11. Vererbung und Polymorphie

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 13-1.5, 29.04.13

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

Begleitende Literatur

- Das **Vorlesungsbuch** von Pearson: **Softwaretechnologie für Einsteiger.** Vorlesungsunterlage für die Veranstaltungen an der TU Dresden. Pearson Studium, 2009. 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. 2004.
  - Noch Exemplare in Buchhandlung vorhanden!
- 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

- Zuser Kap 7, Anhang A
- Störrle, Kap. 5.2.6, 5.6
- Java
  - Balzert LE 9-10
  - Boles Kap. 7, 9, 11, 12

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

Code-Explosion durch Copy-And-Paste-Programming: CAPP

- http://c2.com/cgi/wiki?CopyAndPasteProgramming
- http://en.wikipedia.org/wiki/Copy_and_paste_programming
- Codeverschmutzung durch CAPP: Nach einer Weile entdeckt man in einem gewachsenen System, das jede Menge Code repliziert wurde
  - Große Software kann 10-20% an *Replikaten (code clones)* enthalten *(Code-Explosion, code bloat)*
- Gründe für Copy-And-Paste-Programming:
  - **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

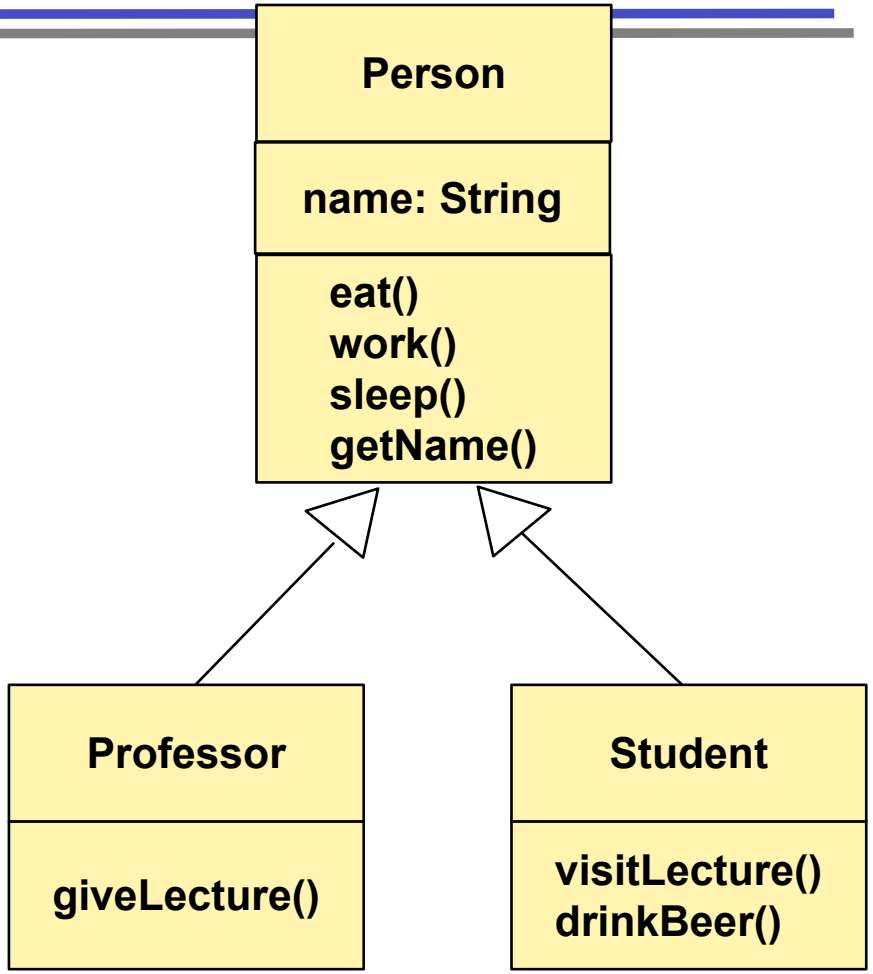
- Interessante Technik, Code-Replikate zu finden und dauerhaft zu verlinken:
  - Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In VL/HCC, pages 173-180. IEEE Computer Society, 2004.
  - http://harmonia.cs.berkeley.edu/papers/toomim-linked-editing.pdf
- 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

11.1 Vererbung zwischen Klassen

Ähnlichkeit von Klassen sollten in Oberklassen ausfaktorisiert werden

Einfache Vererbung

- **Vererbung**: Eine Klasse kann Merkmale von einer Oberklasse **erben**
  - Die Unterklasse ist damit ähnlich zu dem Elter und den Geschwistern
  - Vererbung drückt Gemeinsamkeiten aus
  - Vererbung stellt *is-a*-Beziehung her (<)
- Bei e**infacher Vererbung** hat jede Klasse nur *eine* Oberklasse
  - Dann ist die Vererbungsrelation ein Baum



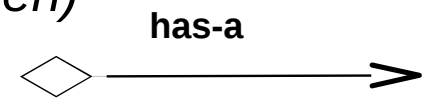
- Hier: Oberklasse Person enthält alle gemeinsamen Merkmale der Unterklasse
- Einfache textuelle Schreibweise mit infix-Prädikaten:

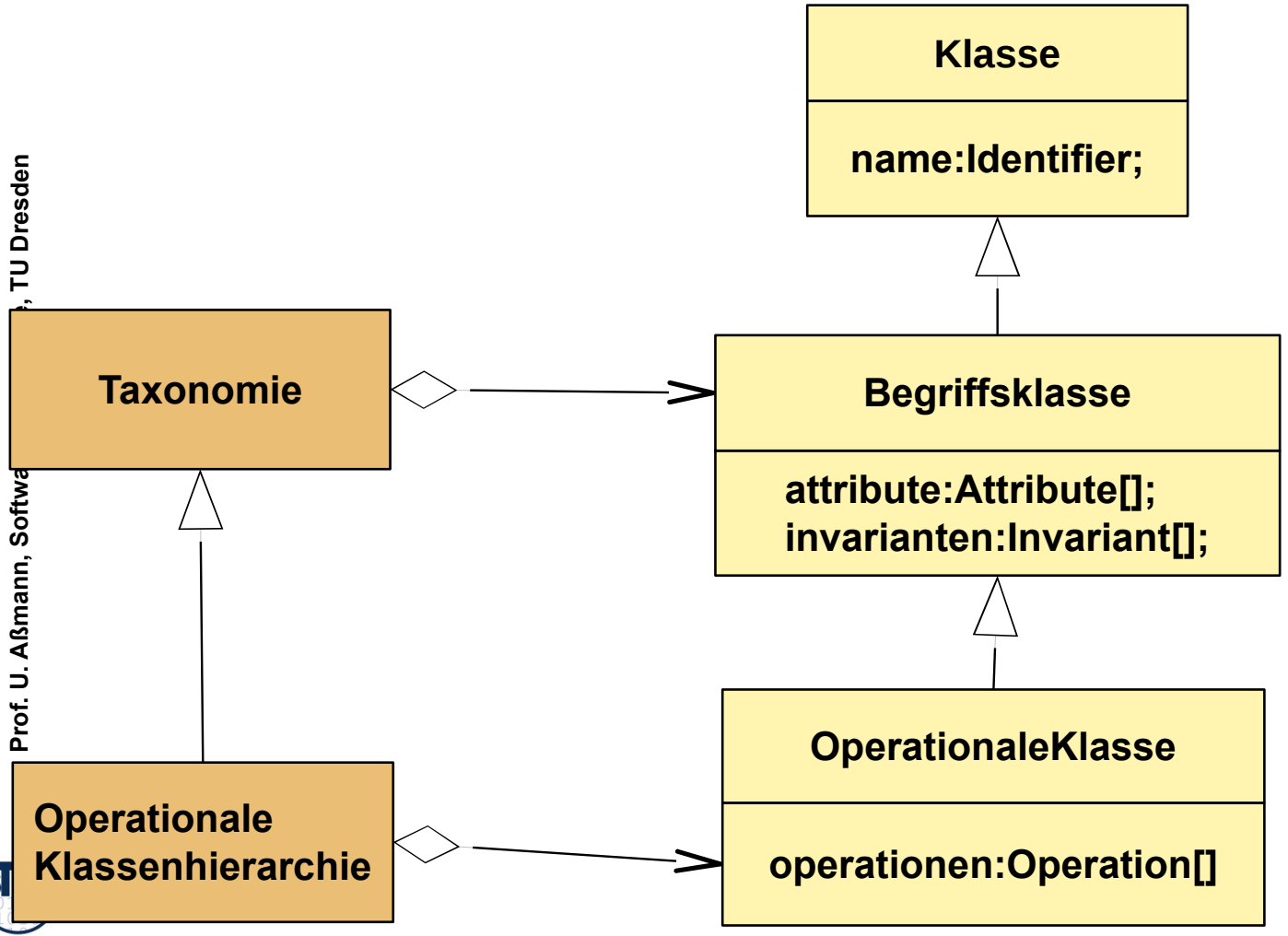
Professor < Person. Student < Person.

Professor is-a Person. Student is-a Person.

Begriffshierarchien (Taxonomien) nutzen einfache Vererbung

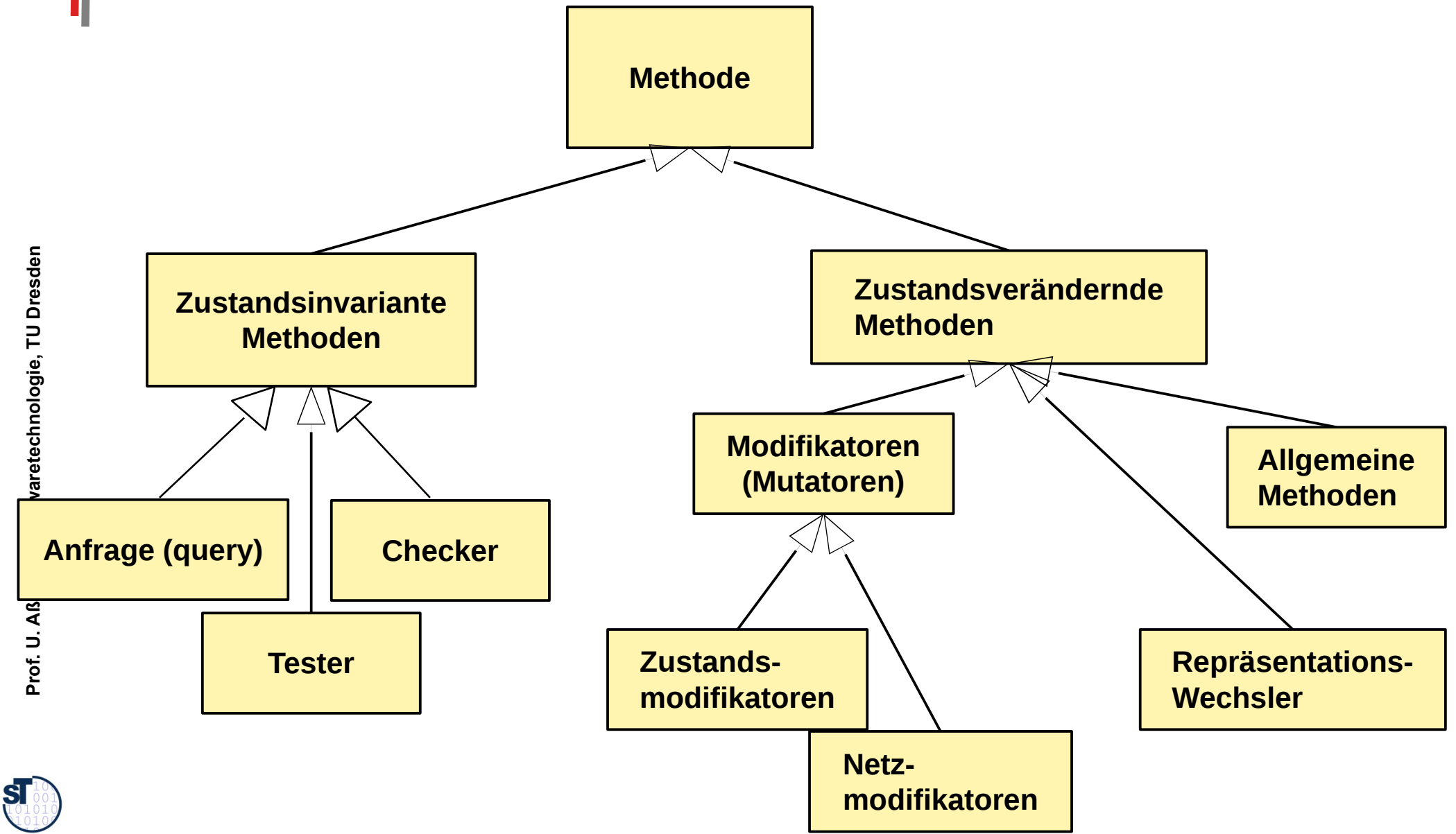
- Domänenmodelle werden oft durch Klassifikation ermittelt
- Klassifikationen führen zu *Begriffshierarchien (Taxonomien)*
  - *Begriffsklassen* besitzen nur Attribute und Invarianten
  - *Operationale Klassen* auch Operationen





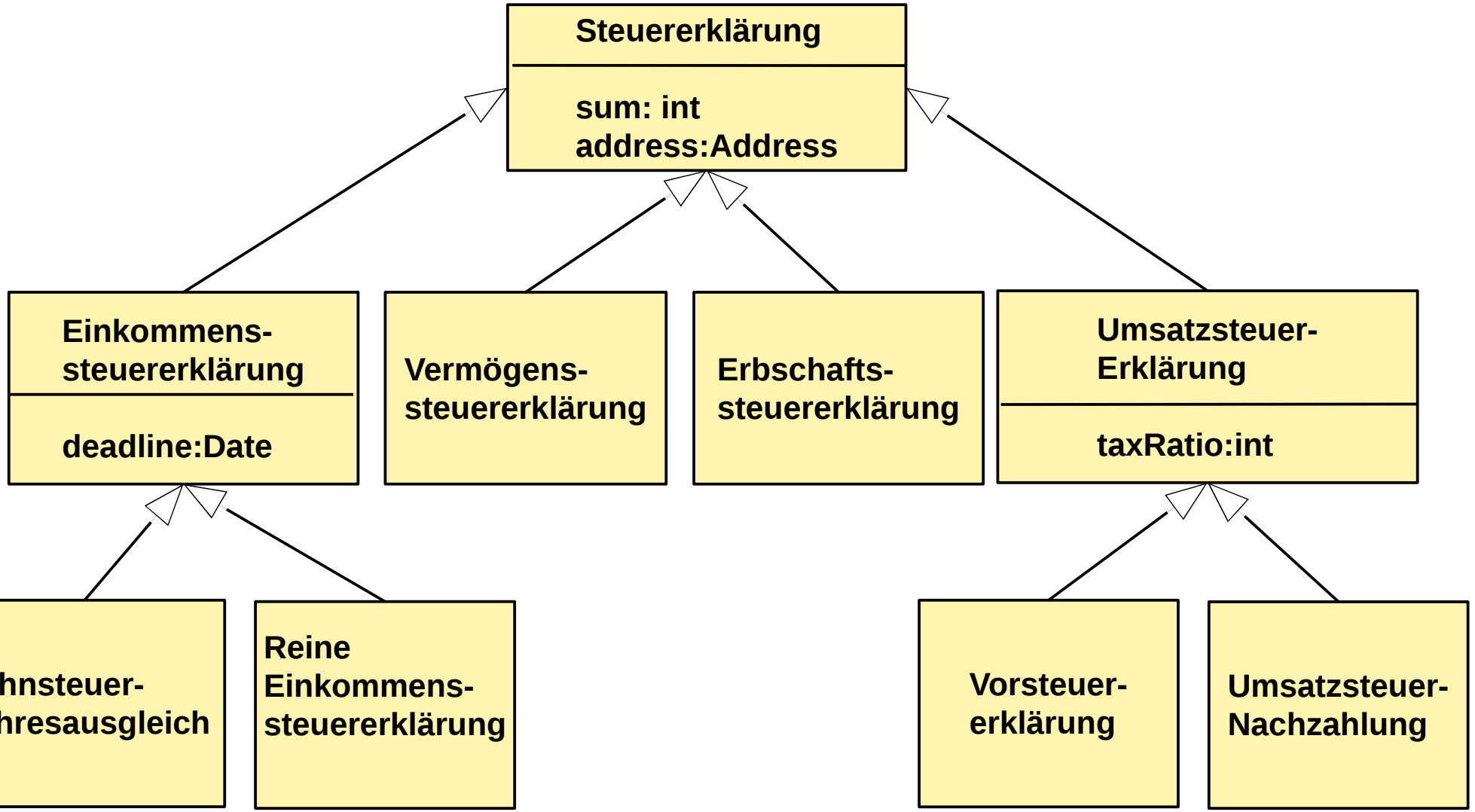
Bsp: Begriffshierarchie (Taxonomie) der Methodenarten

