11. Vererbung und Polymorphie
Die Filter gegen Codeverschmutzung
Die Basismittel zur Erweiterung von Software

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 21-1.1, 19.04.21

1) Vererbung zwischen Klassen
2) Vererbung im Speicher
3) Polymorphie

Sprechstunde Prof. Aßmann:

- **Montags während der Vorlesungszeit 11:10**
- **Donnerstags nach Vereinbarung 11:00-13:00**

https://matrix.tu-dresden.de/#/room/#Sprechstunde-INF-ST-Prof.Assmann:tu-dresden.de

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 in SLUB
- Noch ein sehr gutes, umfassend mit Beispielen ausgestattetes Java-Buch:
  - C. Heinisch, F. Müller, J. Goll. Java als erste Programmiersprache. Vo… Einsteiger zum Profi. Teubner.
- Für alle, die sich nicht durch Englisch abschrecken lassen:
- eBooks, von unserer Bibliothek SLUB gemietet:
  - **http://www.dbod.de/db/start.php?database=ebl_ebl (DBoD)**
- Free Books: http://it-ebooks.info/
  - Kathy Sierra, Bert Bates: Head-First Java http://it-ebooks.info/book/255/

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

http://www.dbod.de/db/start.php?database=ebl_ebl

Obligatorische Literatur

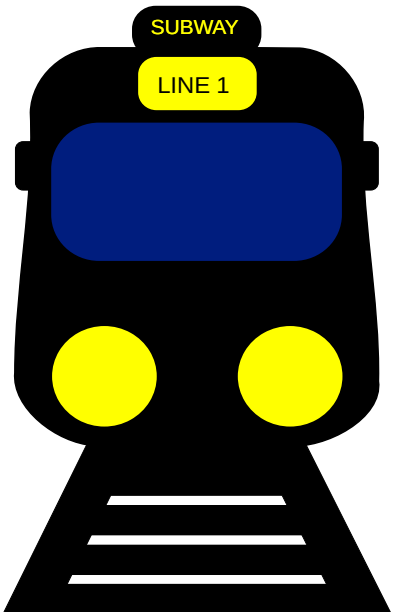
- ST für Einsteiger Kap.4+ 9, Teil II (Störrle, Kap. 5.2.6, 5.6)
  - Zuser Kap 7, Anhang A
- Java
  - Oracle Tutorial https://docs.oracle.com/javase/tutorial/java/IandI/index.html
  - Balzert LE 9-10
  - Boles Kap. 7, 9, 11, 12

Ziele

- Elementare Techniken der **Wiederverwendung** von objektorientierten Programmen kennen
  - **Generalisierung** und **Spezialisierung** mit einfacher Vererbung zwischen Klassen, konzeptuell und im Speicher
  - **Merkmalssuche** in einer Klasse und in der Vererbungshierarchie aufwärts nachvollziehen können
    - Überschreiben von Merkmalen verstehen
- **Dynamische Architektur** eines objektorientierten Programms verstehen
  - **Lebenszyklen** von Objekten verstehen
  - **Polymorphie** verstehen

Java Herunterladen

- Das Java Development Kit (JDK)
- https://adoptopenjdk.net/, brew tap adoptopenjdk/openjdk
- http://openjdk.java.net/, brew info openjdk



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

Hinweise zu weiterer Literatur: 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

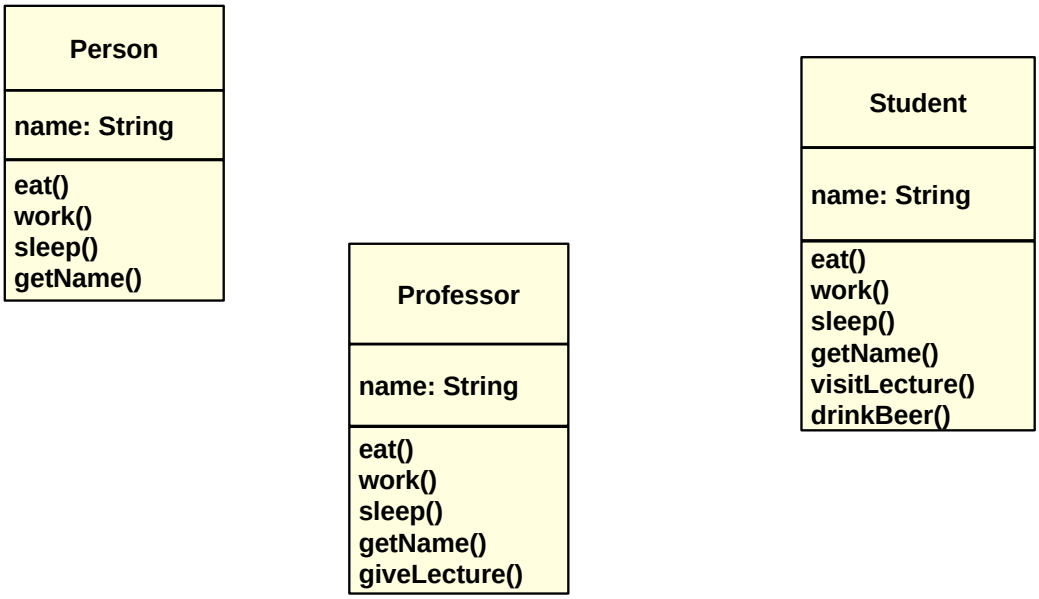
In der Vorlesung werden viele wissenschaftliche Papiere zur zusätzlichen Lektüre empfohlen. Sie sind dank der SLUB mit Lizenzen für viele elektronische Bibliotheken dieser Welt ausgestattet. Nutzen Sie sie!

11.1 Vererbung zwischen Klassen
beseitigt Codereplikate

Ähnlichkeit von Klassen sollten in Oberklassen ausfaktorisiert werden

Codeverschmutzung am Beispiel (“unsoziales Programmieren”)

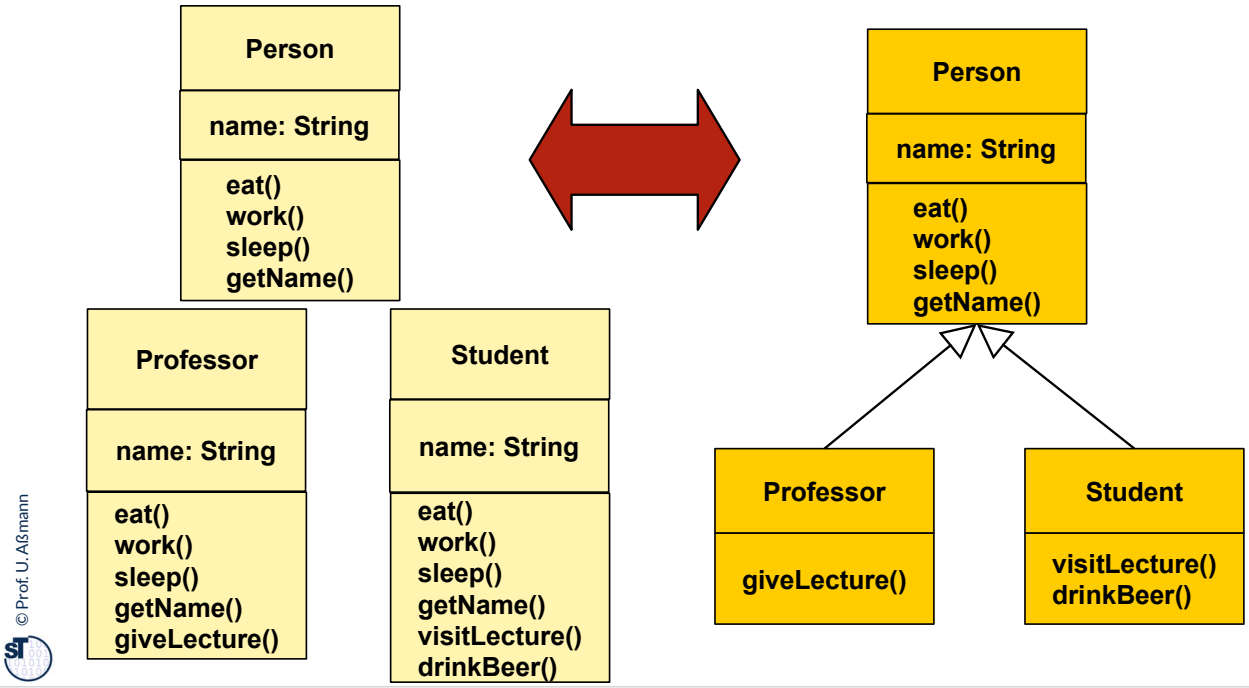
- **Hier:** Person wurde zu Professor und Student kopiert und danach erweitert
- Warum ist diese Art des Programmierens “unsozial”?



Wahlloses Hinzufügen von Attributen und Methoden zu Klassen führt in eine Sackgasse, denn meist werden die Funktionalitäten (“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

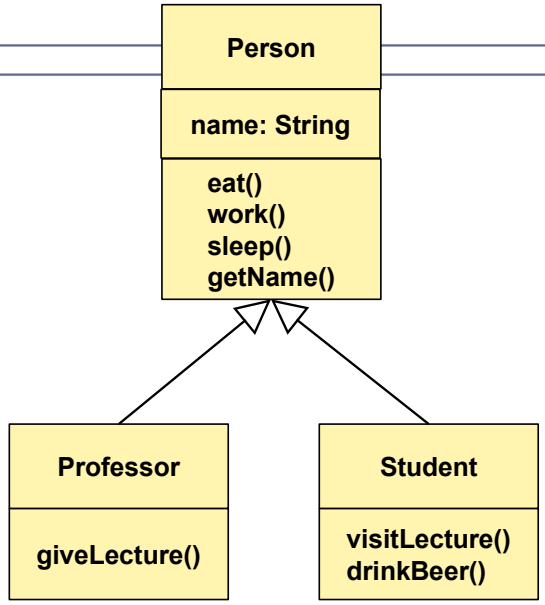


Zum Glück gibt es *Vererbung* von Merkmalen von Oberklassen zu Unterklassen.

