Wiederverwendung. Sie müssen aber keine Objekte sein, sondern können Objektgruppen, Code-Fragmente, Spezifikationen etc. sein, statisch oder dynamisch
  - Sie tragen *angebotene* und *benötigte* Schnittstellen
- *Fragmentkomponenten* sind (unvollständige) Code-Komponenten bzw. Kompositionseinheiten (CBSE-Vorlesung)
  - *Schablonen (templates)* sind *unvollständige, generische* Fragmentkomponenten
  - Generische Klassen sind Klassen-Schablonen
- *Komponenten-Klassen* sind Klassen, die als Komponenten instanziiert und wiederverwendet werden können
- *COTS-Komponenten (component-off-the-shelf)* sind binäre kauffertige Komponenten (oder -klassen)



Bemerkung

- Wir benutzen i.A. mehrere Darstellungen für Klassen- und Objektdiagramme
  - UML-Strukturdiagramme
  - Tripel-Sätze: SPO mit infix-Prädikaten
  - Venn-Diagramme (mengenorientierte Sicht)
- Venn-Diagramme
  - Sie können sowohl die Zugehörigkeit eines Objekts zu einer Klasse als auch Vererbung zwischen Klassen mit einem *Einkapselung* einheitlich beschreiben (mengentheoretischer und Ähnlichkeitsaspekt)
  - Sie können sowohl die statischen Beziehungen von Klassen, als auch die dynamischen Beziehungen von Objekten und Klassen beschreiben
  - Sie werden oft für die Spezifikation von Begriffshierarchien (Taxonomien, Ontologien) verwendet

Beispiel als Venn-Diagramm

- Vererbung kann auch durch Enthaltensein von Rechtecken ausgedrückt werden
  - Einfach-Vererbung ergibt ein Diagramm ohne Überschneidungen
- Für Objekt-Extents einer Klassenhierarchie gilt, dass jede Klasse die eigenen Objekte verwaltet
  - Der Objekt-Extent einer Oberklasse ergibt sich aus der Vereinigung der Extents aller Unterklassen (Person aus Professor und Student). Genau das drückt das Venn-Diagramm aus