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



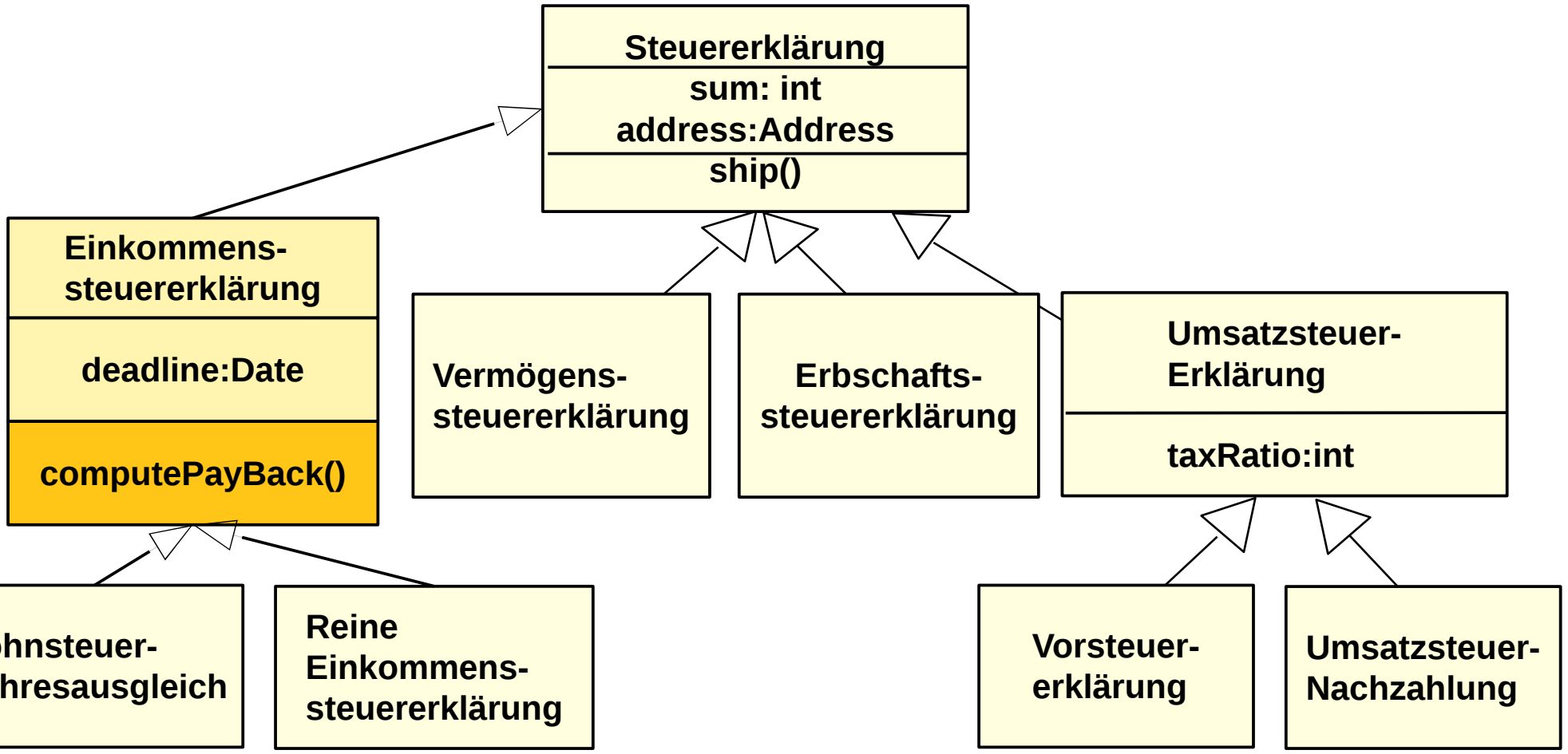
Bsp. Taxonomie der Steuererklärungen

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



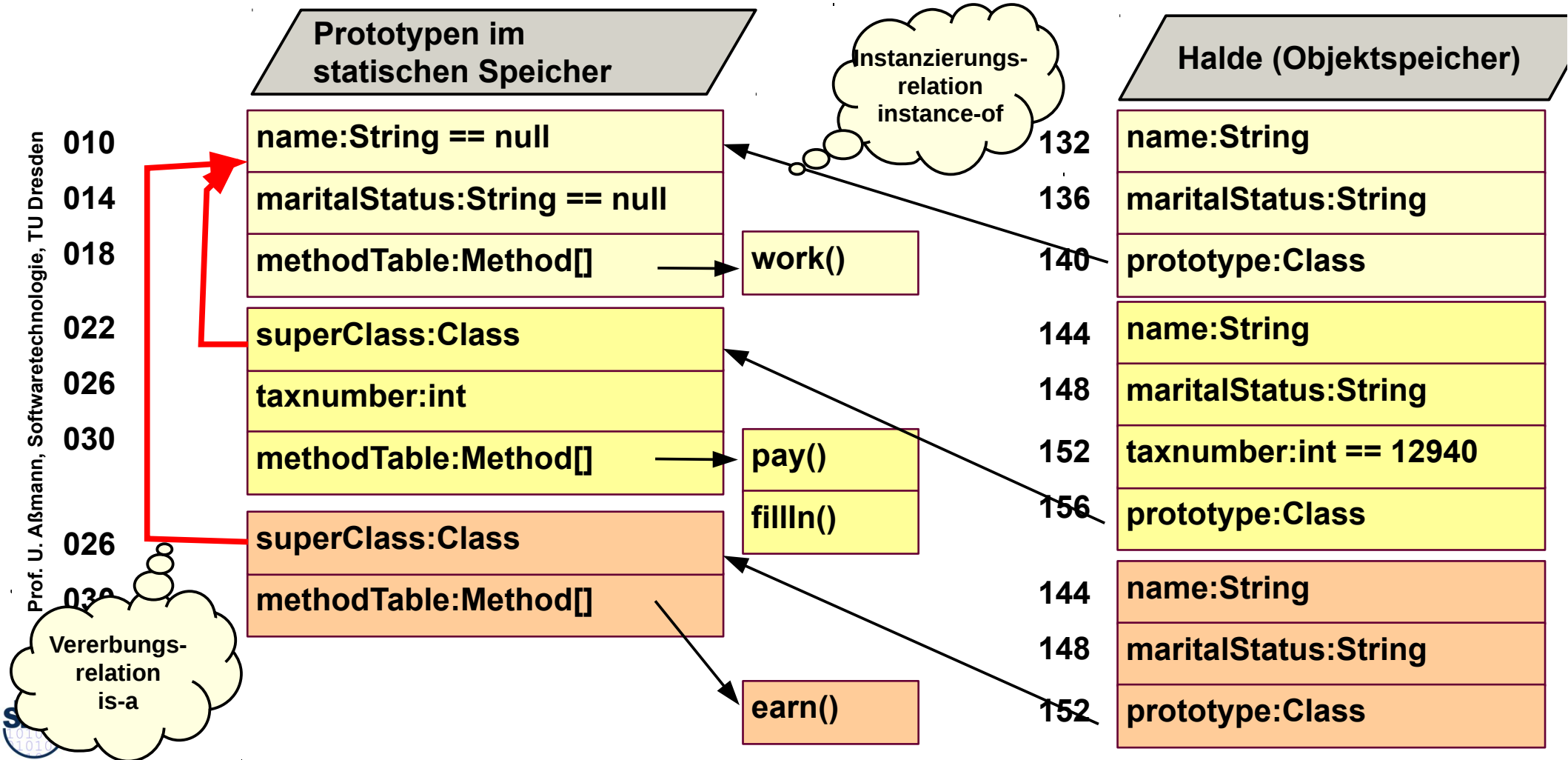
Bsp. Erweiterung einer Begriffshierarchie hin zu operationalen Klassenhierarchie

- 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.