Vererbung faktorisiert Merkmale aus Unter- in Oberklassen aus.

Einfache Vererbung

- **Vorteil:** Vererbung drückt Gemeinsamkeiten aus
  - Die Unterklasse ist damit ähnlich zu dem Elter und den Geschwistern
  - Vererbung stellt *is-a*-Beziehung her "`erbt-Merkmale-von`"
- **Hier**: Oberklasse Person enthält alle gemeinsamen Merkmale der Unterklasse
- Vererbung entspricht *Ausfaktorisierung*
- 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.

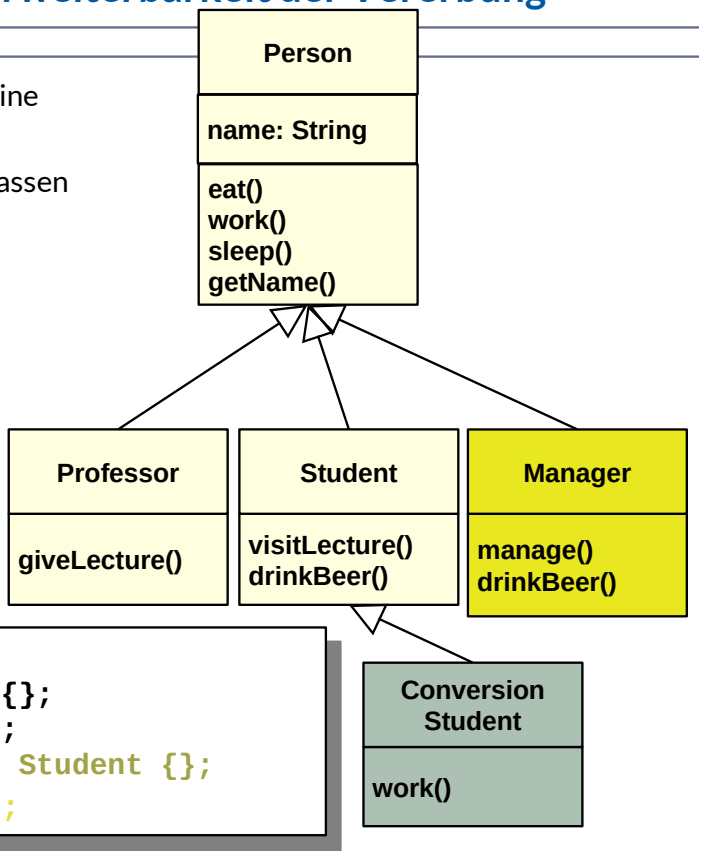
In anderen Programmiersprachen wird u.U. andere Syntax für Vererbungsoperatoren benutzt:

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

11.1.1 Vererbungshierarchien erlauben nachträgliche Erweiterbarungen

Vorteil:
Horizontale und vertikale Erweiterbarkeit der Vererbung

- **Vorteil:** Mit Vererbung kann man eine Klassenhierarchie erweitern
- ***Horizontal*** durch neue Schwesterklassen
- ***Vertikal*** durch neue Unterklassen
- Und wie mittendrin?




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


Eine Vererbungshierarchie kann später erweitert werden, unter voller Wiederverwendung aller Attribute aller Oberklassen:

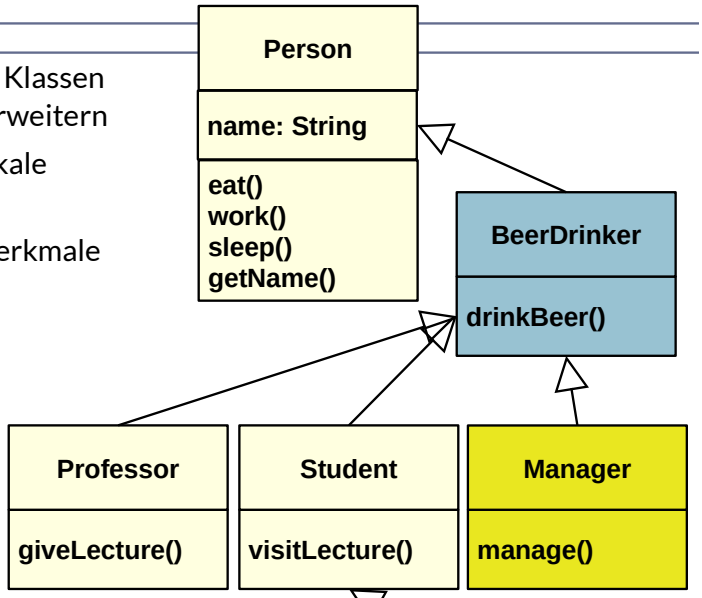
- Horizontal durch neue Schwesterklassen
- Vertikal durch neue Unterklassen
- Middle-Out durch neue zwischengeschobene Klassen (Interzeptoren)

Dadurch ergibt sich der größte Vorteil der Objektorientierung:

- Nachträgliches Erweitern um neue Funktionalität (biologisches Wachstum)
- Wiederverwendung aller bisher geleisteten Arbeit

Vorteil: “Middle-Out” Erweiterbarkeit der Vererbung

- **Vorteil:** Mit zwischengeschobenen Klassen kann man eine Klassenhierarchie erweitern
- Und für neue horizontale und vertikale Erweiterungen vorbereiten
- “Zwischenschieben” faktorisiert Merkmale in einer Vererbungshierarchie um (***Refactoring***)




```
// Java
BeerDrinker extends Person {};
Professor extends BeerDrinker {};
Student extends BeerDrinker {};
ConversionStudent extends Student {};
Manager extends BeerDrinker {};
```


Middle-Out Erweiterung wird im Refactoring benutzt, um Programme für zukünftige Erweiterungen (horizontal, vertikal) vorzubereiten.

Dadurch ergibt sich der größte Vorteil der Objektorientierung:

- Nachträgliches Erweitern um neue Funktionalität (biologisches Wachstum) wird erleichtert, weil Refactoring neue “Mittelklassen” einführt, die weitere Wiederverwendung erlauben, auch mit *neuen, horizontalen oder vertikalen Erweiterungen*
- Refactoring ist eine wesentliche Operation zur Verlängerung des Lebens eines objektorientierten Programms.

Vorteil: Verhaltenskonforme Ersetzung

(Liskow'sches Ersetzungsprinzip, (Liskow's Substitution Principle, LSP)

https://de.wikipedia.org/wiki/Liskovsches_Substitutionsprinzip

- Es gibt Programmiersprachen, in denen das LSP per Sprachdefinition gilt.
- In Java muss das LSP leider durch Testen abgesichert werden (Kapitel Test)

Ein Programm, das mit einem Objekt einer Klasse verwendet,
kann fehlerfrei mit jedem Objekt einer ihrer Unterklassen arbeiten.

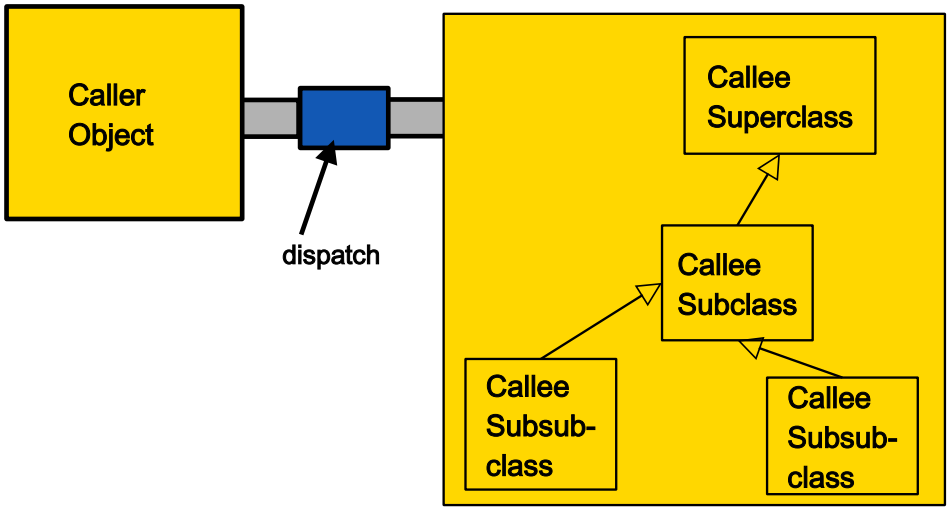
- Horizonale, vertikale und zwischengeschobene Erweiterungen werden immer angwendet, um Code zu erweitern(!)
  - aber das LSP muss abgesichert werden.

Biologisches Programmieren in Java bedeutet,
•eine Vererbungshierarchie nachträglich horizontal, vertikal zu erweitern,
•zu refaktorisieren, um weitere Erweiterungen vorzubereiten und
•das Liskow'sche Prinzip für die Erweiterungen mit Tests zuzusichern.

Es gibt Sprachen, in denen das LSP (Liskow substitution principle) per Definition der Sprache gilt (z.B. Eiffel, Sather, Haskell). Allerdings ist in diesen Sprachen die Verwendung der Vererbungsrelation eingeschränkt. Man gewinnt dafür, dass der Testaufwand fällt.

Liskow'sches Ersetzungsprinzip

- Egal, welches Objekt einer Klasse aus einer Klassenhierarchie für die Abarbeitung eines Aufrufs genommen wird, - der Aufruf muss immer funktionieren und darf nicht zum Absturz des Aufrufers führen



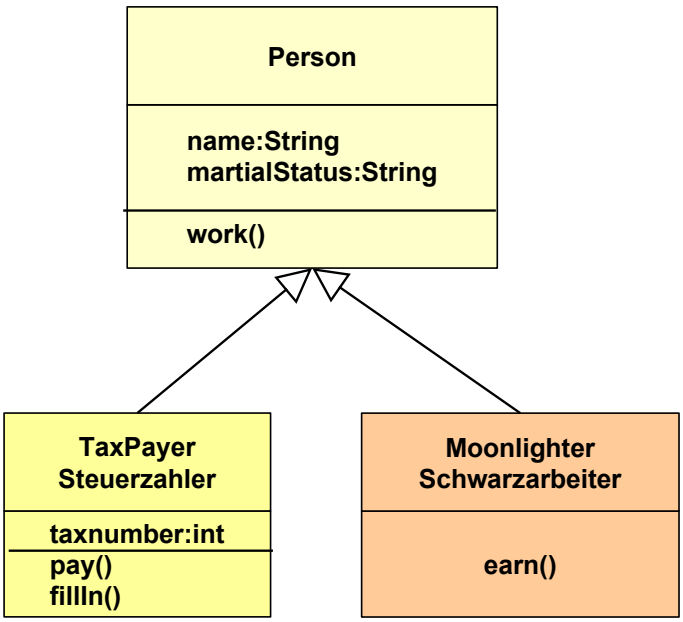
Der Typ eines Aufgerufenen (bzw. des Empfängers einer Botschaft) kann innerhalb des Vererbungsbaumes variieren. Der Typ des Aufgerufenen ist *vielgestaltig* (Polymorphie).