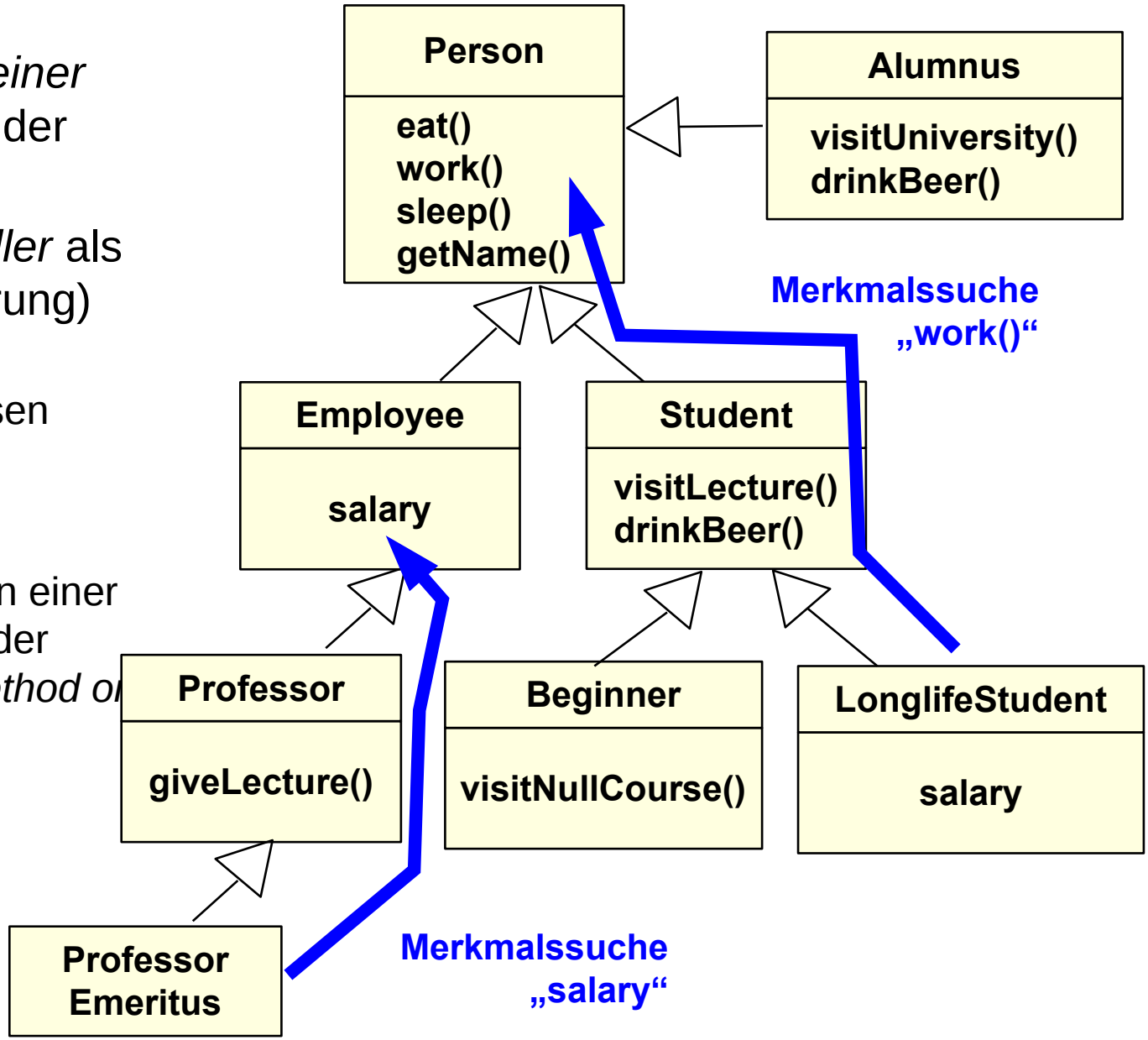
Vererbung im Speicher

- 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*



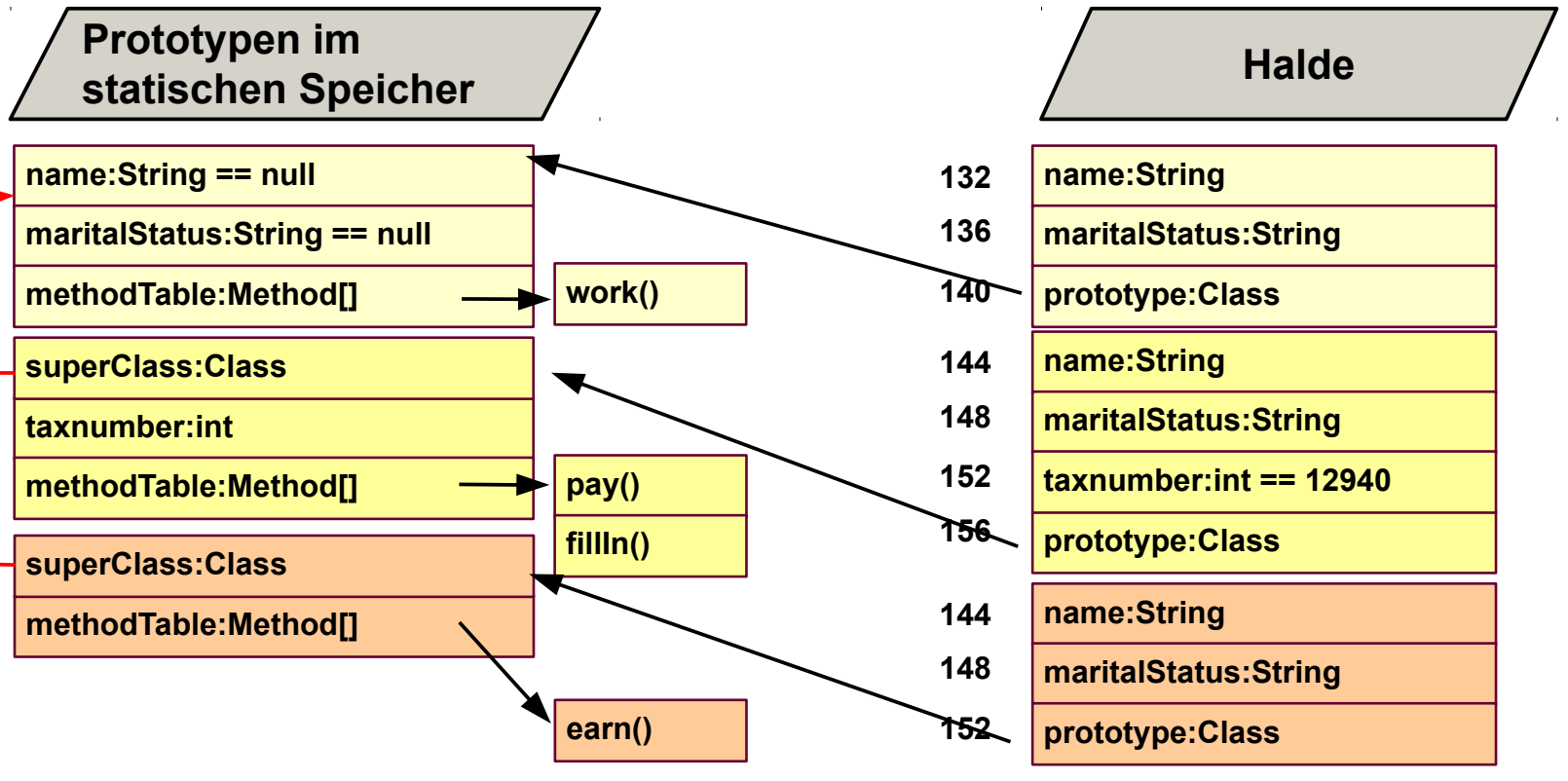
Merkmalssuche im Vererbungsbaum

- 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)*



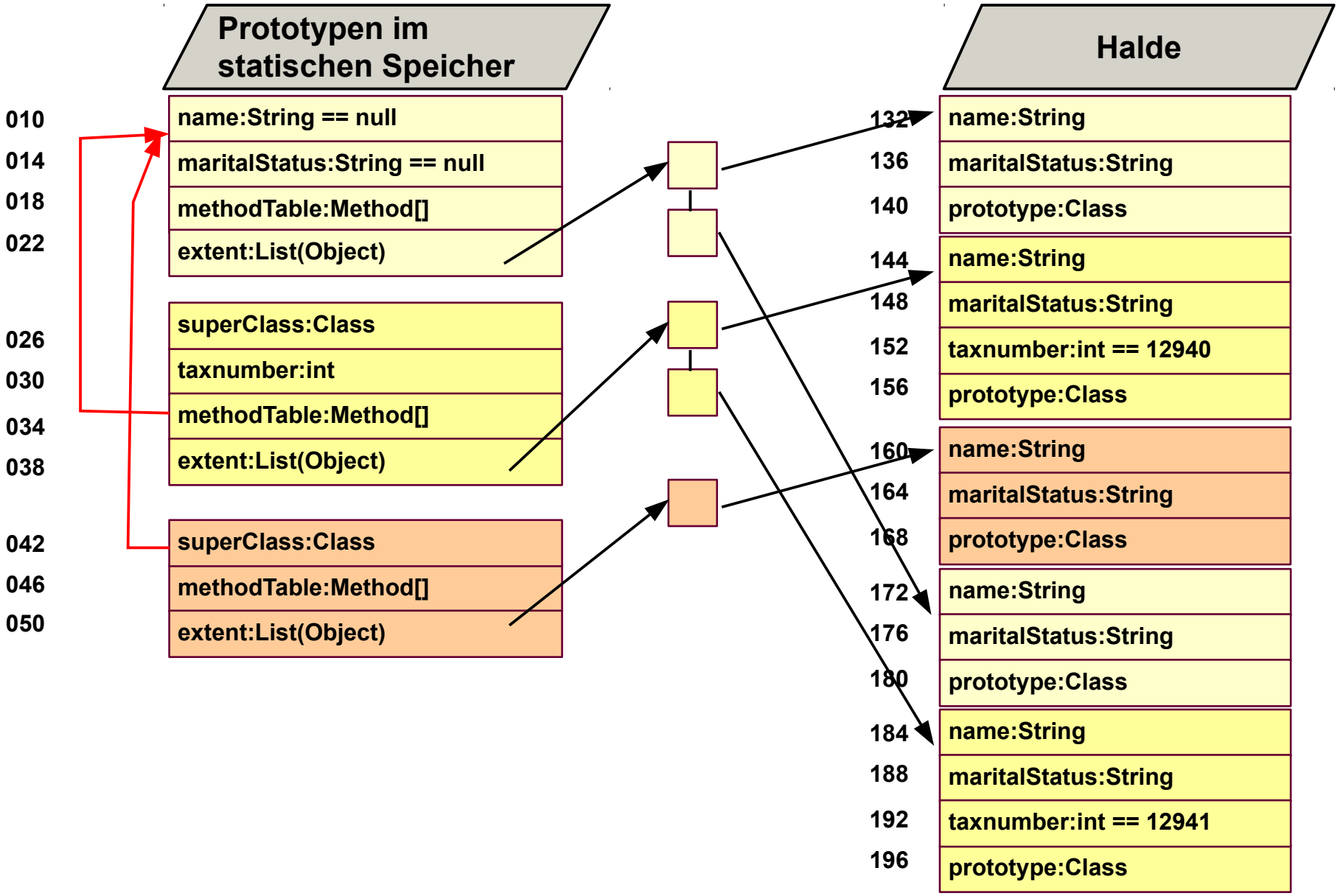
Merkmalssuche im Speicher - Beispiele

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"*



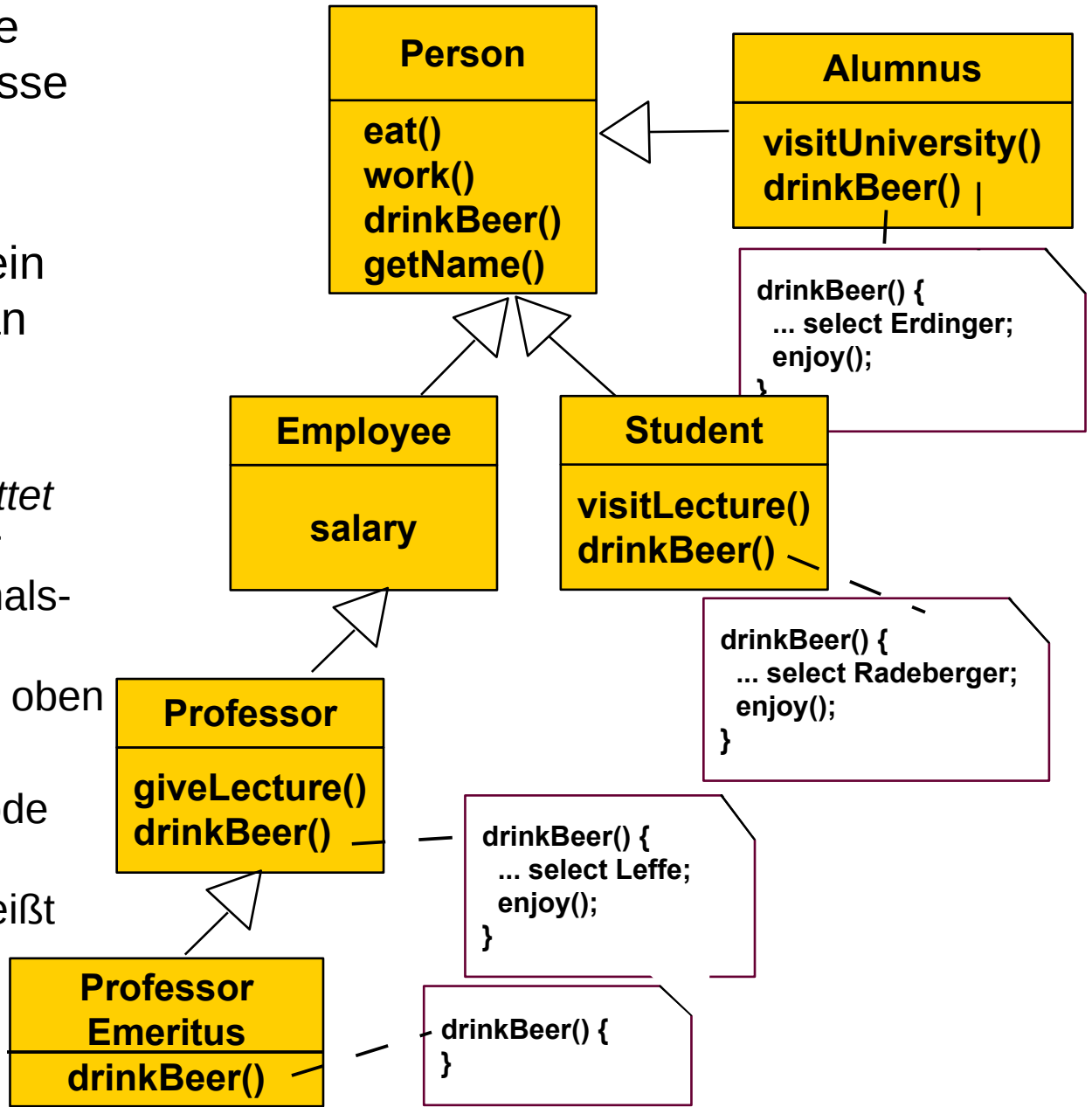
Objekt-Extent im Speicher, mit Vererbung

- 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*

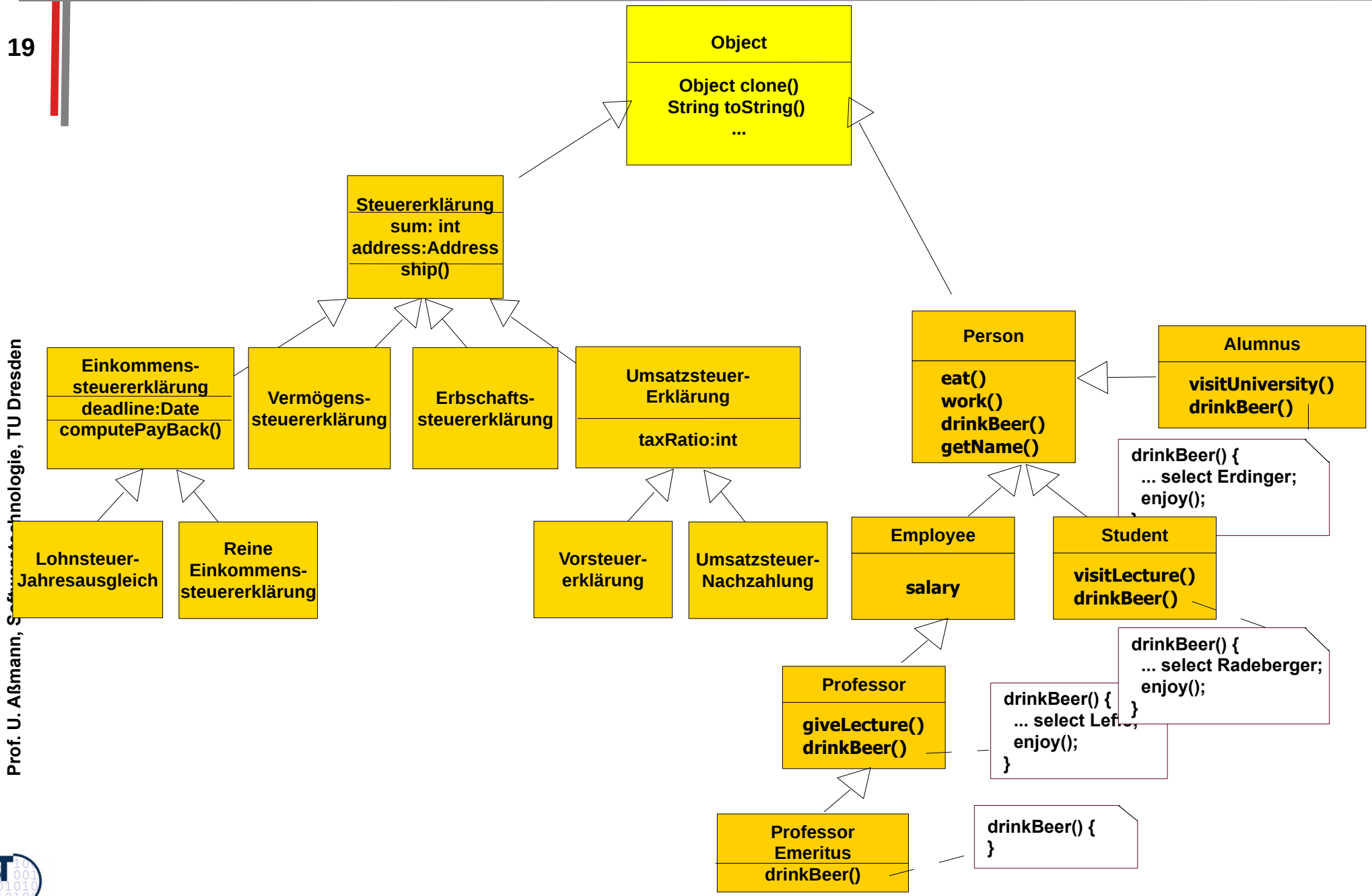


Die oberste Klasse "Object"

- **java.lang.Object:** allgemeine Eigenschaften aller Klassen und Objekte.
  - Jede Klasse ist Unterklasse von Object (“extends Object”).
  - Diese Vererbung ist *implizit* (d.h. man kann “extends Object” weglassen).
- 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

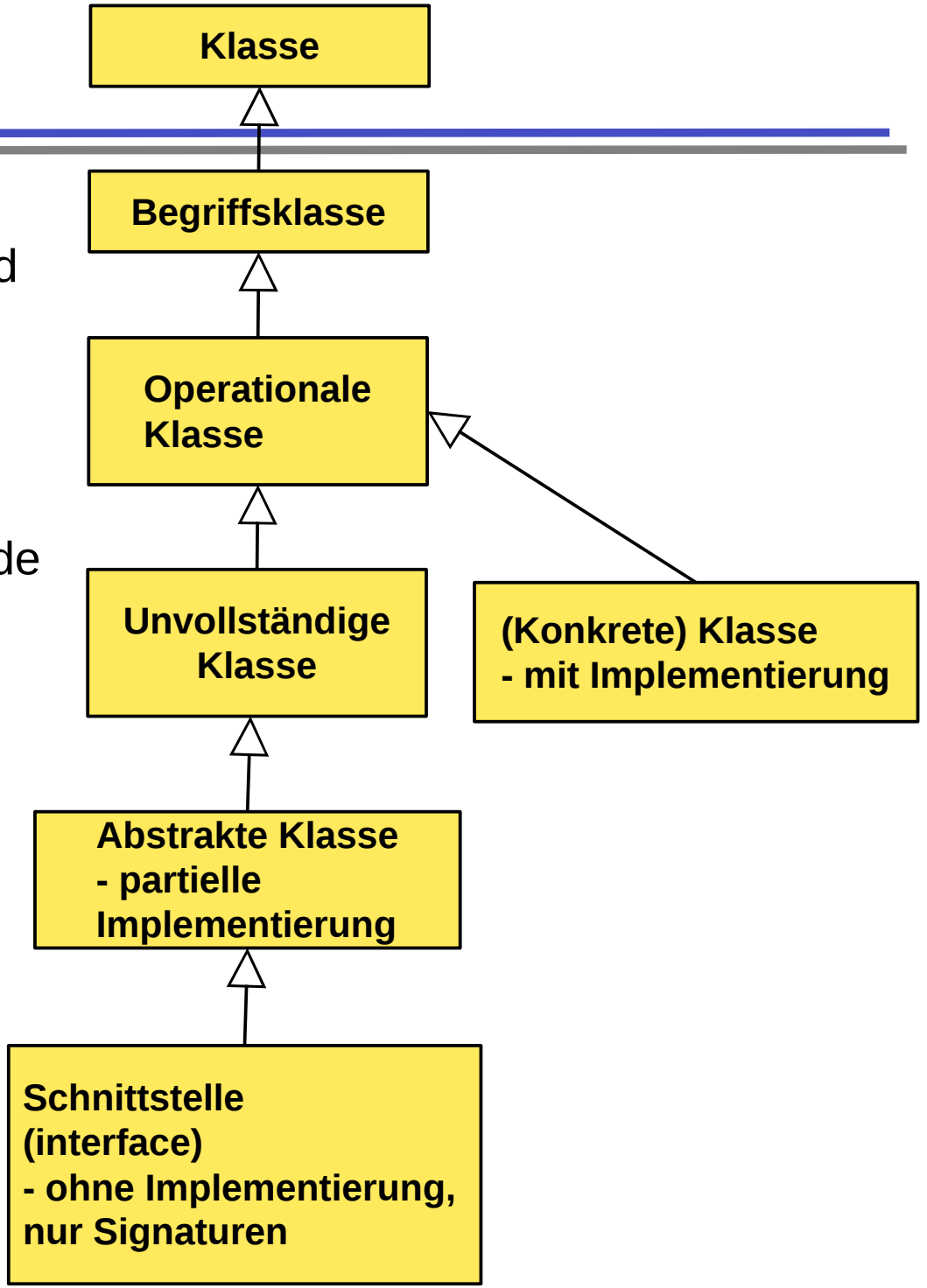


11.2 Schnittstellen und Abstrakte Klassen

Typen können verschiedene Formen annehmen

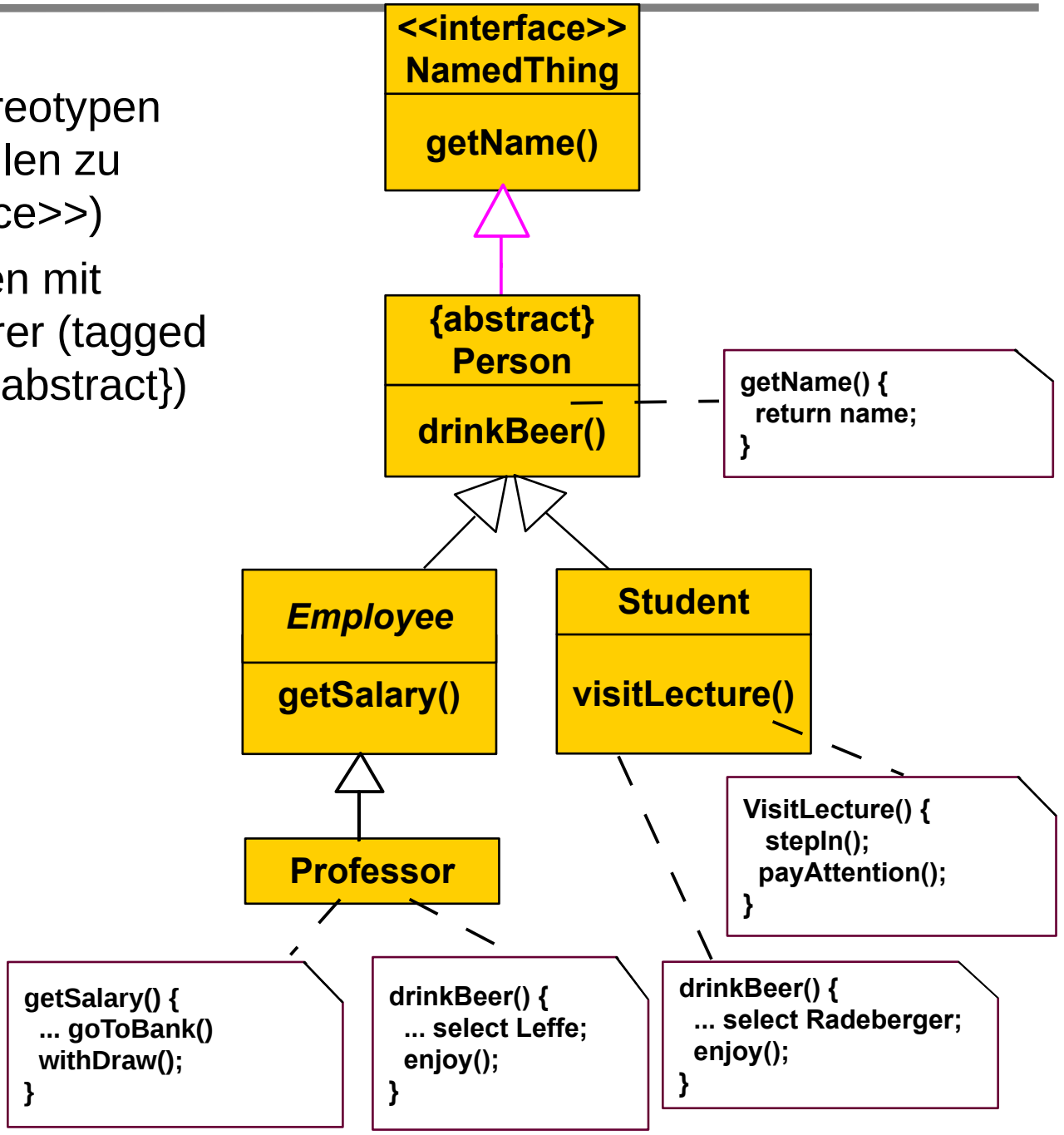
Begriffshierarchie von Klassen (Erw.)

- 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



Schnittstellen und Klassen in UML

- In UML werden sog. Stereotypen vergeben, um Schnittstellen zu kennzeichnen (<<interface>>)
- Abstrakte Klassen werden mit einem speziellen Markierer (tagged value) gekennzeichnet ({abstract}) oder *kursiv* gemalt



Schnittstellen und Klassen in Java


```
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(); }
}
```

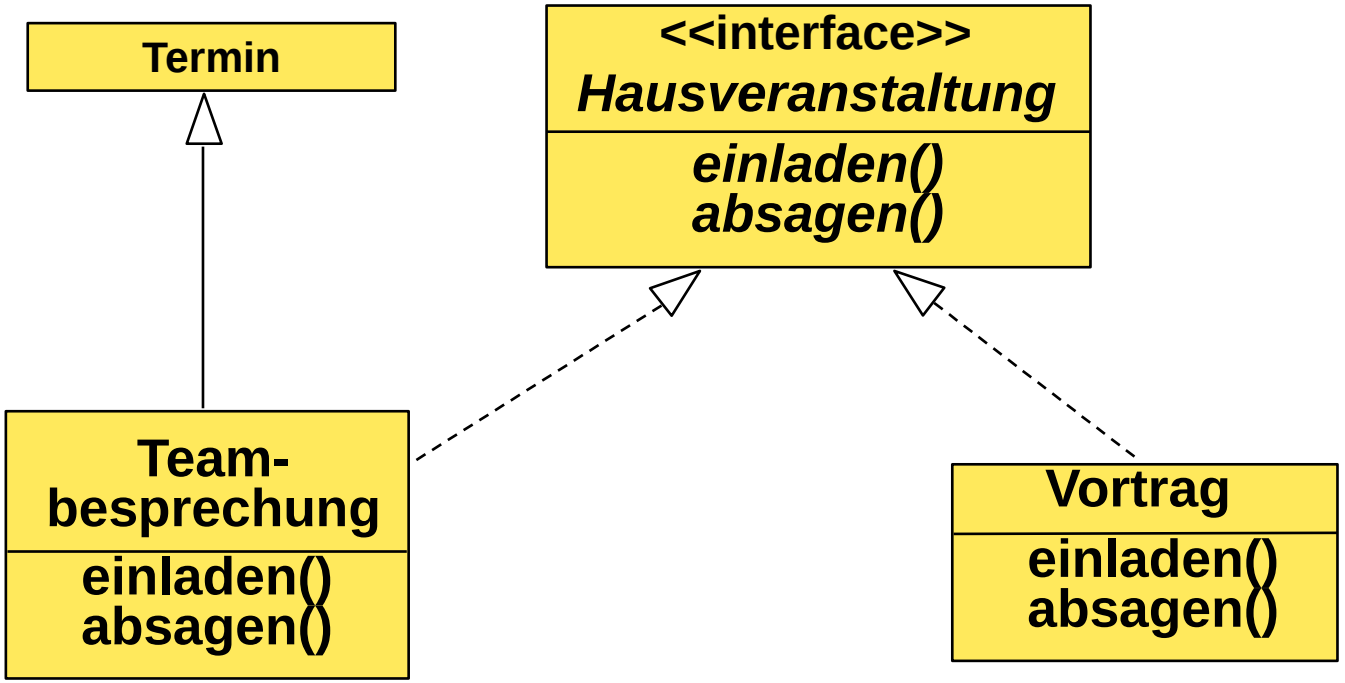
Abstrakte Klassen und Abstrakte Operationen


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

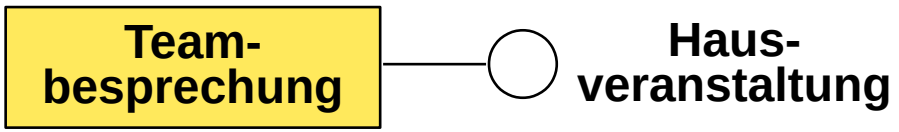
```
class Teambesprechung
   extends Termin {
   ...
   boolean verschieben (Hour neu)    {
      boolean ok = abstimmen(neu, dauer);
      if (ok) {
         beginn = neu;
         raumFestlegen();
      };
      return ok;
   };
};
```

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

Einfache Vererbung von Typen durch Schnittstellen



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



Vergleich von Schnittstellen und abstrakte Klassen

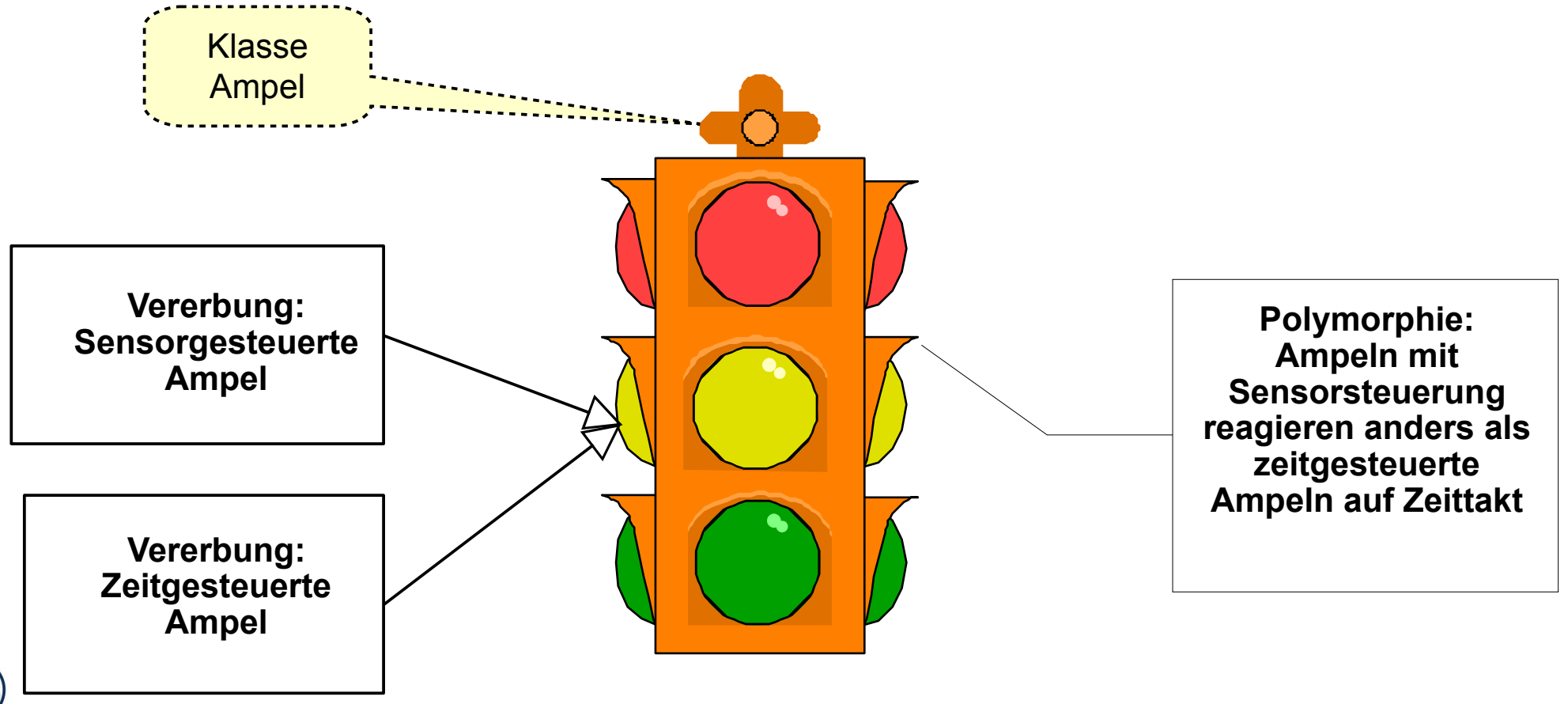
Abstrakte Klasse	Schnittstelle
Enthält Attribute und Operationen	Enthält nur Operationen (und ggf. Konstante)
Kann Default-Verhalten festlegen	Kann kein Default-Verhalten festlegen
Wiederverwendung von Schnittstellen und Code, aber keine Instanzbildung	Redefinition unsinnig
Default-Verhalten kann in 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

11.3. Polymorphie

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

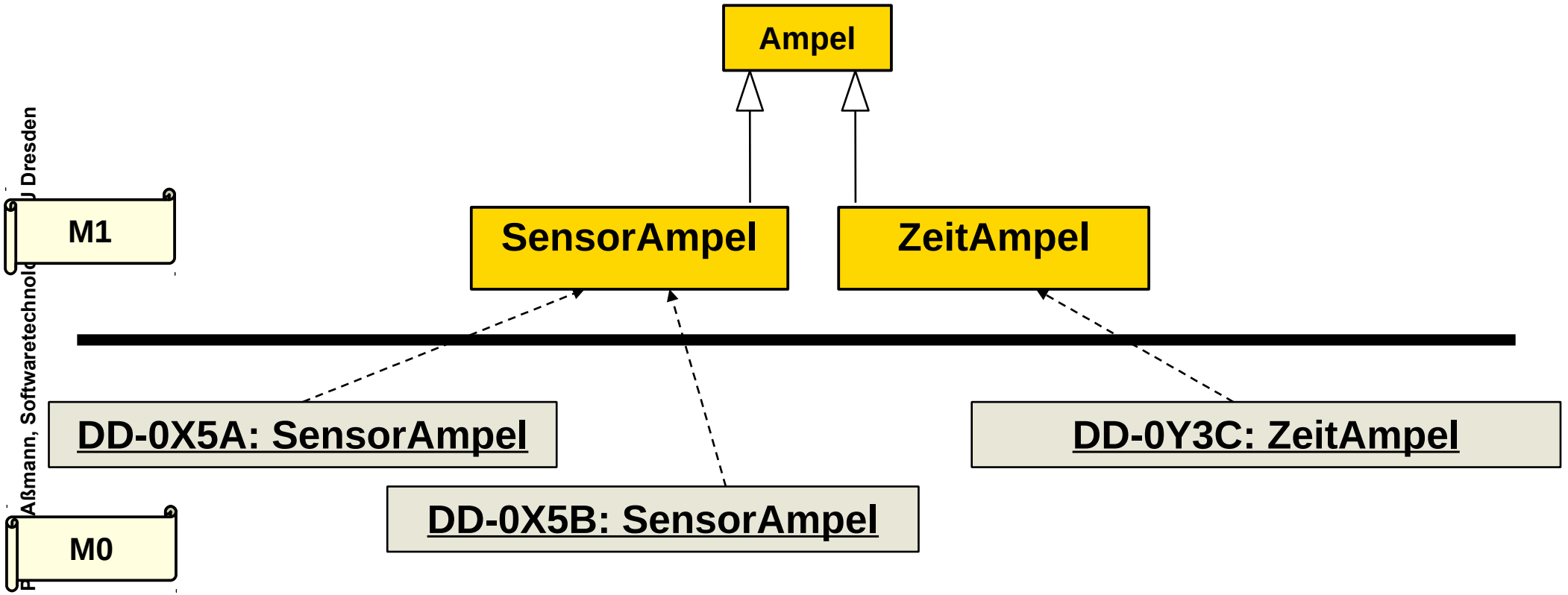
Vererbung und Polymorphie

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



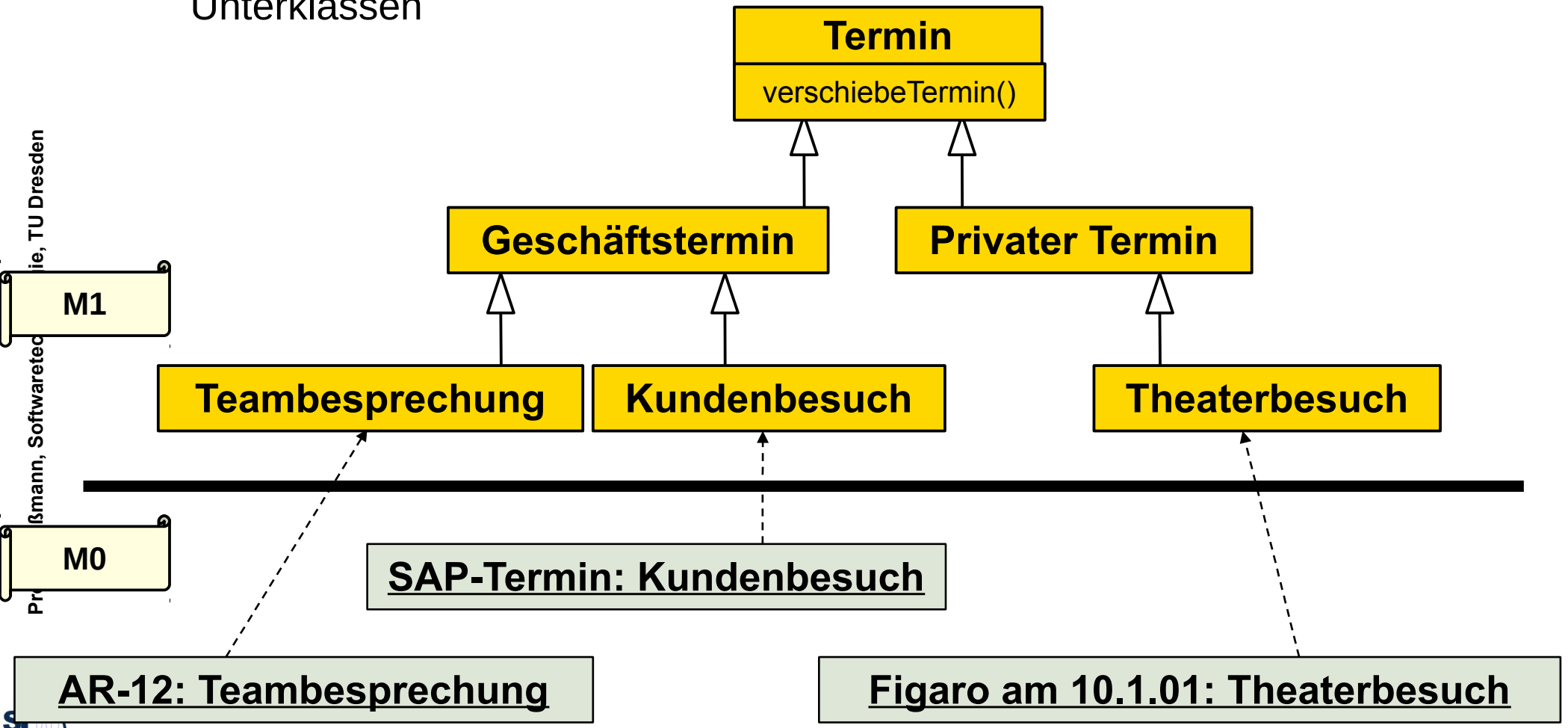
Beispiel: Ampel-Klasse und Ampel-Objekte

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



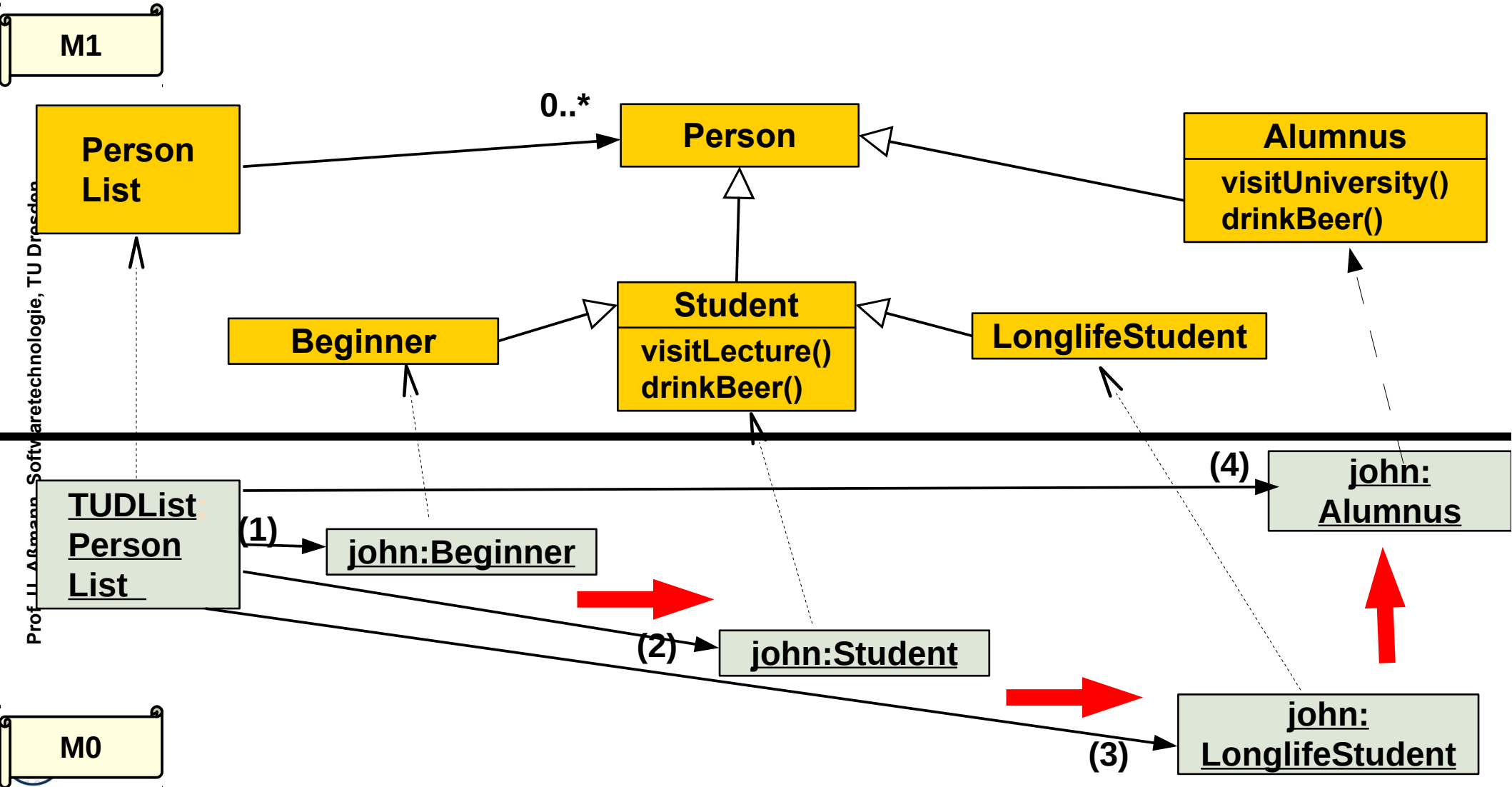
Beispiel: Termin-Klasse und Termin-Objekte

- 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)

- Zur Laufzeit kann jedes Objekt einer Unterklasse ein Objekt einer Oberklasse *vertreten.* Das Objekt der Oberklasse ist damit *vielgestaltig (polymorphic).*



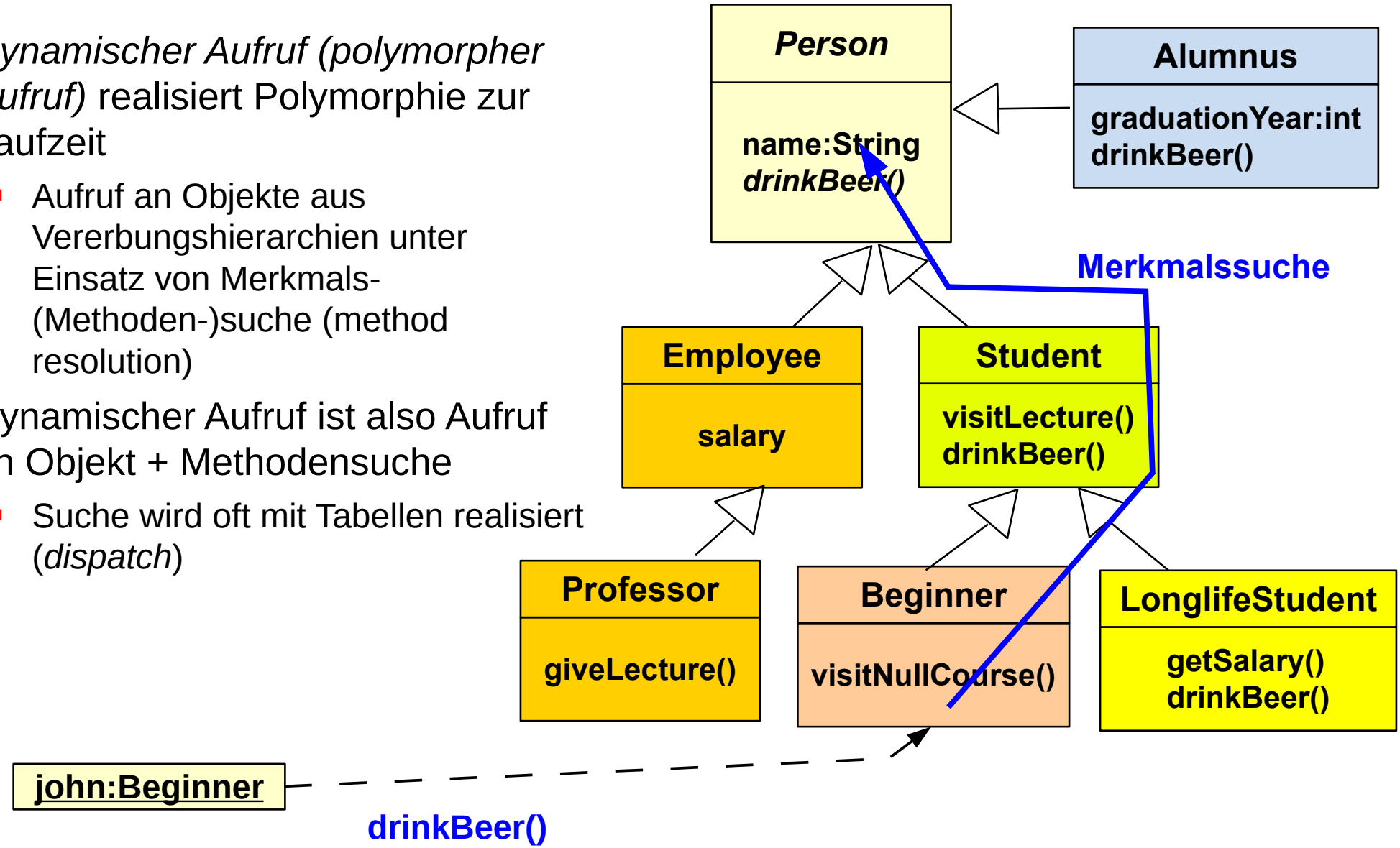
Wechsel der Gestalt (Polymorphie)

- Die genaue Unterklasse eines Objektes wird festgestellt
  - Beim Erzeugen (der Allokation) des Objekts (Allokationszeit, oft in der Aufbauphase des Objektnetzes)
  - 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 here?
....