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)*.

11.2 Wie stellt sich Vererbung im Speicher der JVM dar?

11.1.2 Vererbung im Speicher

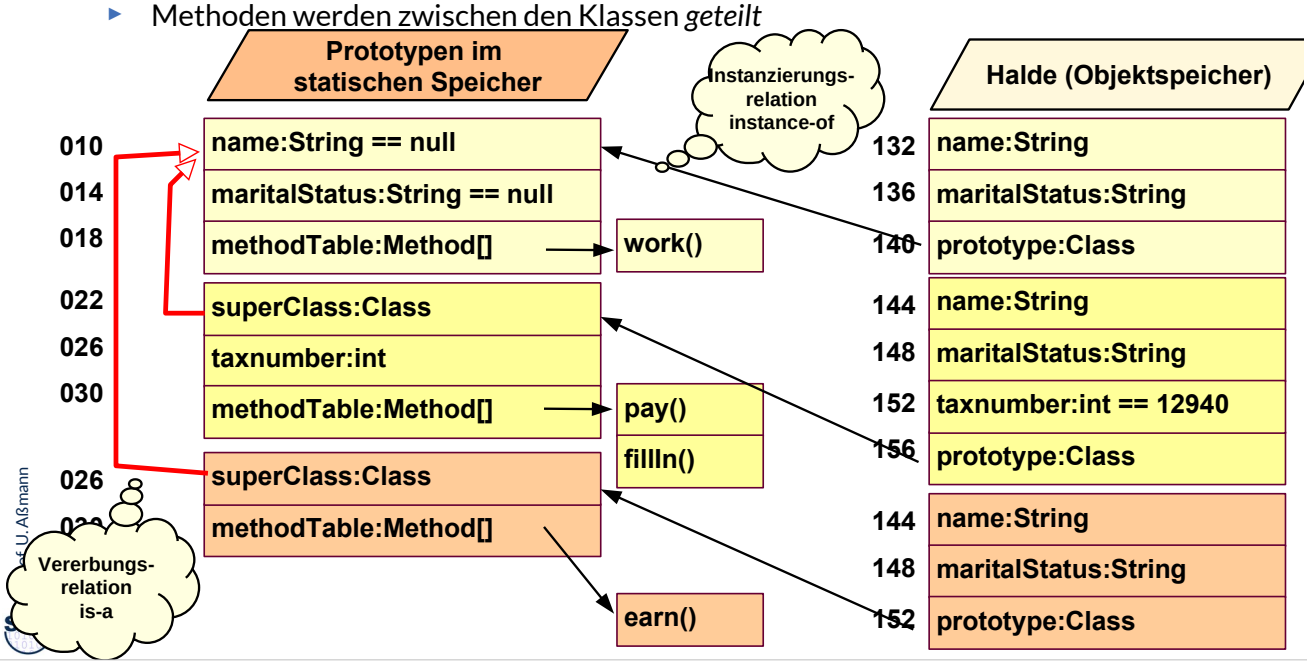
- ... am Beispiel Steuerzahler



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

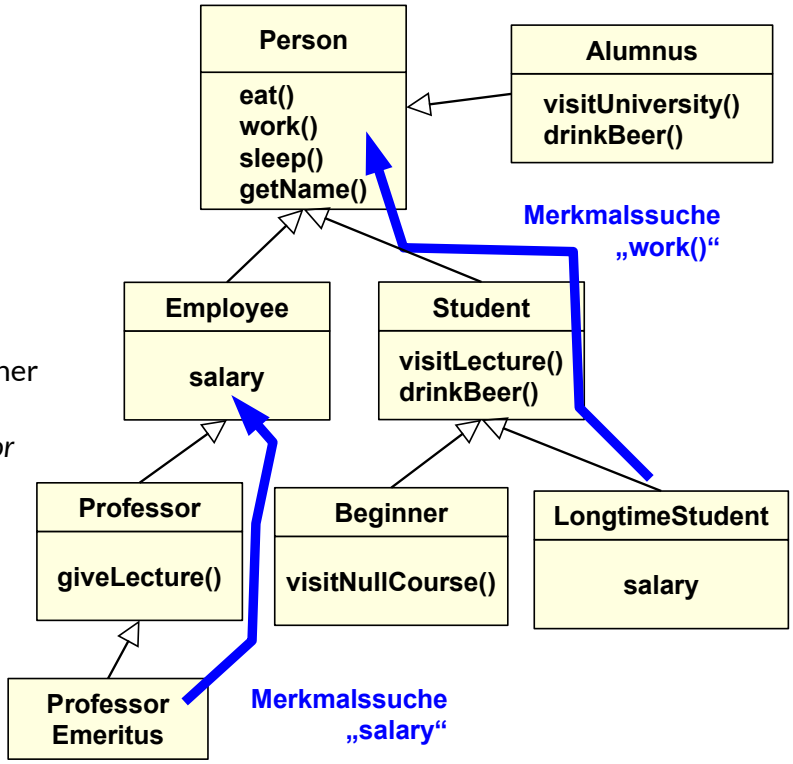
- Die Vererbungsrelation wird im Speicher als *Baum* zwischen den Prototypen der Ober- und Unterklassen dargestellt (Verzeigerung von unten nach oben)
  - Unterscheide davon die Objekt-Prototyp-Relation instance-of!
- Methoden werden zwischen den Klassen *geteilt*



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)*



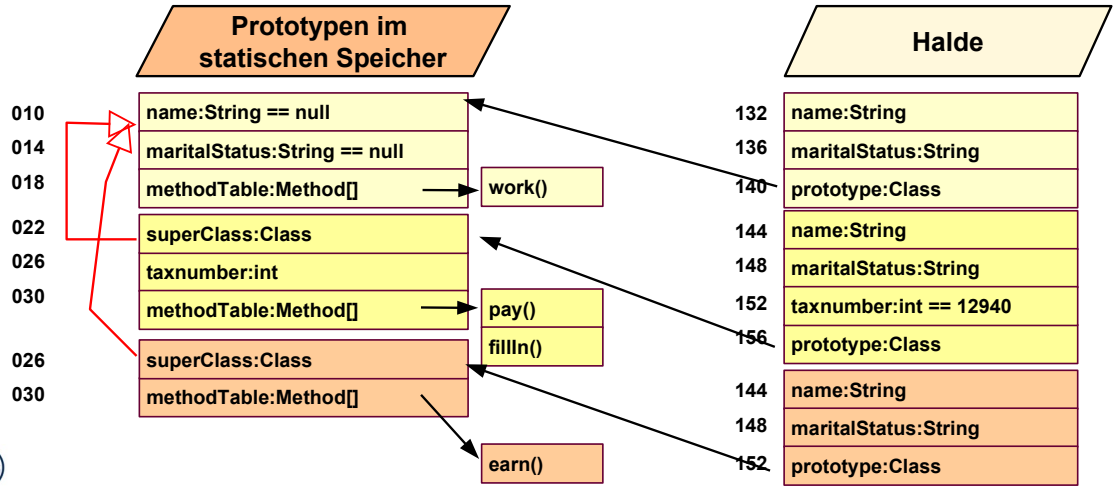
PersonInheritanceDrinkBeer.java

PersonInheritanceDemo.java

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

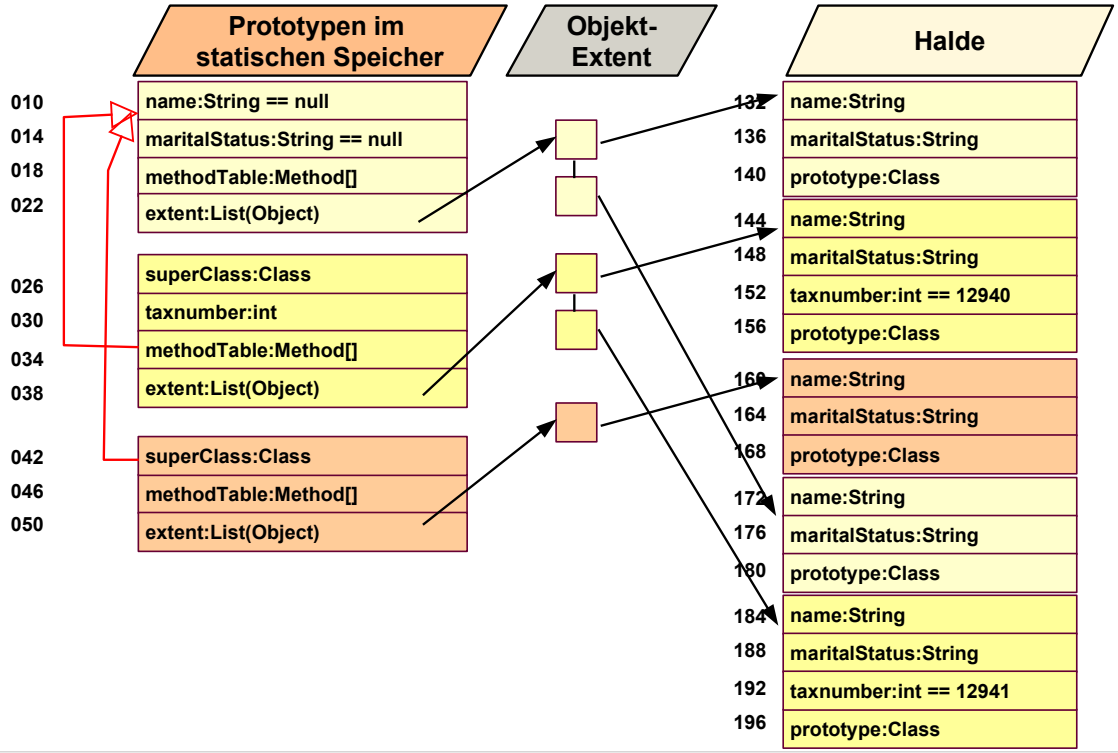
1) Suche Attribut *name* in Steuerzahler: direkt vorhanden
2) Suche Methode *pay()* in Steuerzahler: Schlage Prototyp nach, finde in Methodentabelle des Prototyps
3) Suche Methode *work()* in Steuerzahler: Schlage Prototyp nach, Schlage Oberklasse nach (Person), finde in Methodentabelle von Person
4) 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