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

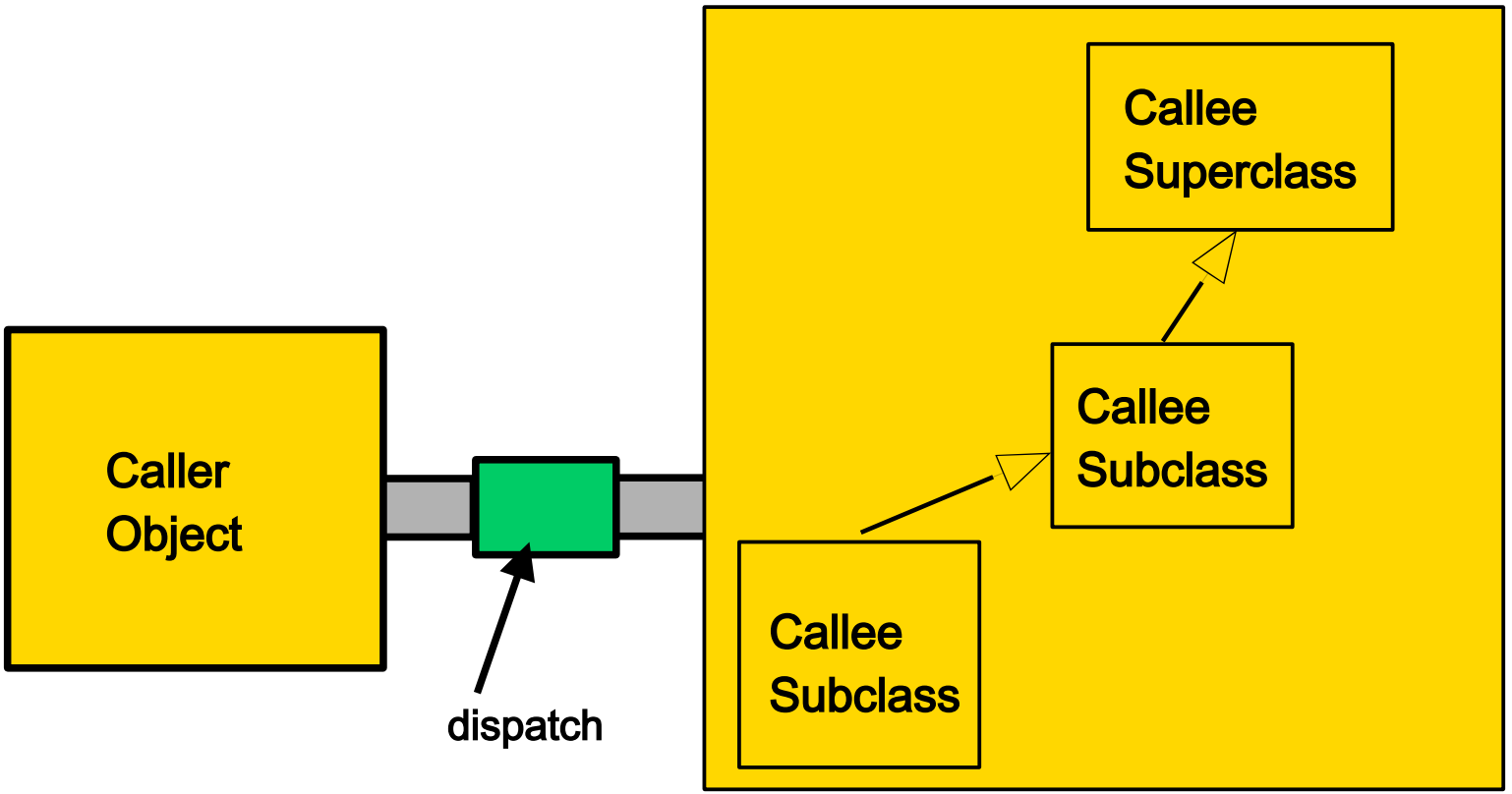
Dynamischer Aufruf (Dynamic dispatch)

- *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*)



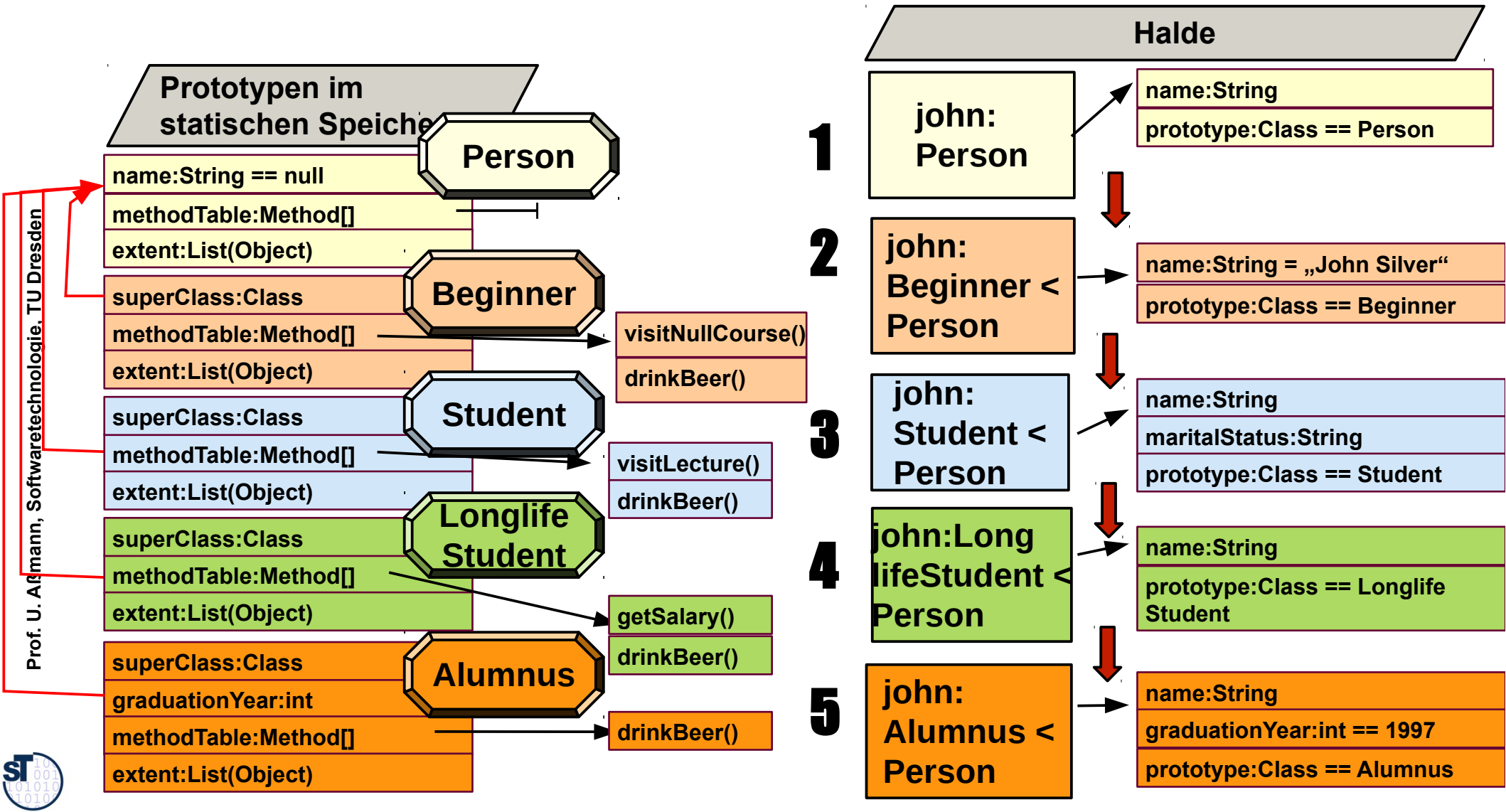
Dynamischer Aufruf (Dynamic Dispatch)

- 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)



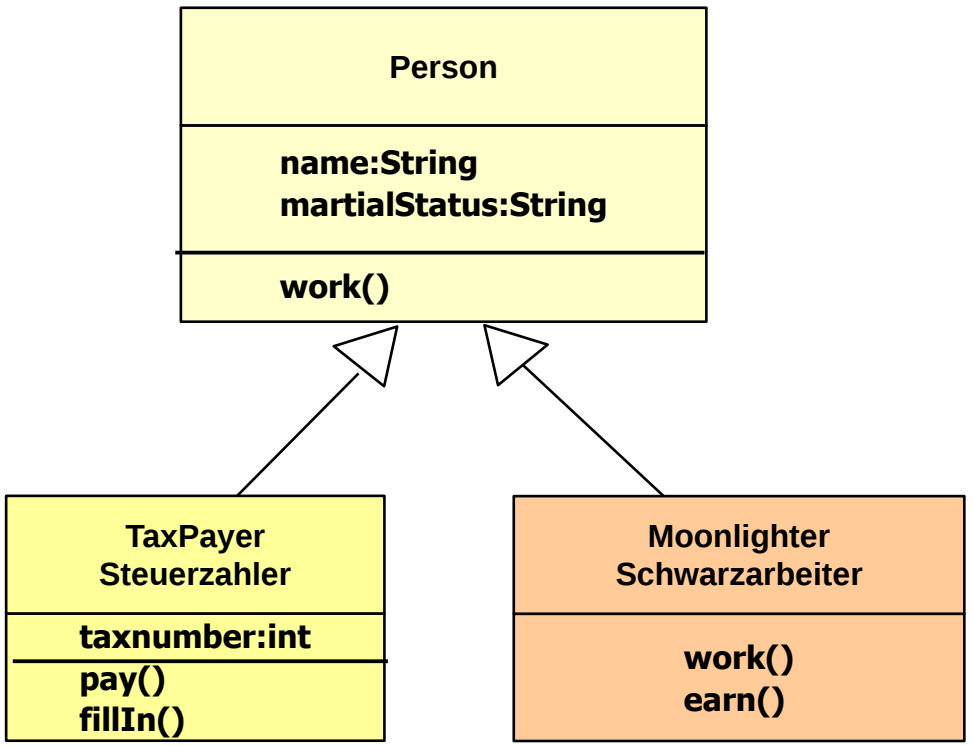
Was passiert beim polymorphen Aufruf im Speicher?

- 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?



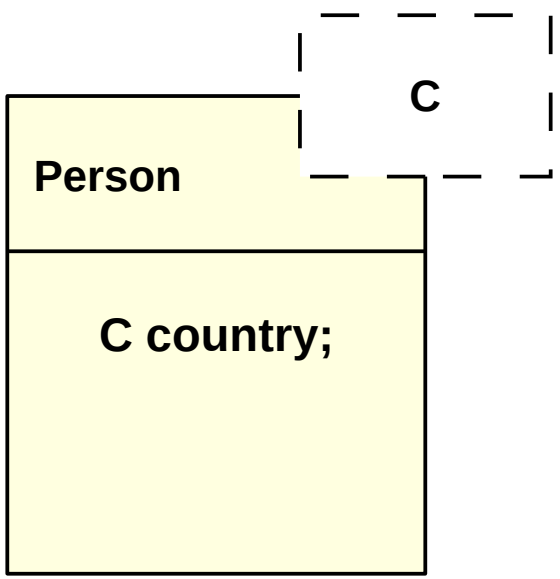
11.4. Generische Klassen (Klassenschablonen)

Parametrische Klassen ermöglichen typisierte Wiederverwendung von Code

Generische Klassen

Eine generische (parametrische) Klasse ist eine Klassenschablone, die mit einem oder mehreren Typparametern versehen ist.

- In UML



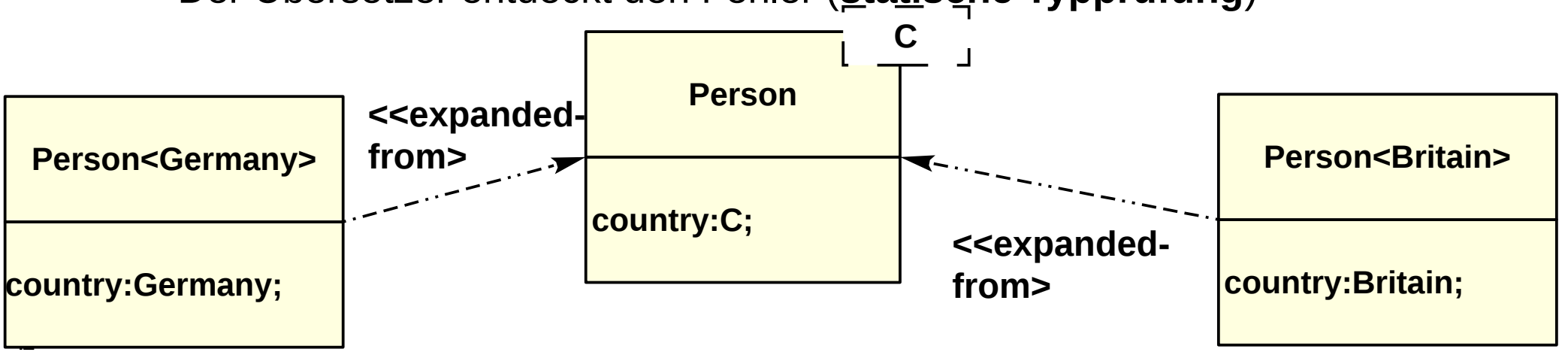
- In Java
  - Sprachregelung: “Person of C”

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

```
/* Type definition */
Person<Germany> egon;
Person<Britain> john;
```