was a bug in 0.1

- Zu einer Klasse vereinige man alle Extents aller Oberklassen

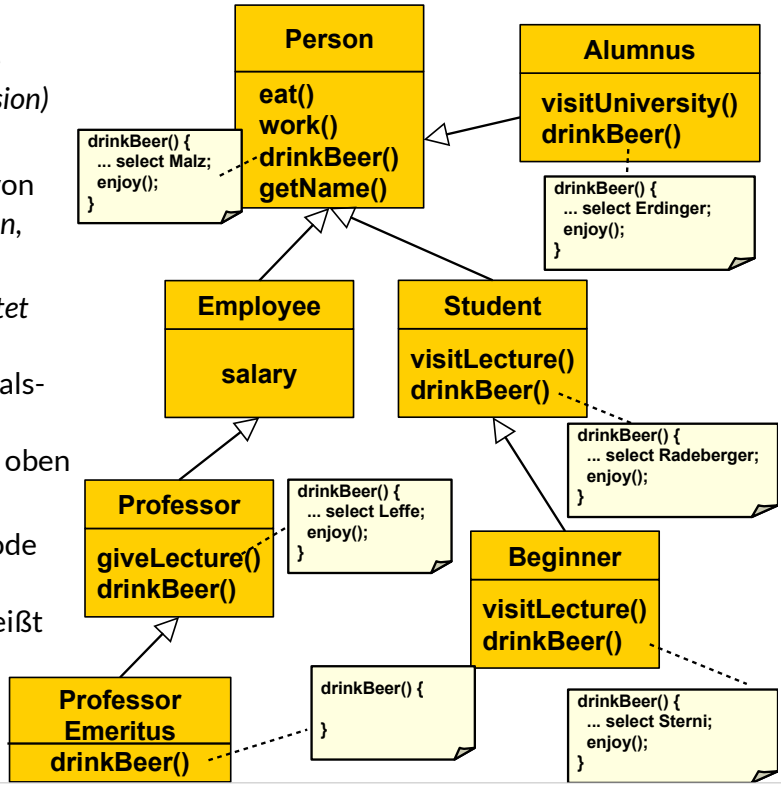


Unter der Vererbungsrelation erbt ein Objekt alle Merkmale seiner Oberklassen. Damit gehören auch alle Objekte aller seiner Oberklassen zu “seinen Geschwistern”, d.h. zum Extend seiner Klasse.

Diese Definition erweitert also die Defintion von Kap. 10, indem nun die Oberklassen mit einbezogen werden.

Erweitern und Überschreiben von Merkmalen

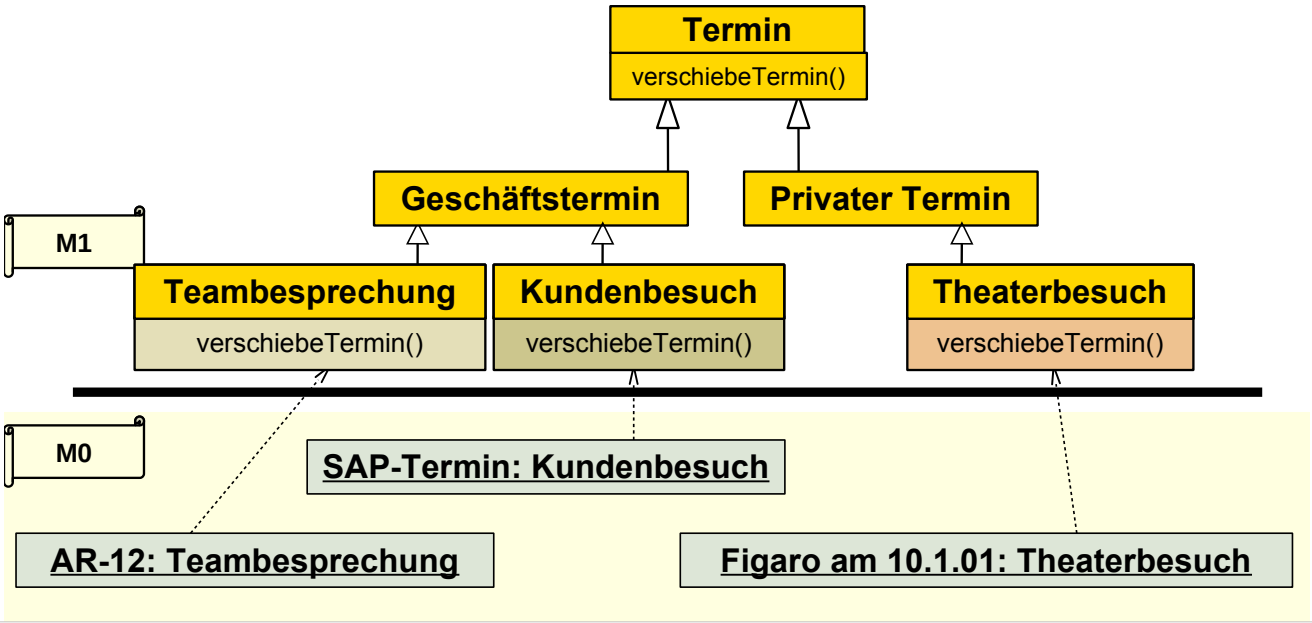
- Eine Unterklasse kann neue Merkmale zu einer Oberklasse hinzufügen *(Erweiterung, extension)*
- Definiert eine Unterklasse ein Merkmal erneut, spricht man von einer *Redefinition (Überschreiben, overriding)*
  - Dieses Merkmal *überschattet (verbirgt)* das Merkmal der Oberklasse, da der Merkmals-suchalgorithmus 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.

Beispiel: Termin-Klasse und Termin-Objekte im fachlichen Modell “Terminverwaltung”

- 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



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.

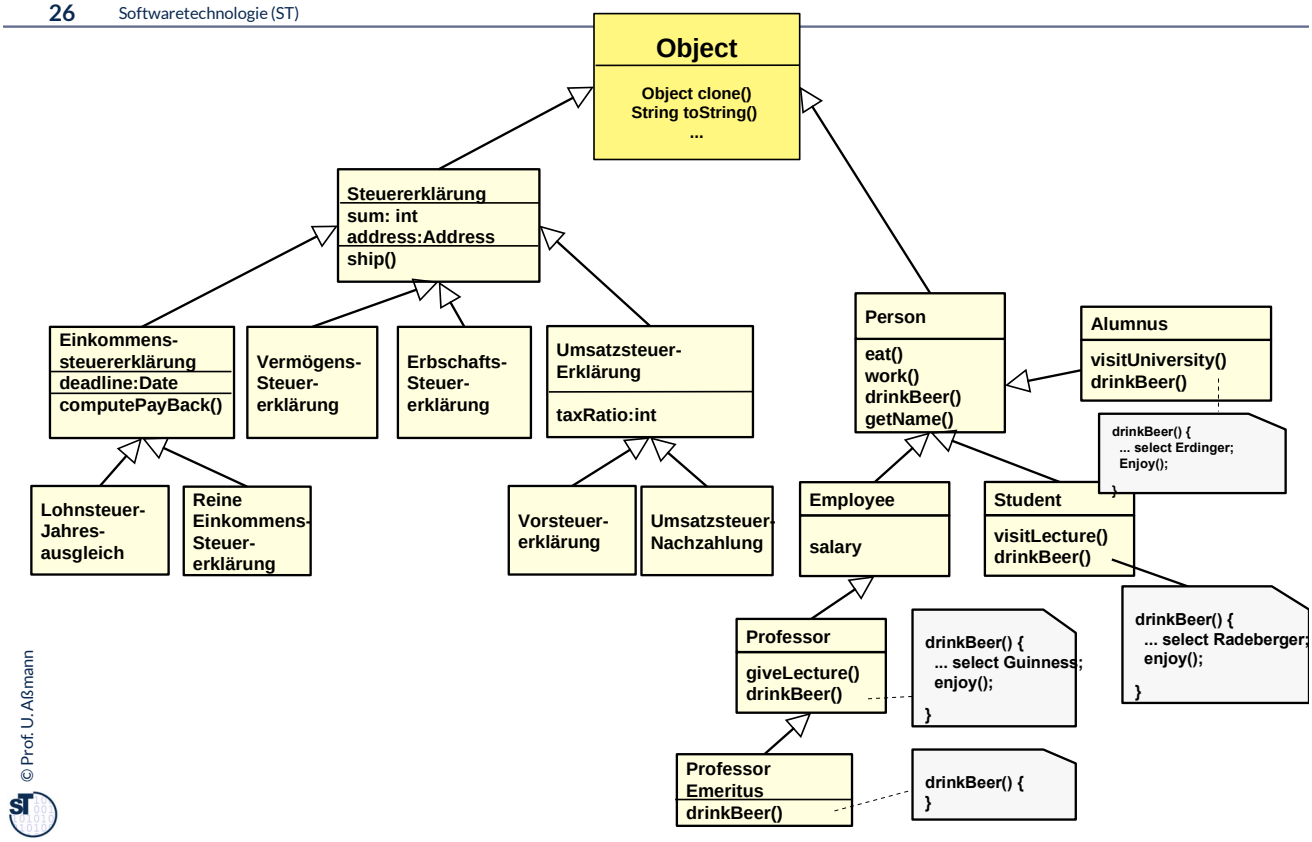
HashCode() ermittelt einen eindeutigen Identifikator des Objekts (große Zahl), die mit hoher Wahrscheinlichkeit eindeutig ist. Zwei Objekte mit verschiedenem Hashcode sind unterschiedlich.