Feinere statische Typüberprüfung

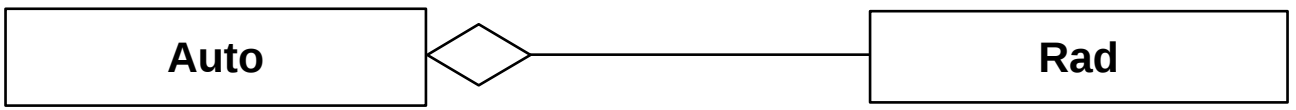
- Zwei Typen, die durch Parameterisierung aus einer generischen Klassenschablone entstanden sind, sind nicht miteinander kompatibel
- Der Übersetzer entdeckt den Fehler (**statische Typprüfung**)




```
/* 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*)

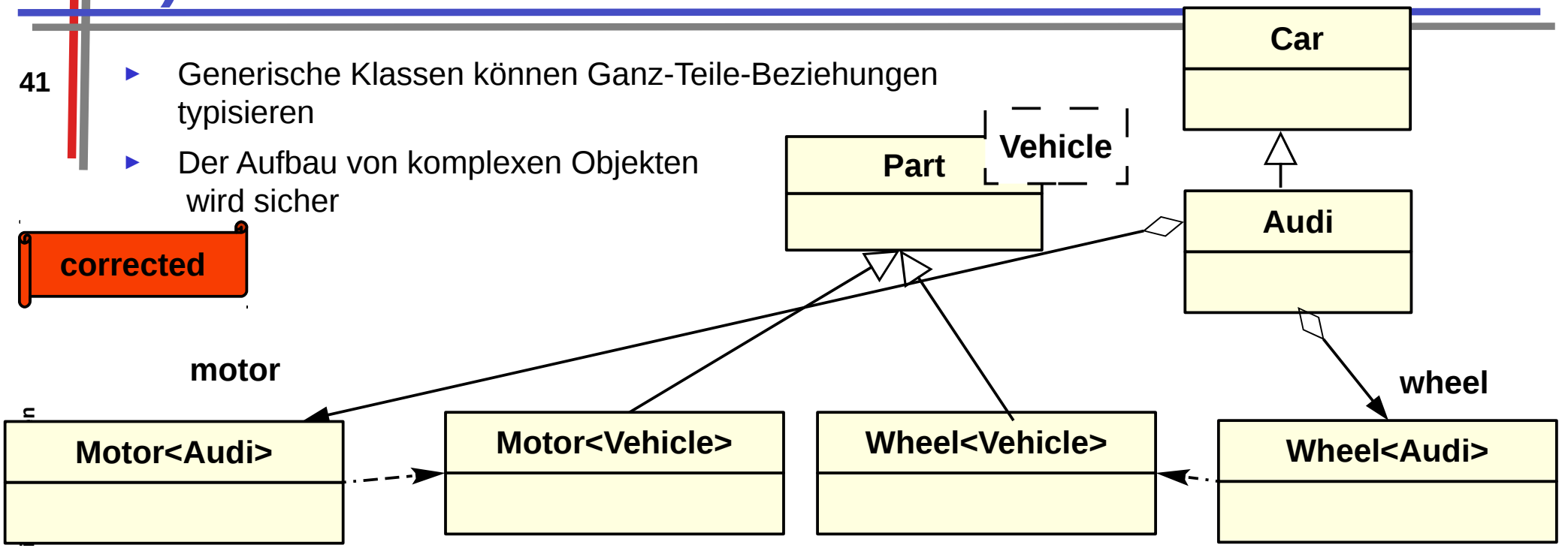


Lies: „Auto hat ein Rad“

Einsatzzweck: Typsichere Aggregation (has-a)

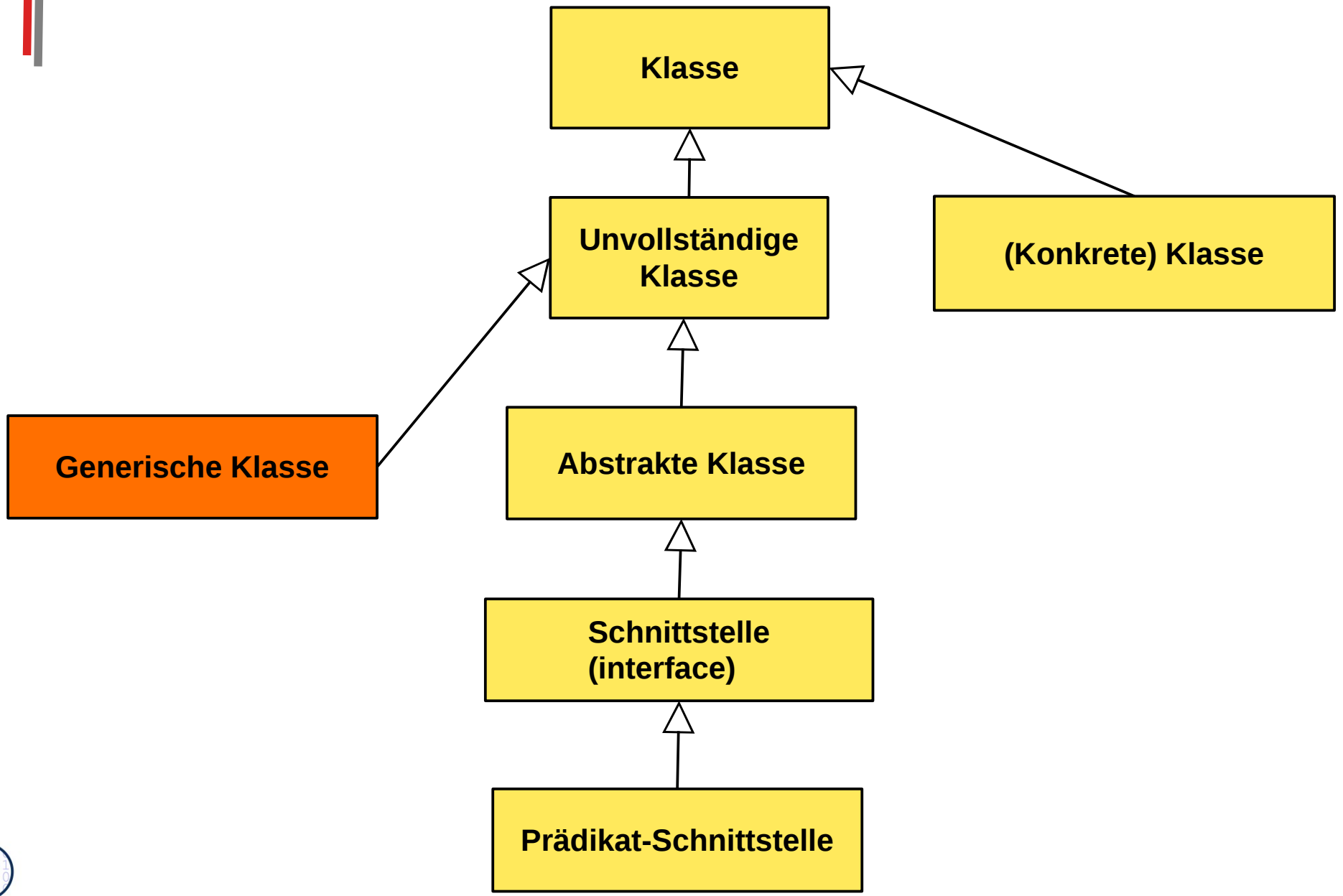
- Generische Klassen können Ganz-Teile-Beziehungen typisieren
- Der Aufbau von komplexen Objekten wird sicher

corrected




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

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

Begriffshierarchie von Klassen (Erweiterung)



11.5 Rollen-Polymorphie

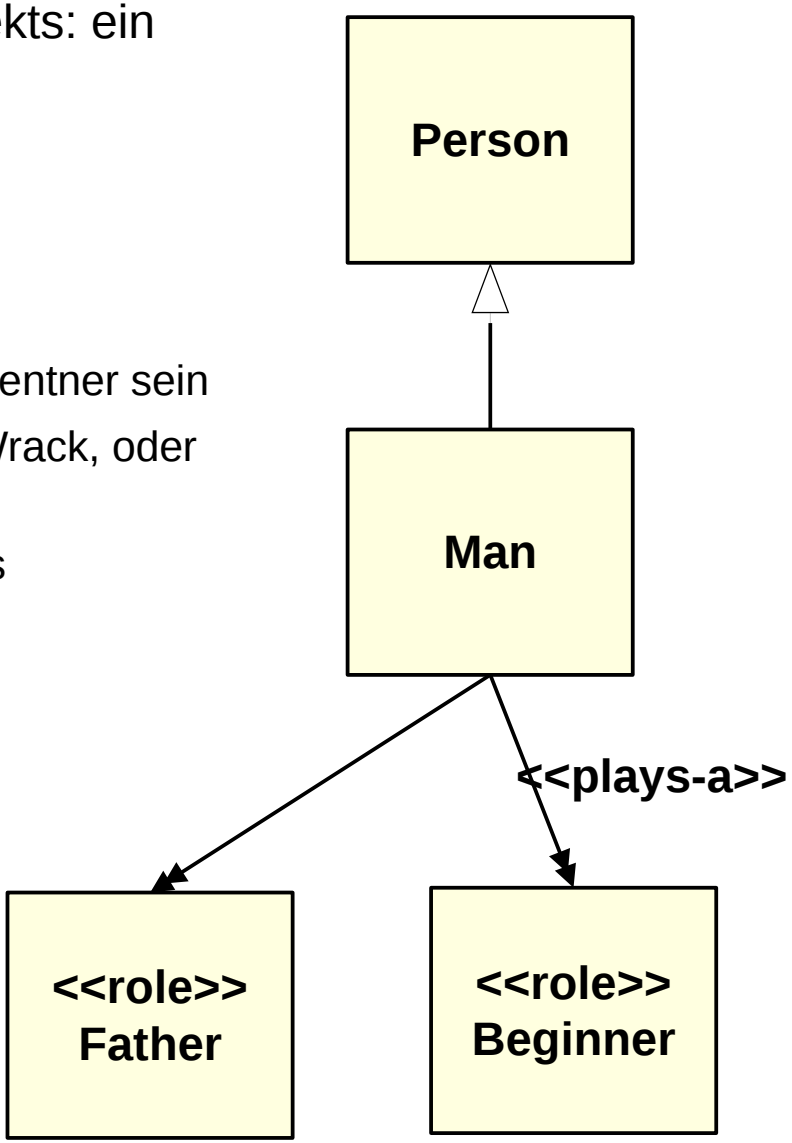
Polymorphie ergibt sich aus dem Wechsel von Rollen

(optional)

Programm PersonRoles.java

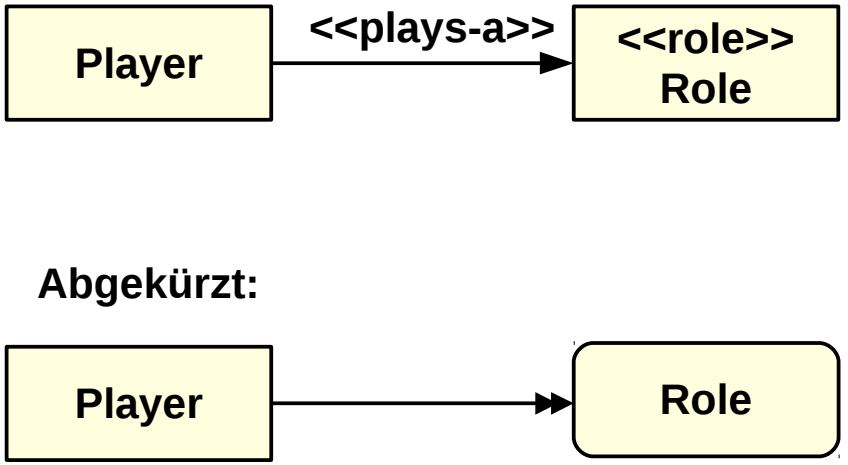
Rollen als Grund für Polymorphie

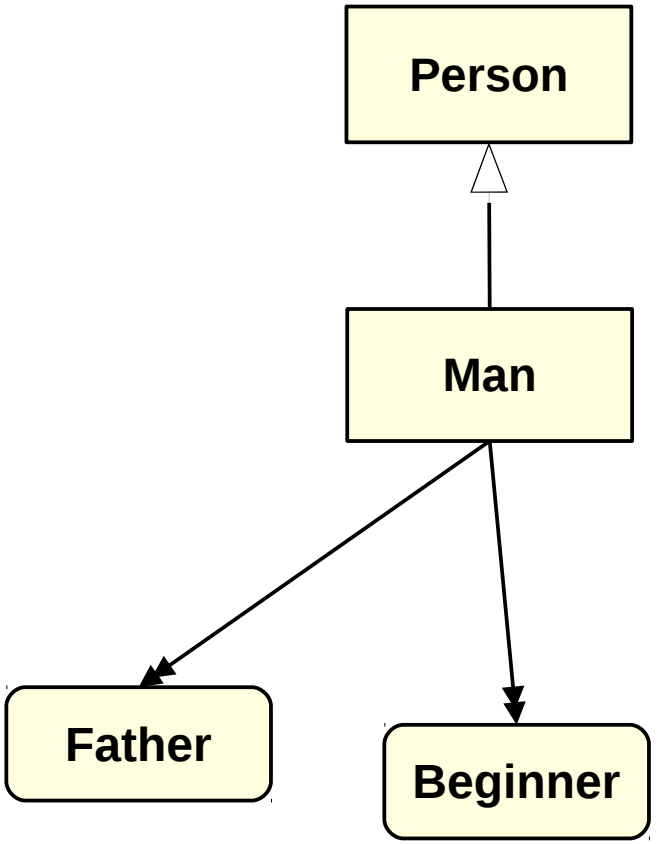
- Polymorphie ist gerade dann ein wichtiges Konzept, wenn Objekte temporär **Rollen spielen**
- Eine **Rolle** ist ein dynamischer Zustand eines Objekts: ein Objekt spielt eine Rolle für einige Zeit
  - Akademiker spielen verschiedene Rollen bez. ihres Studentenlebens (vgl. unser Beispiel)
  - Männer können ledig, verheiratet, geschieden, wiederverheiratet sein
  - Menschen können Kinder, Teenager, Erwachsene, Rentner sein
  - Autos können fabrikneu, Jahreswagen, gebraucht, Wrack, oder Schrott sein
- Weil ein Objekt verschiedene Rollen spielen will, muss es polymorph benutzbar sein!



Spezielle Icons für Stereotypen

- Für alle grafischen Elemente in UML können spezielle Icone vereinbart werden.
- Für Rollenklassen wählen wir ein Oval, für die Plays-a-Relation einen Pfeil mit Doppel-Kopf-Dreieck.

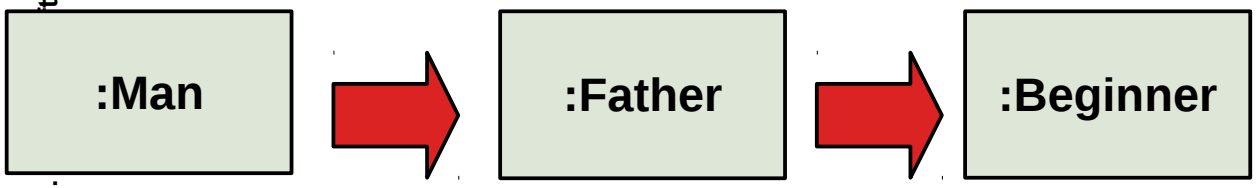


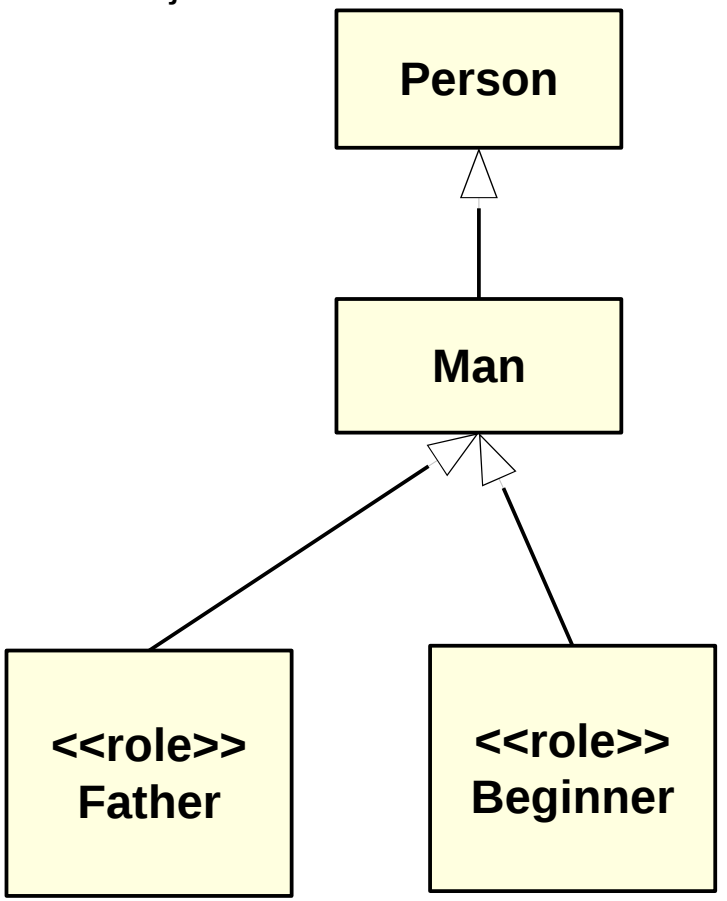


Implementierungsmuster: “Rollen mit Unterklassen”

- In Java gibt es kein Sprachkonzept “Rolle”
- Aber man kann mit Unterklassen die Rollen einer Oberklasse realisieren (“Implementierungsmuster”)
- Man implementiert den Wechsel einer Rolle durch den Wechsel der entsprechenden Unterklasse durch