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

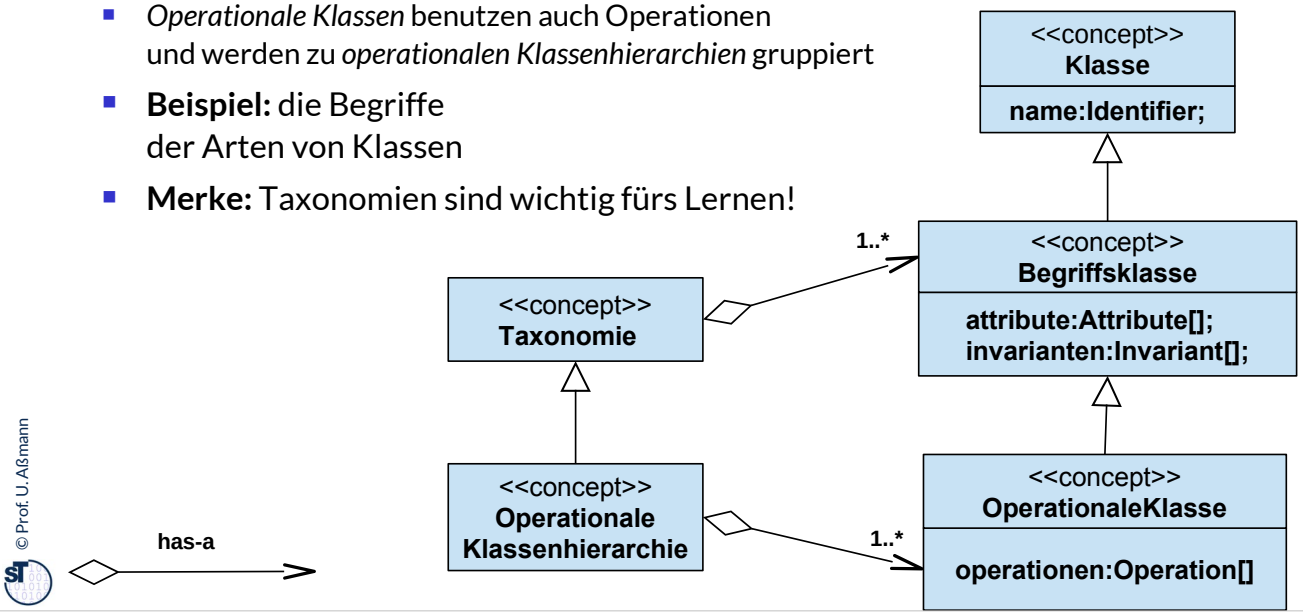
"Der einzige Weg, auf welchem wahre Kenntnis erreicht werden kann, ist durch liebesvolles Studium."

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

http://www.aphorismen.de/

Q1: Begriffshierarchien (Taxonomien) nutzen einfache Vererbung

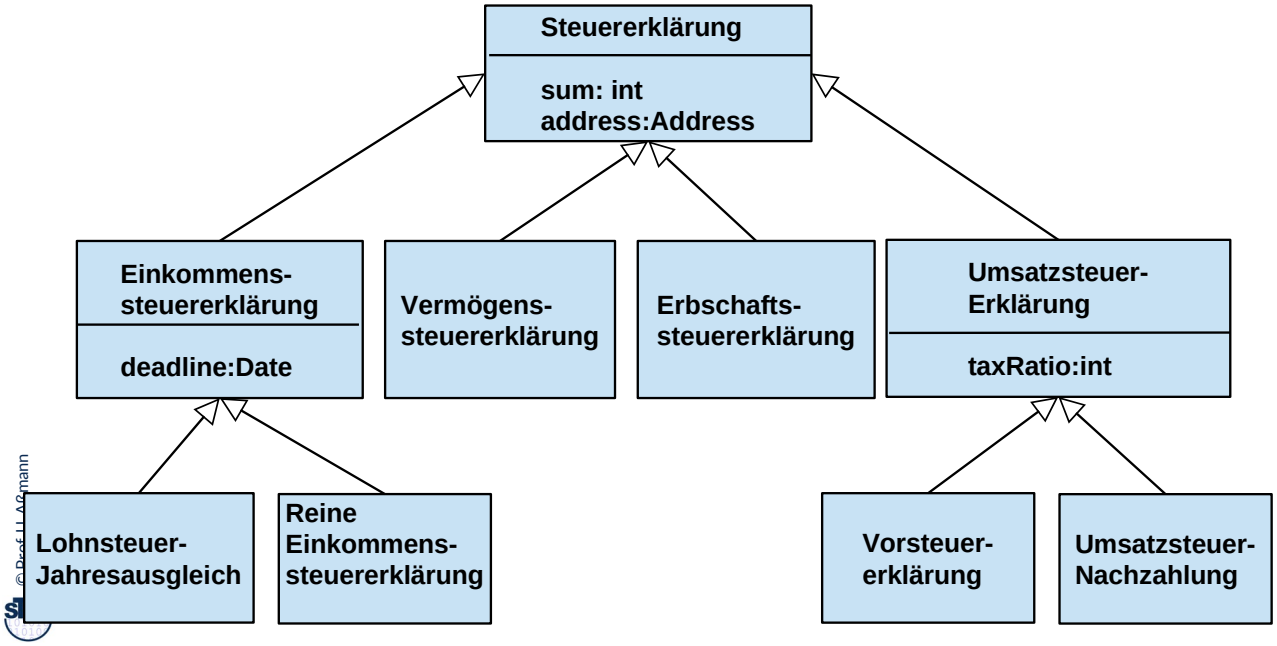
- Domänenmodelle werden durch *Klassifikation* der Domänenobjekte und Domänenkonzepte ermittelt
- Klassifikationen führen zu ***Begriffshierarchien (Taxonomien)***
  - *Begriffsklassen* besitzen nur Attribute und Invarianten (leicht blau)
- *Operationale Klassen* benutzen auch Operationen und werden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



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

Bsp. Taxonomie der Steuererklärungen im fachlichen Modell “Steuererklärung”

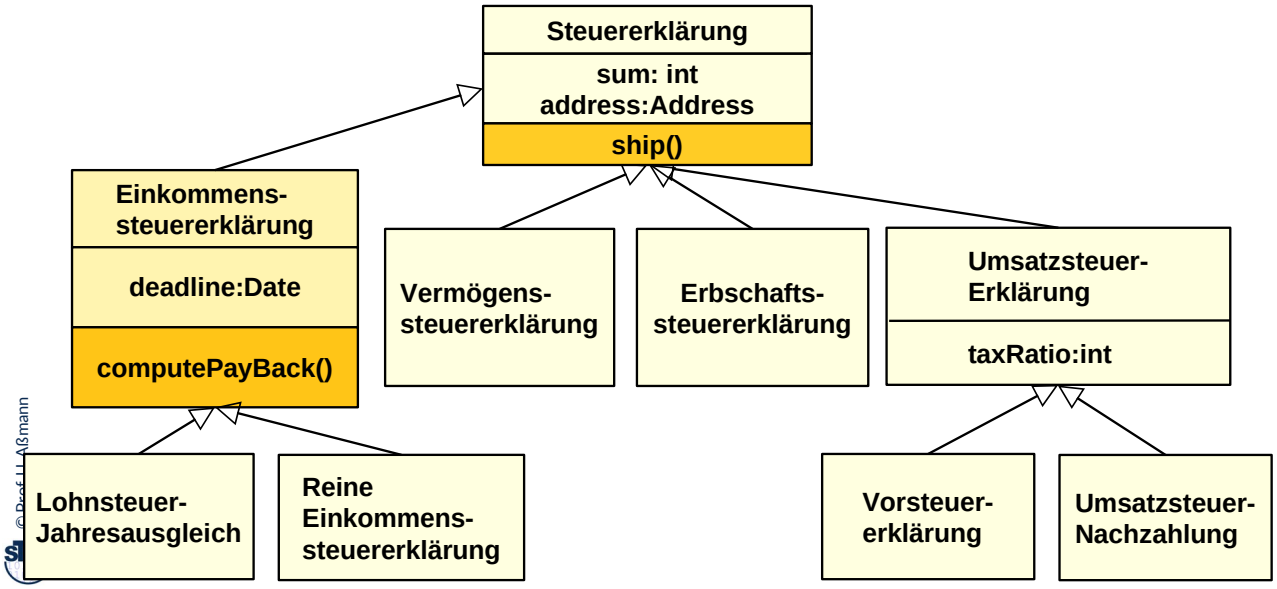
- Domäne: Finanzbuchhaltung: Das deutsche Steuerrecht kennt viele Arten von Steuererklärungen
- Eine Klassifikation führt zu einer Begriffshierarchie
- Warum haben Informatiker durch ihr Verständnis von Begriffshierarchien große Vorteile im Leben?



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.

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
- Die Strukturkarte enthält alle **Begriffshierarchien,** die in der VL diskutiert wurden
- **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: ProQuest eBooks Online

- **Achtung, neuer e-Book-Service unserer Bibliothek SLUB:**
- **http://www.dbod.de/db/start.php?database=ebl_ebl (DBoD)**
- **https://www.slub-dresden.de/recherche/datenbanken/erweitertes-angebot-an-e-medien-waehrend-covid-19/ (Alle)**

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

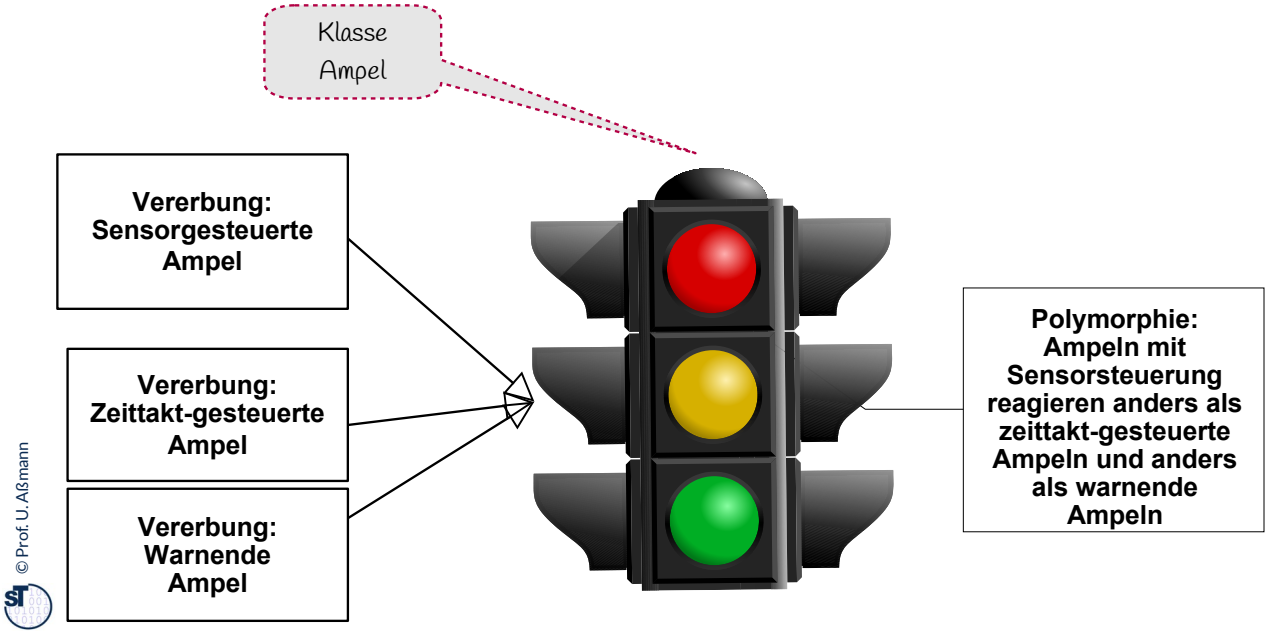
- Stickel-Wolf, Wolf. Wissenschaftliches Arbeiten und Lerntechniken. Gabler. Blau. Sehr gutes Überblicksbuch für Anfänger.
- **Kurs "Academic Skills in Software Engineering" (2/2/0)**
  - Sommersemester
  - https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/2307165920?20
  - https://tu-dresden.de/ing/informatik/smt/st/studium/lehrveranstaltungen?leaf=1&lang=en&subject=418&embedding_id=47eddfa7c5a54ed5be49042aff35a31b

11.3. Polymorphie (Wechsel der Gestalt)
.. verändert das Verhalten einer Anwendung, ohne den Code zu verändern

- Polymorphie erlaubt *dynamische Architekturen*
  - Dynamisch wechselnd
  - Unbegrenzt viele Objekte
- Polymorphie erlaubt die Spezifikation von *Lebenszyklen von Objekten*
- 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)?

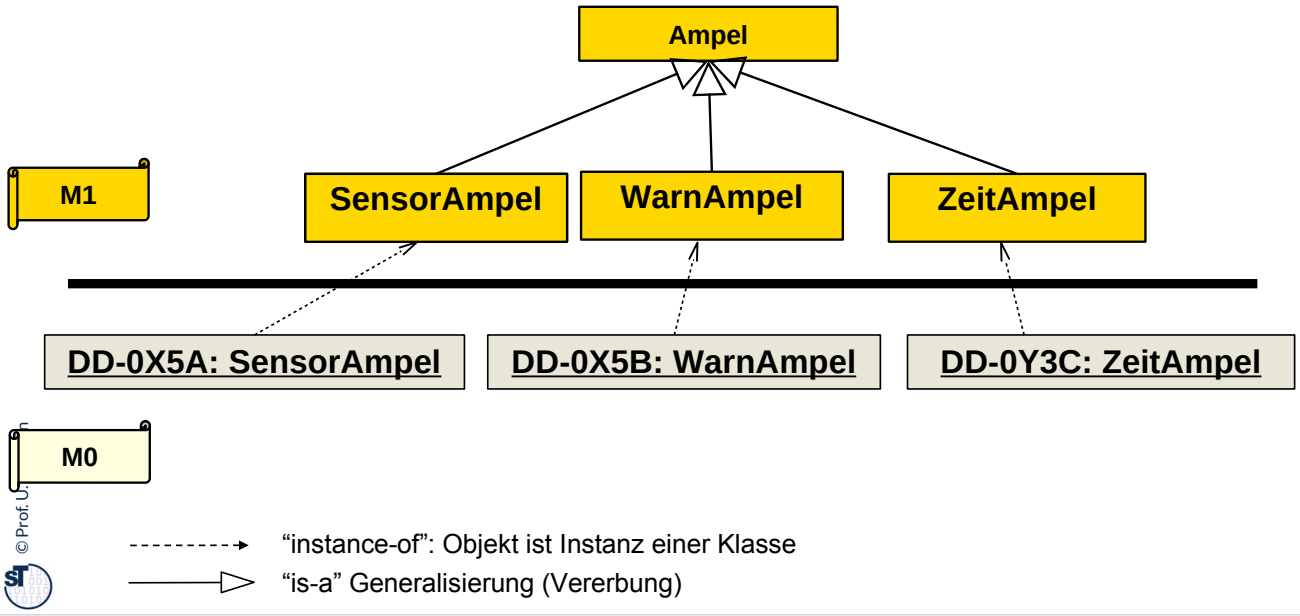


Ampeln können verschiedene Steuerungsalgorithmen haben (zeitgetaktet, sensorgesteuert, warnend- blinkend). Diese Algorithmen können über die Phasen des Tages, dem Lebenszyklus der Ampel, wechseln.

Beispiel: Der Lebenszyklus von Ampeln

Jede Ampel schaltet auf eine spezifische Weise

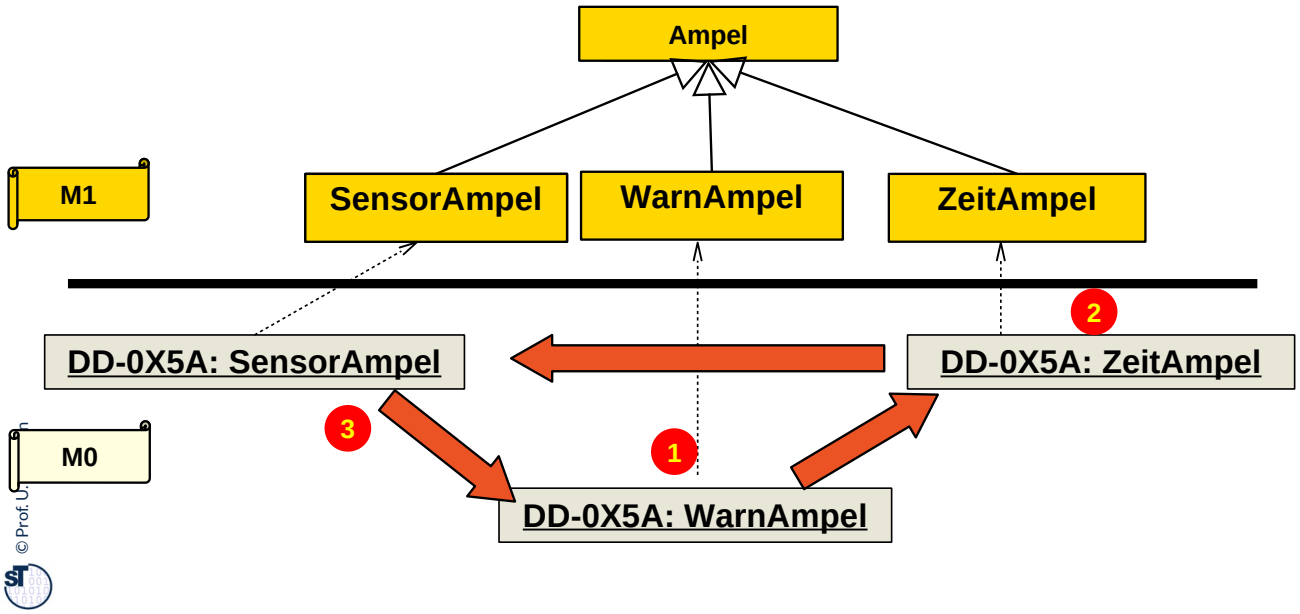
- Die Klasse **ZeitAmpel** schreibt vor, daß auf die Nachricht „Zeittakt“ mit Schalten reagiert werden muss
- Die Klasse **SensorAmpel** schreibt vor, auf das Sensorereignis “Auto kommt an” geschaltet werden muss
- Die Klasse **WarnAmpel** schreibt nur vor, dass geblinkt wird



Man sieht hier das Zusammenspiel eines Objekt- mit einem Klassendiagramm.