  - Alloziere und fülle neues Objekt aus den Werten des alten Objektes heraus
  - Setze Variable um auf neues Objekt
  - (Deralloziere altes Objekt)

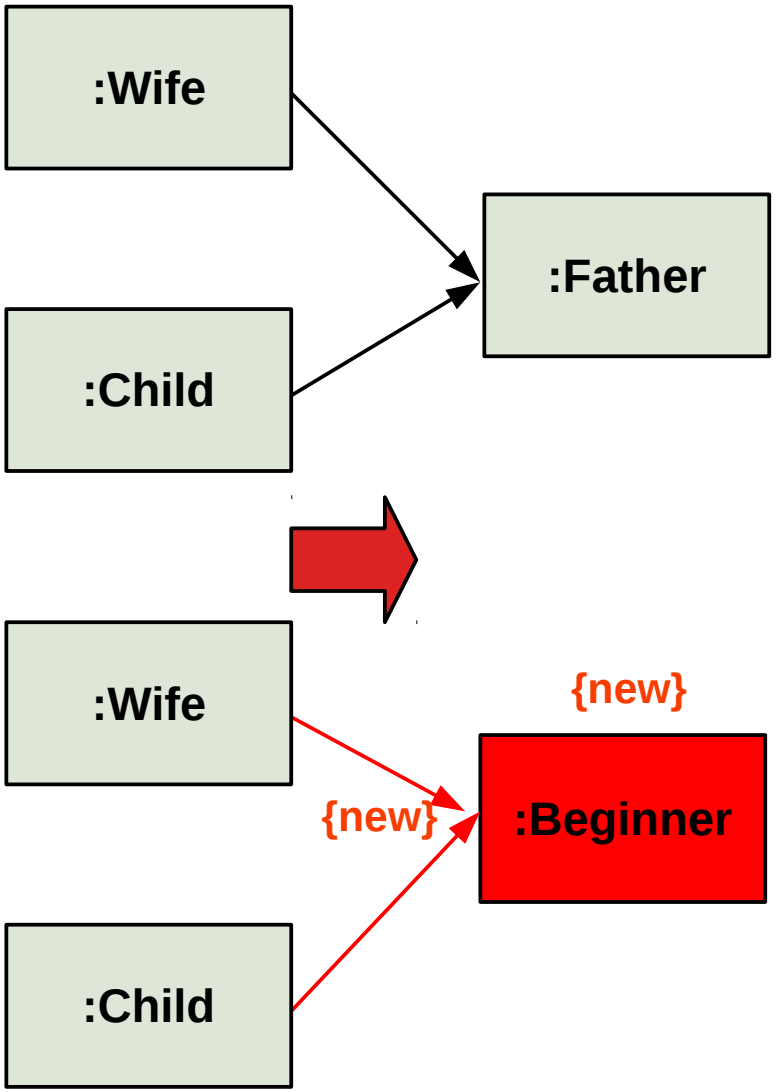
Objektdiagramm (snapshots):

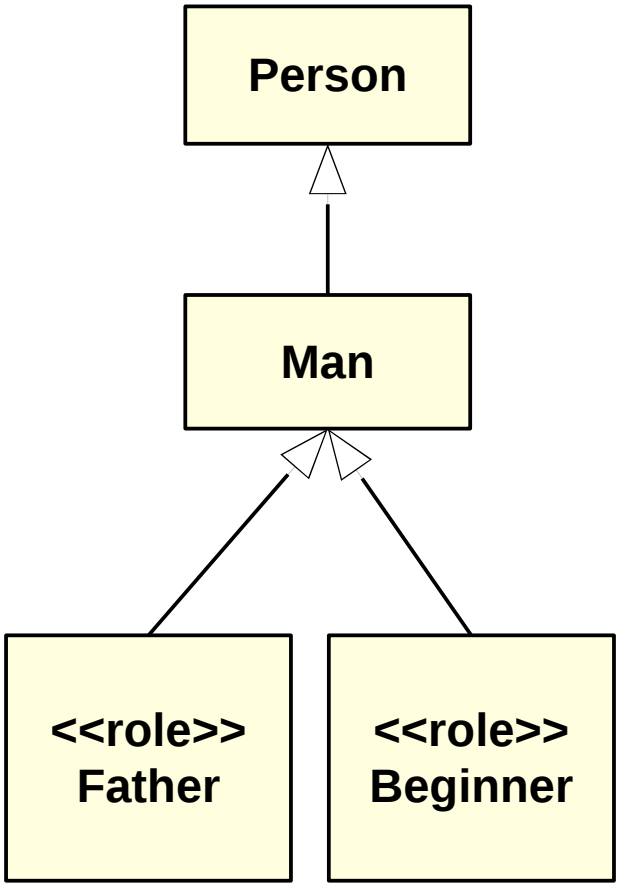




Implementierungsmuster: “Rollen mit Unterklassen”

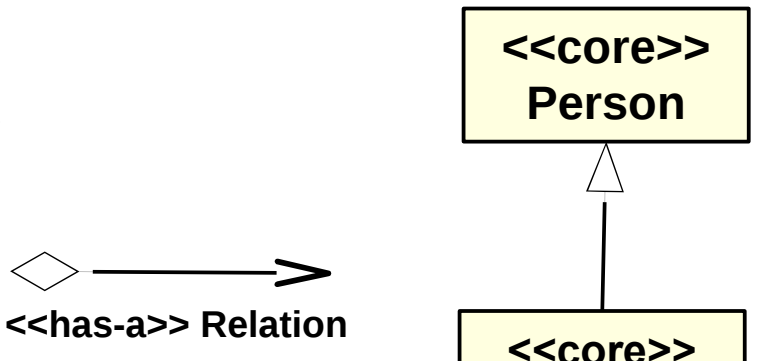
- Problem: Referenzen auf Objekte müssen bei Rollenwechsel umgesetzt werden



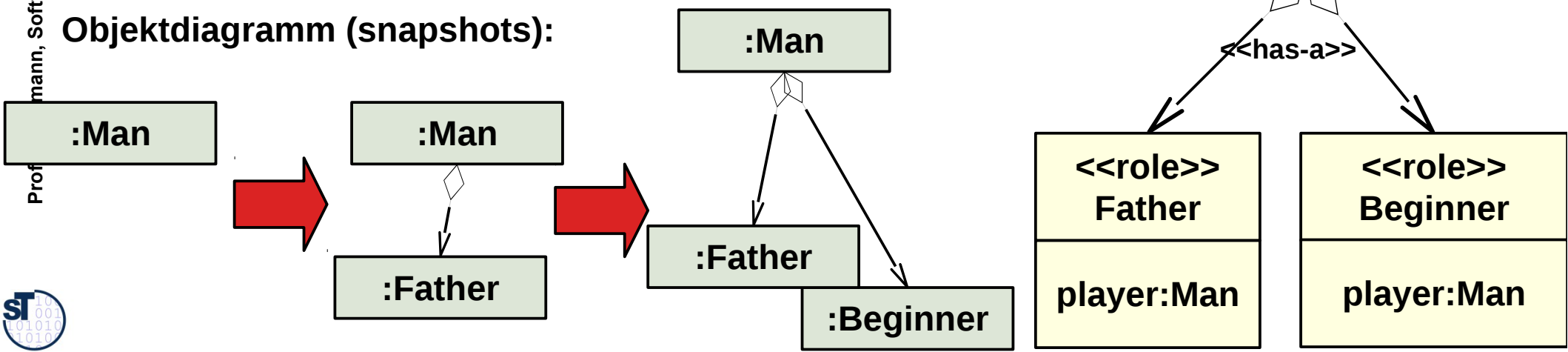


Implementierungsmuster: “Rollen mit einfacher Aggregation”

- Rollen können auch durch *Unterobjekte* eines *Kernobjektes* modelliert werden *(Rollen durch has-a, Aggregation)*
  - Vorteil: Kernobjekt kann viele Rollen spielen
  - Vorteil: Referenzen auf Kernobjekt bleiben bei Rollenwechsel erhalten!
- Man implementiert den Wechsel einer Rolle durch den Wechsel der entsprechenden Unterklasse des Rollen-Unterobjektes durch
  - Alloziere und fülle neues Rollen-Unterobjekt
  - Setze Variable um auf neues Objekt
  - Dealloziere altes Rollen-Unterobjekt

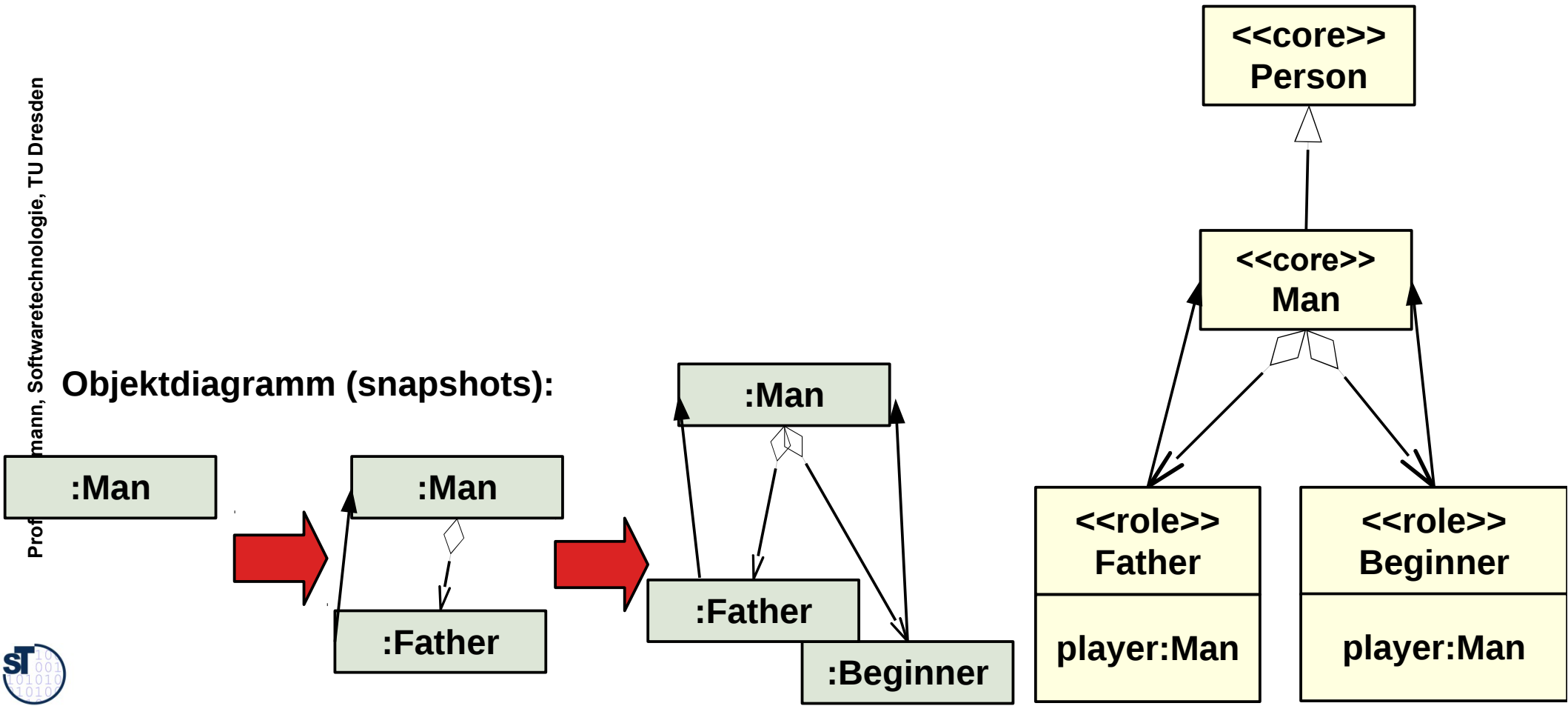


Objektdiagramm (snapshots):



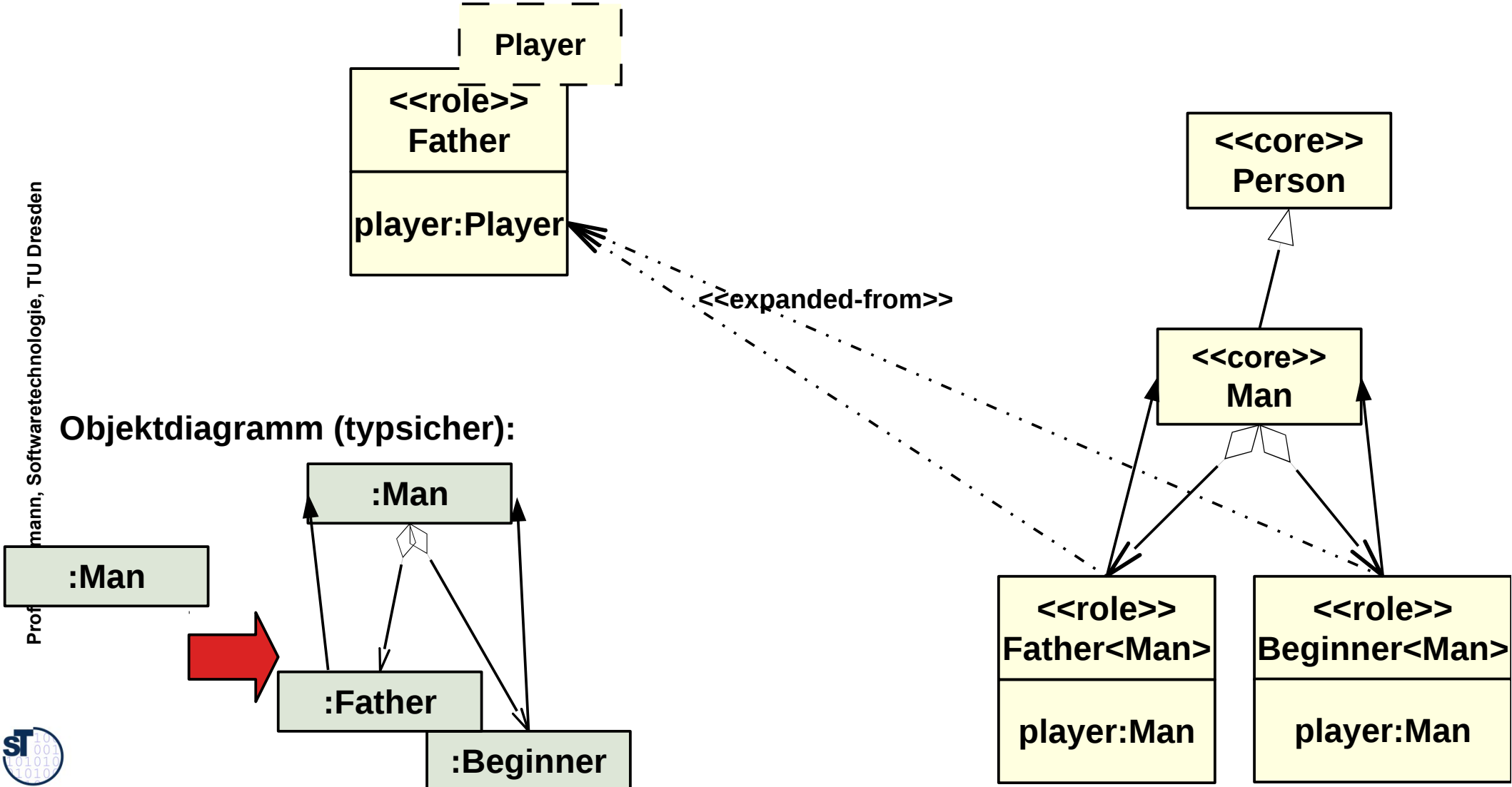
Implementierungsmuster: “Rollen mit Aggregation und Kern-Link”

- Rollen können zusätzlich durch einen Rückwärts-Link `player` mit dem Kernobjekt verbunden sein, da sie semantisch zu ihm gehören
  - Vorteil: Kernobjekt kann viele Rollen spielen und ist dennoch eng mit ihnen verbunden



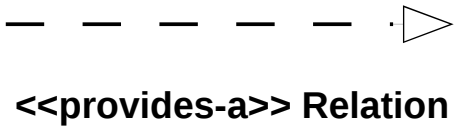
Implementierungsmuster: “Rollen mit generischem Player-Link”

- Rollen können als generische Klassen gesehen werden, die als generischen Parameter den Player erwarten
  - Dann ist das Netz zwischen Kernobjekt und Rollen typsicher

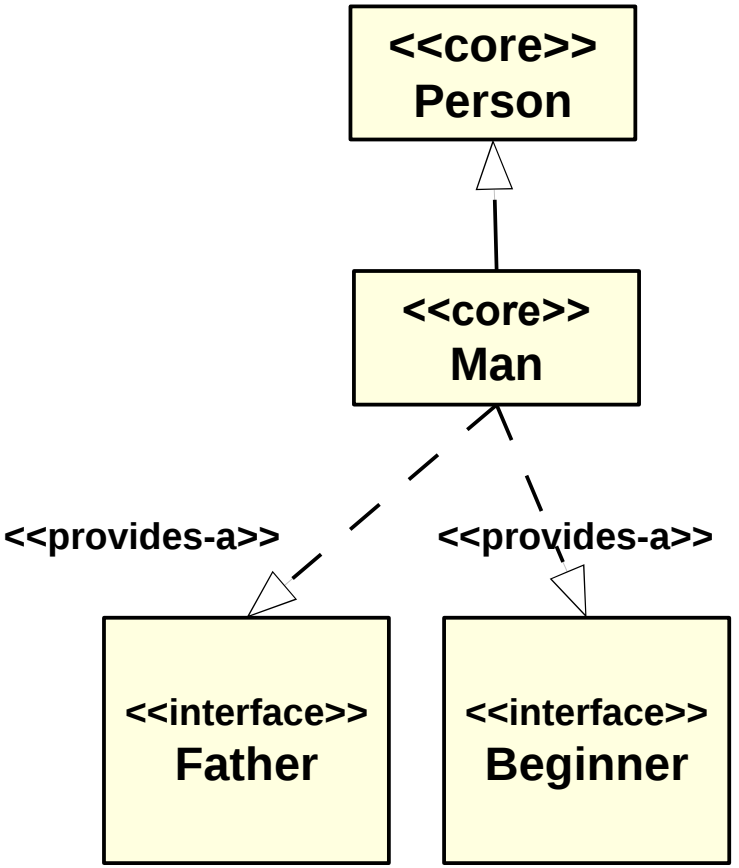


Implementierungsmuster: “Rollen mit Mehrfachvererbung”

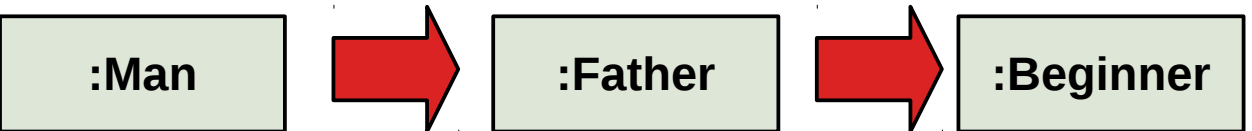
- Rollen können auch durch *Mehrfachvererbung* realisiert werden
  - .. ähnlich zur Realisierung mit Einfachvererbung
- Vorteil: Kernobjekt erbt Rollenverhalten
- Nachteil: nur in Scala und C++ möglich



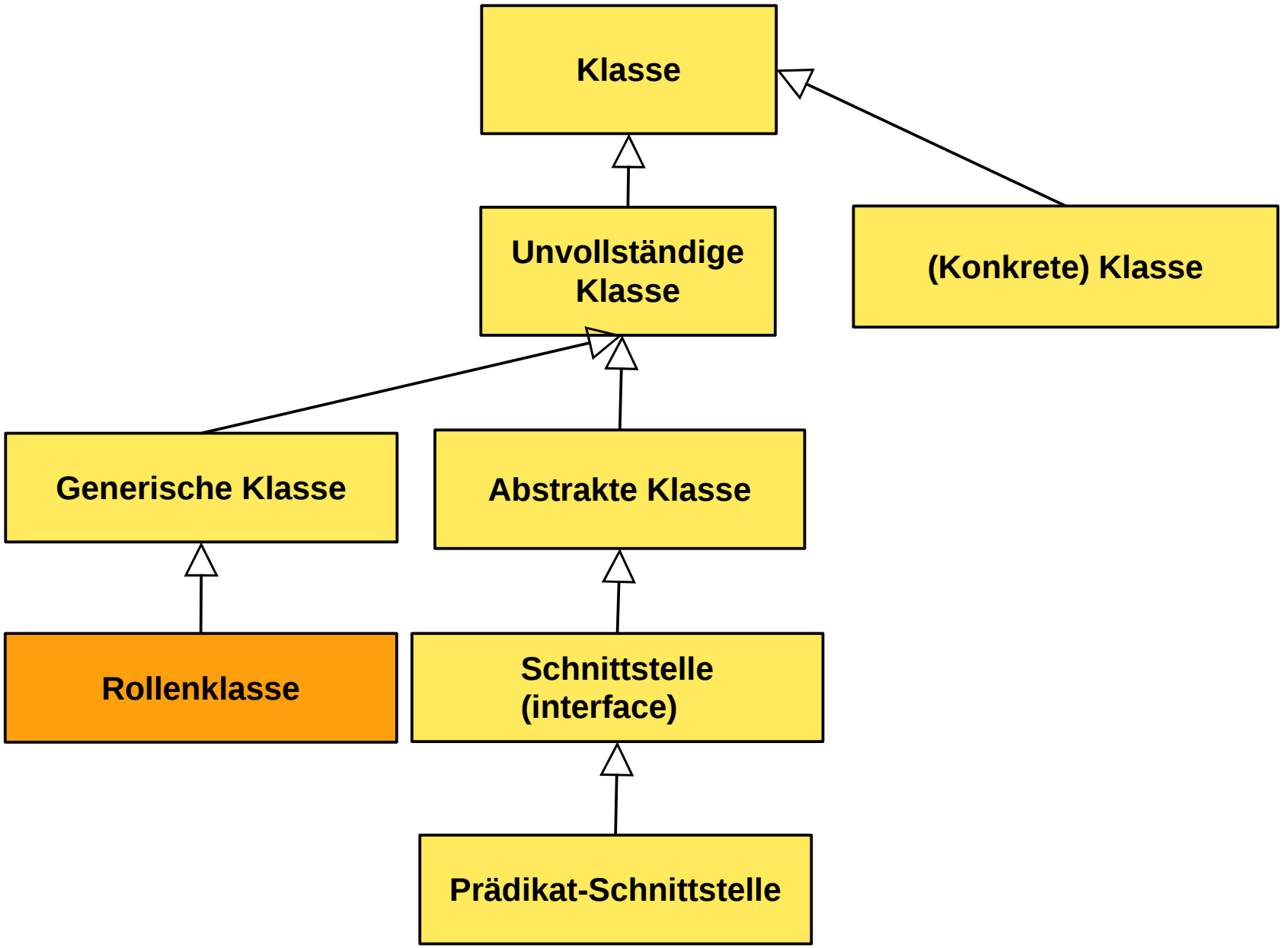
<<provides-a>> Relation



Objektdiagramm (snapshots):