Beispiel: Lebenszyklus von Ampeln (Polymorphie)

- Ampeln folgen ***Lebenszyklen***: nachts blinken sie, zur Rushhour sind sie Zeit-getaktet, und ansonsten sensor-getrieben

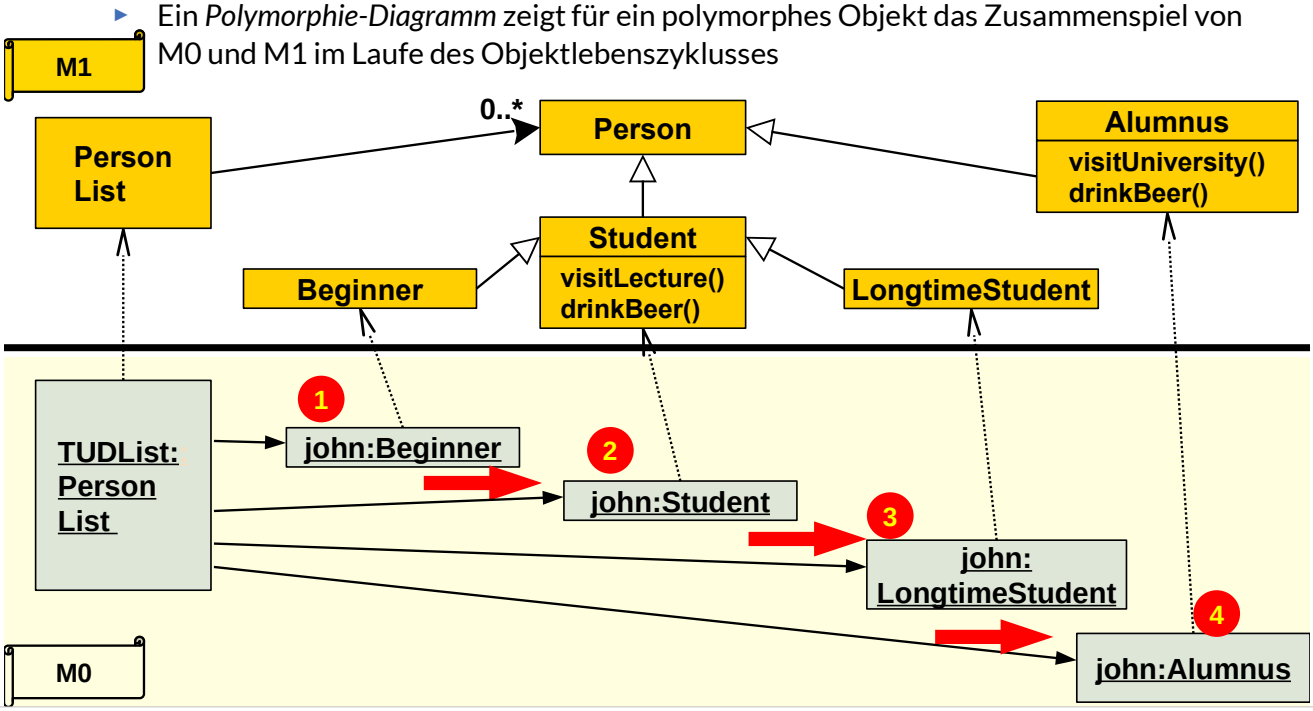


Während M1, die Klassenebene während der Ausführung eines Programms gleich bleibt, ändert sich der Snapshot des Objektnetzes auf M0 ständig (Lebensphase).

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

Vorteil: Polymorphie als Wechsel von Phasen des Lebens eines Objekts (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)