Begriffshierarchie von Klassen (Erweiterung)



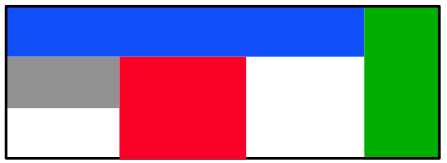
Was haben wir gelernt?

- Vererbung zwischen Klassen erlaubt die *Wiederverwendung* von Merkmalen aus Oberklassen, sowohl Schnittstellen als auch Implementierungen
  - Einfache Vererbung führt zu Vererbungshierarchien
- 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 die Merkmalssuche eingespart werden kann
- Die Klasse `Object` enthält als implizite Oberklasse der Java-Bibliothek gemeinsam nutzbare Funktionalität für alle Java-Klassen
- Schnittstellen sind vollständig abstrakte Klassen
- Generische Klassen ermöglichen typsichere Wiederverwendung von Code über Typ-Parameter
- Rollenklassen sind dynamische, unvollständige Klassen

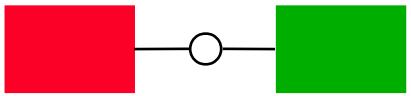
Warum ist das wichtig?

- Wiederverwendung ist eines der Hauptprobleme des Software Engineering
- Vererbung, generische Klassen und Rollenklassen können Code-Replikate und Code-Explosion weitgehend vermeiden
  - Der Test von Software wird systematisiert
- 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

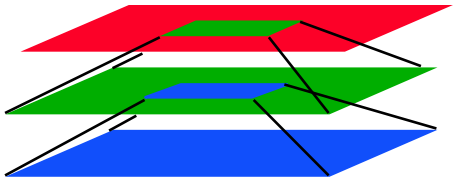
Prinzipielle Vorteile von Objektorientierung



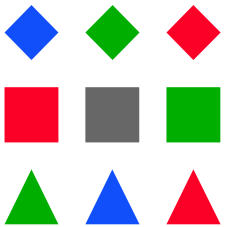
Zuständigkeitsbereiche



Klare Schnittstellen



Hierarchie



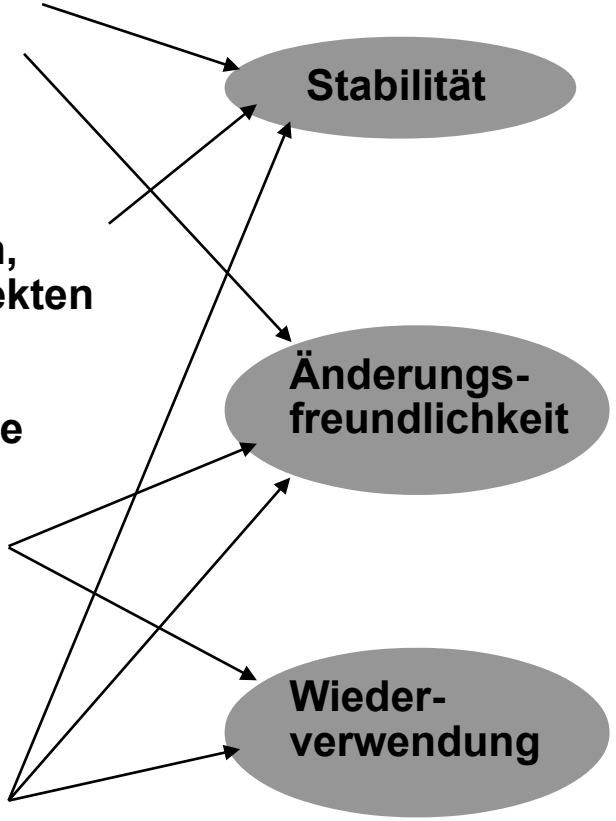
Baukastenprinzip

Lokalität: Lokale Kapselung von Daten und Operationen, gekapselter Zustand

Typen und Typsicherheit
Definiertes Objektverhalten, Nachrichten zwischen Objekten

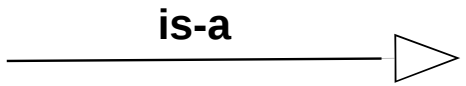
Vererbung und Polymorphie (Spezialisierung), Wiederverwendung
Klassenschachtelung

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



Appendix

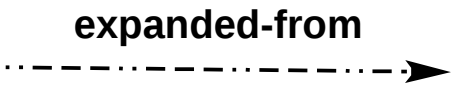

knows-a


is-a

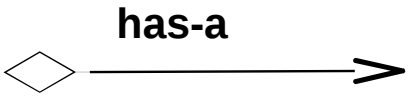

implements


provides-a


is-instance-of

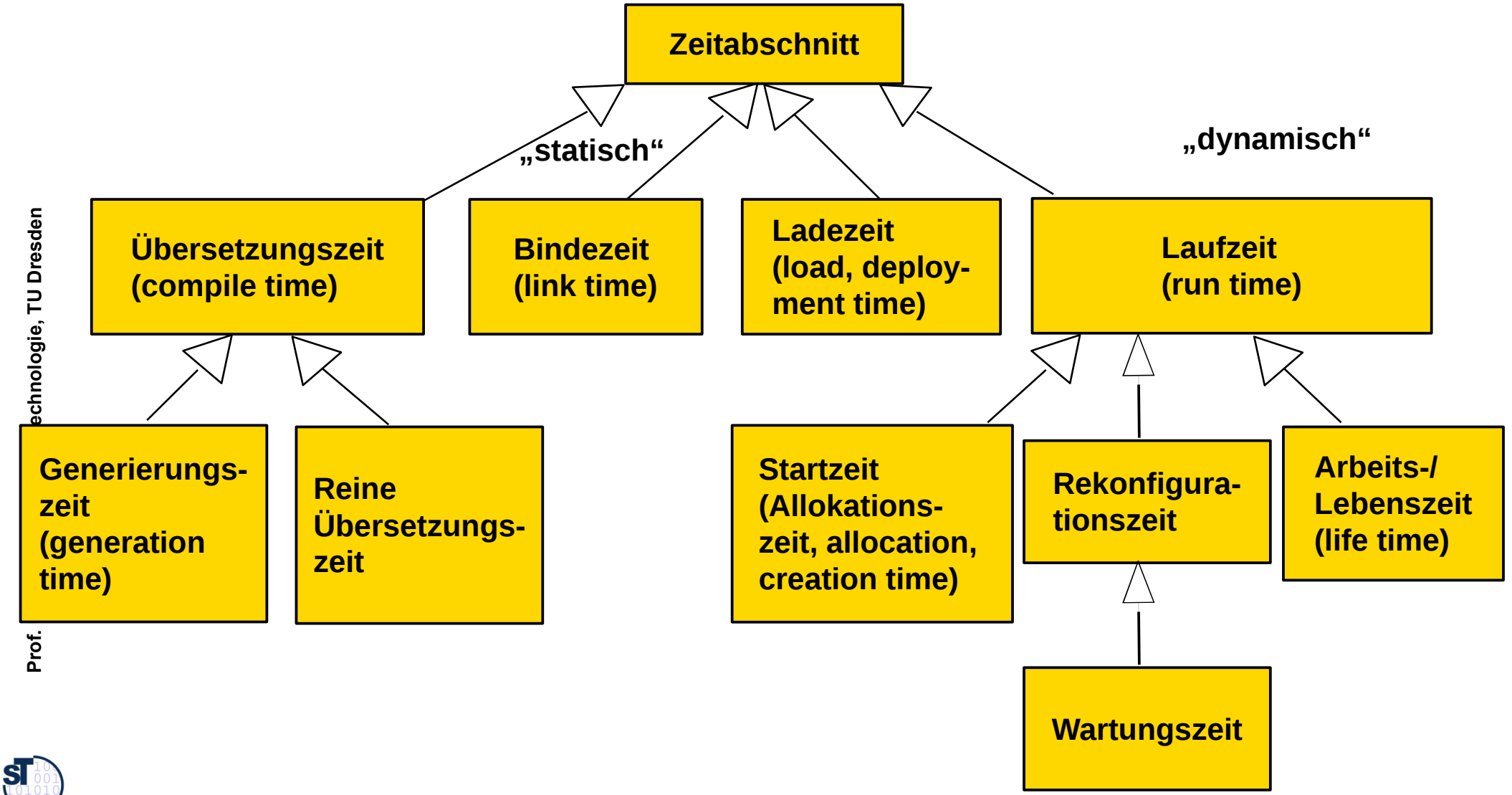

expanded-from


plays-a


has-a

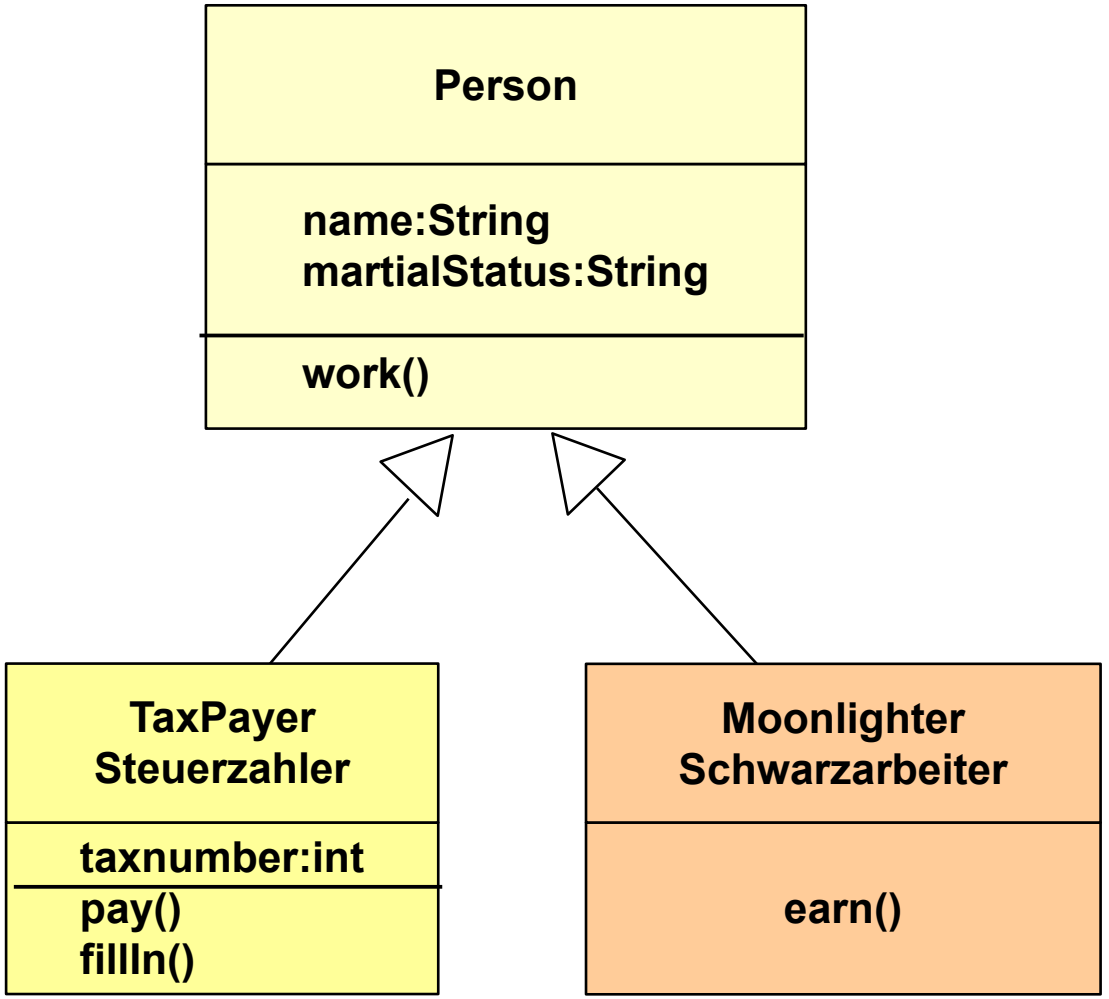
Bsp. Begriffshierarchie: Verschiedene Zeiten im Lebenszyklus einer Software

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



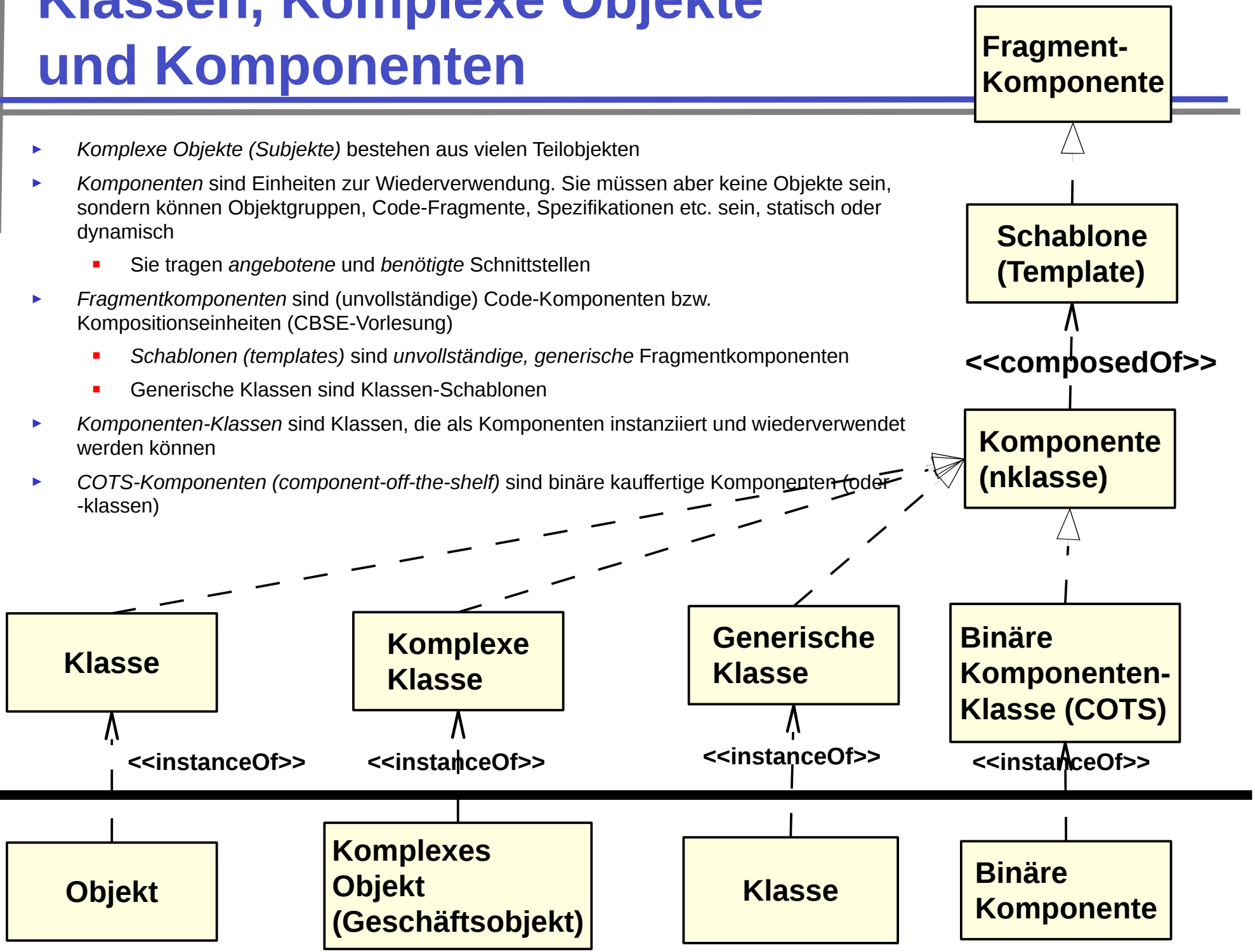
Vererbung im Speicher

- ... am Beispiel Steuerzahler



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