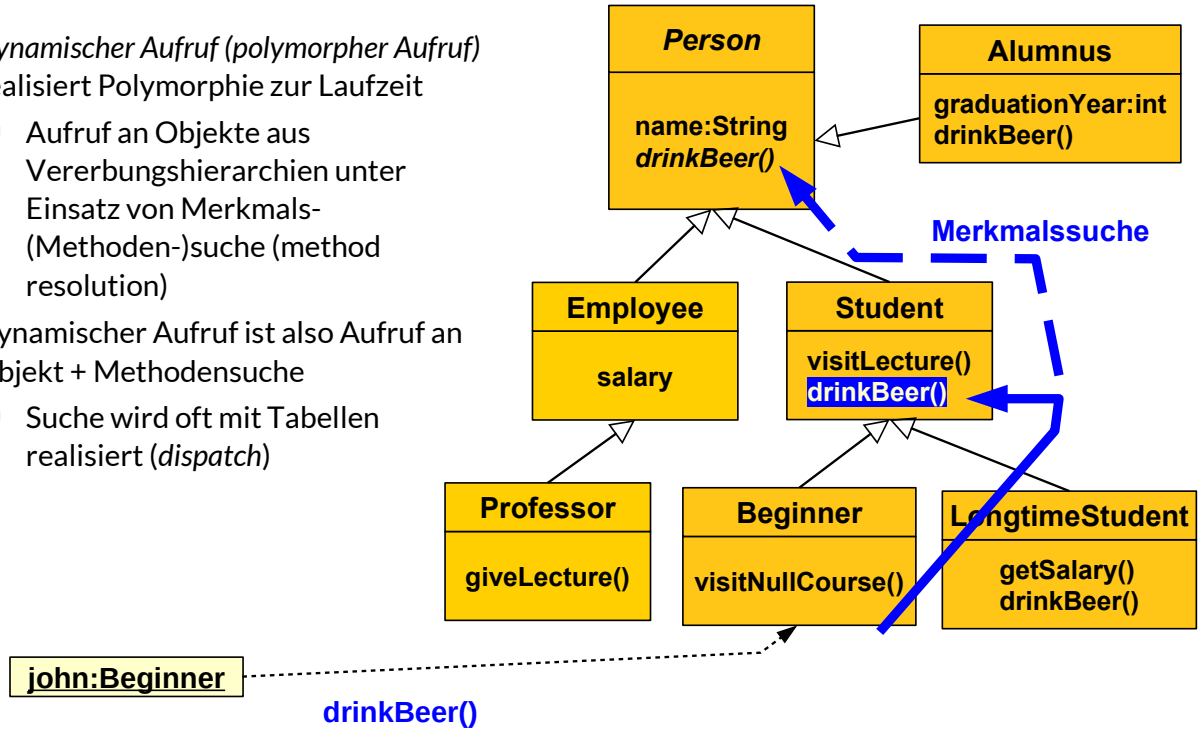
- Die genaue Unterklasse einer Variablen wird festgelegt
  - Beim **Erzeugen** (der Allokation) des Objekts (Allokationszeit, oft in der Aufbauphase des Objektnetzes), oft in einem alternativen Zweig des Programms alternativ festgelegt
  - Bei einer **neuen Zuweisung** (oft in einer Umbauphase des Objektnetzes)

```
Data data; Person john;

if (data.hasLeftUniversity())
   john = new Alumnus();
else if (data.getYear()==1)
   john = new Beginner();
else if (data.getYear()>5)
   john = new LongtimeStudent();
else
    john = new Student();

if (data.hasHabilitated())
   john = new Professor();

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

Vorteil: Dynamischer Aufruf (Polymorpher 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*)

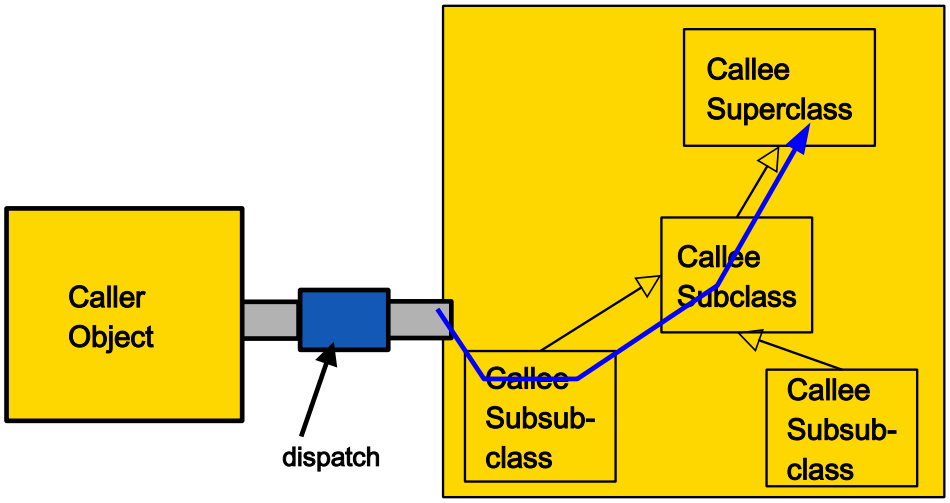


Ruft man in einem objektorientierten Programm eine Methode auf, muss diese sich nicht in dem lokalen Objekt befinden, sondern kann in einer Oberklasse stecken. Daher muss man nach Merkmalen dynamisch im Vererbungsbaum suchen.

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 (Polymorpher Aufruf, dynamic dispatch)

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

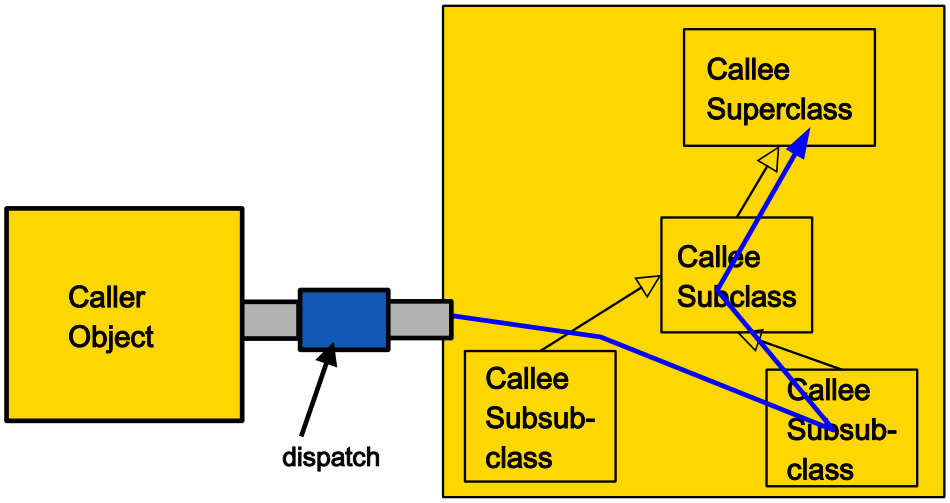


Der Typ eines Aufgerufenen (bzw. des Empfängers einer Botschaft) kann innerhalb des Vererbungsbaumes variieren. Der Typ des Aufgerufenen ist *vielgestaltig* (Polymorphie).

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)*.

Dynamischer Aufruf (Polymorpher Aufruf, dynamic dispatch)

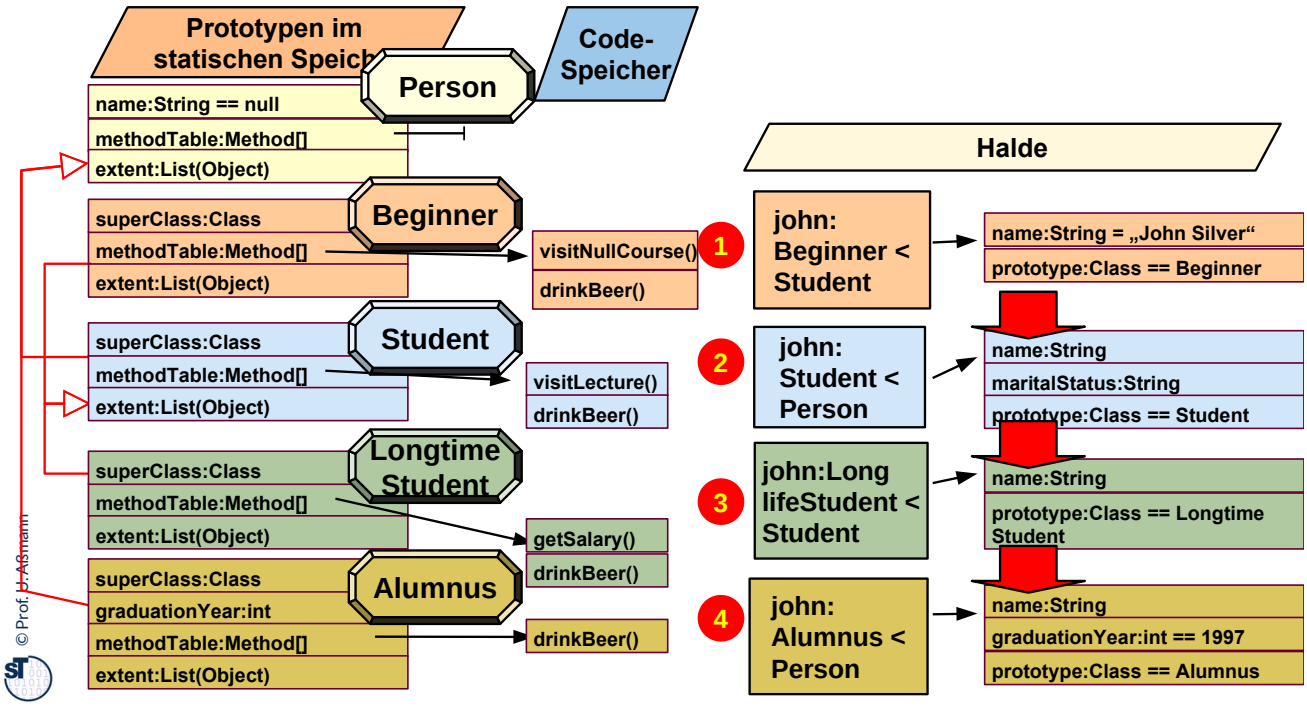
- Dynamischer Wechsel des Typs des Objekts möglich (z.B. auf Schwesterklasse)



Bei Gültigkeit des LSP merkt die Umgebung nichts davon, dass der Callee polymorph ist

Was passiert beim polymorphen Aufruf im Speicher?

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



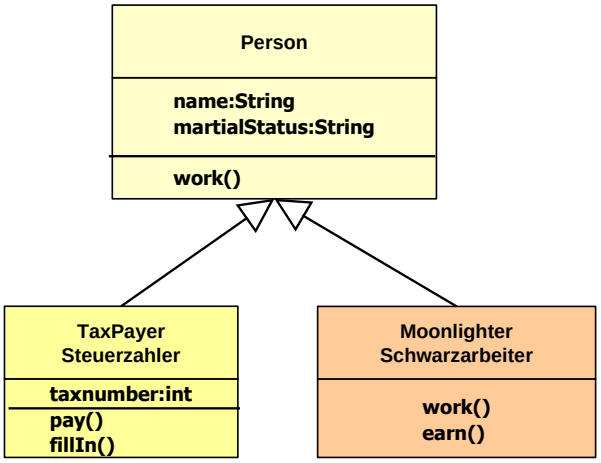
Über der Zeit können Objekte ihren Typ innerhalb des Vererbungsbaumes *ändern (Objektevolution)*, hier durch Farben ausgedrückt. Daher ändern u.U. die aufgerufenen Methoden ihre Natur (Polymorphie).

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 (*statischer Aufruf*). 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

Was haben wir gelernt?

- Codeverschmutzung wird vermieden durch Vererbung
  - Vererbung erlaubt die *Wiederverwendung* von Merkmalen aus Oberklassen,
  - Einfache Vererbung führt zu *Vererbungshierarchien*
- 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 der Name der zu rufenden Methode statisch bekannt ist und der Compiler die Merkmalssuche eingesparen 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?

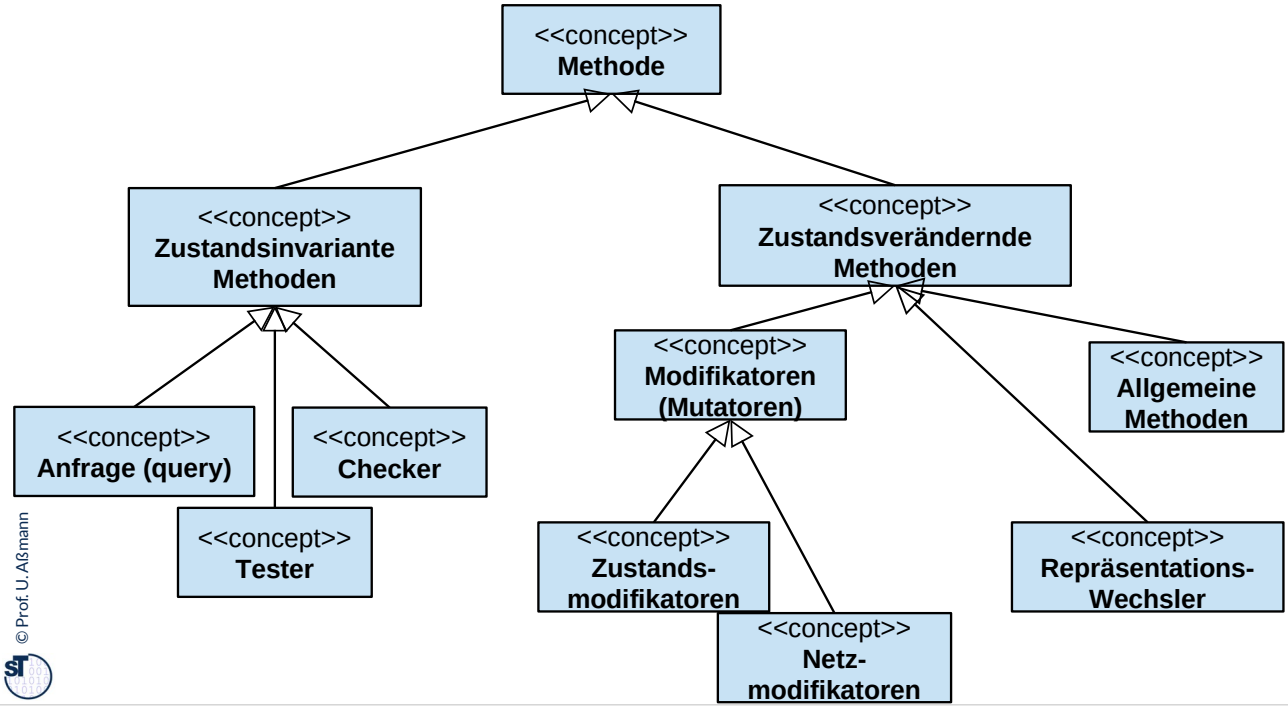
- **Wiederverwendung** ist eines der Hauptprobleme des Software Engineering
  - In einem Programm
  - Von Projekt zu Projekt
  - Von Produkt zu Produkt (Produktfamilien, Produktlinien)
- Wiederverwendung ist das Hauptmittel der Softwarefirmen, um **profitabel** arbeiten zu können:
  - Schreibe und teste 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?
- Was bedeutet der Begriff “Refactoring”? Welche Vorteile bietet er?
- Stellen Sie ein Polymorphie-Diagramm über die 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?
- Welche wesentliche Vorteile hat ein Informatiker in seinem Leben, der das Vererbungskonzept verstanden hat?

Bsp: Begriffshierarchie (Taxonomie) der Methodenarten

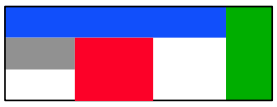
- **Wiederholung**: Welche Arten von Methoden gibt es in einer Klasse?
- **Antwort**: Begriffshierarchie:



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

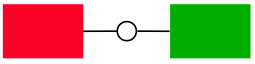
Die Klassifikation ist wichtig, weil die verschiedenen Methoden den Objektzustand ihres Objekts unterschiedlich beeinflussen.

Prinzipielle Vorteile von Objektorientierung



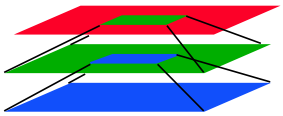
Zuständigkeitsbereiche

Lokalität: Lokale Kapselung von Daten und Operationen, gekapselter Zustand



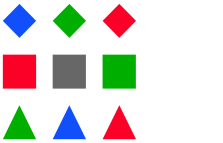
Klare Schnittstellen

Typen und Typsicherheit
Definiertes Objektverhalten,
Nachrichten zwischen Objekten



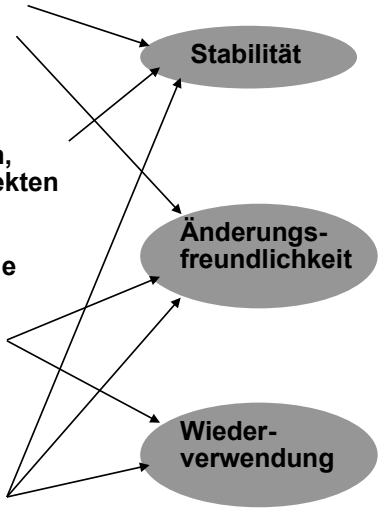
Hierarchie

Vererbung und Polymorphie (Spezialisierung),
Wiederverwendung
Klassenschachtelung



Baukastenprinzip

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



Q10: Relationen