Teil I: Einführung in die objektorientierte Programmierung mit Java
10. Objekte und Klassen

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
Technische Universität Dresden
Version 16-0.2, 25.03.16

Obligatorische Literatur

- Marc Andreesen. Software is eating the world. Wall Street Journal. http://online.wsj.com/news/articles/SB10001424053111903480904576512250915629460
- Störrle: Kap. 3 UML im Kontext (STfE S. 47); Kap. 5 Klassendiagramme (STfE S. 73)
- Zuser Kap 7, Anhang A
- Java: Balzert, LE 3-5
  - Boles, Kap. 1-6

Weiterführende Literatur

- Turing Award winners Ole-Johan Dahl, Kristen Nygaard
  - http://de.wikipedia.org/wiki/Ole-Johan_Dahl
  - http://www.mn.uio.no/ifi/english/about/ole-johan-dahl/
  - http://de.wikipedia.org/wiki/Kristen_Nygaard
  - http://de.wikipedia.org/wiki/Simula
  - http://campus.hesge.ch/Daehne/2004-2005/Langages/Simula.htm
  - http://simula67.at.ifi.uio.no/bibliography.shtml
- Alan Kay. The History of Smalltalk. Second Conference on the History of Programming Languages. ACM. http://portal.acm.org/citation.cfm?id=1057828
- The When, Why, and Why Not of the BETA Programming Language http://dl.acm.org/citation.cfm?id=1238854
- Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. Pages: 4-1-4-59, doi>10.1145/1238844.1238848

Online-Java-Ressourcen

- Über die Homepage der Lehrveranstaltung finden Sie auch Beispielprogramme, die Konzepte der Folien weiter erklären.
- Die Datei **`TaxDeclarationDemo.java`** enthält den Aufbau einer einfachen Halde mit Objekten
- Die Datei **`Terminv.java`** enthält eine vollständige Umsetzung des Beispiels "Terminverwaltung" in lauffähigen Java-Code.
- Empfohlene Benutzung:
  - Lesen
  - Übersetzen mit dem Java-Compiler `javac`
  - Starten mit dem Java-Interpreter (Java Virtual Machine) `java`
  - Verstehen
  - Modifizieren
  - Kritisieren

Ziele

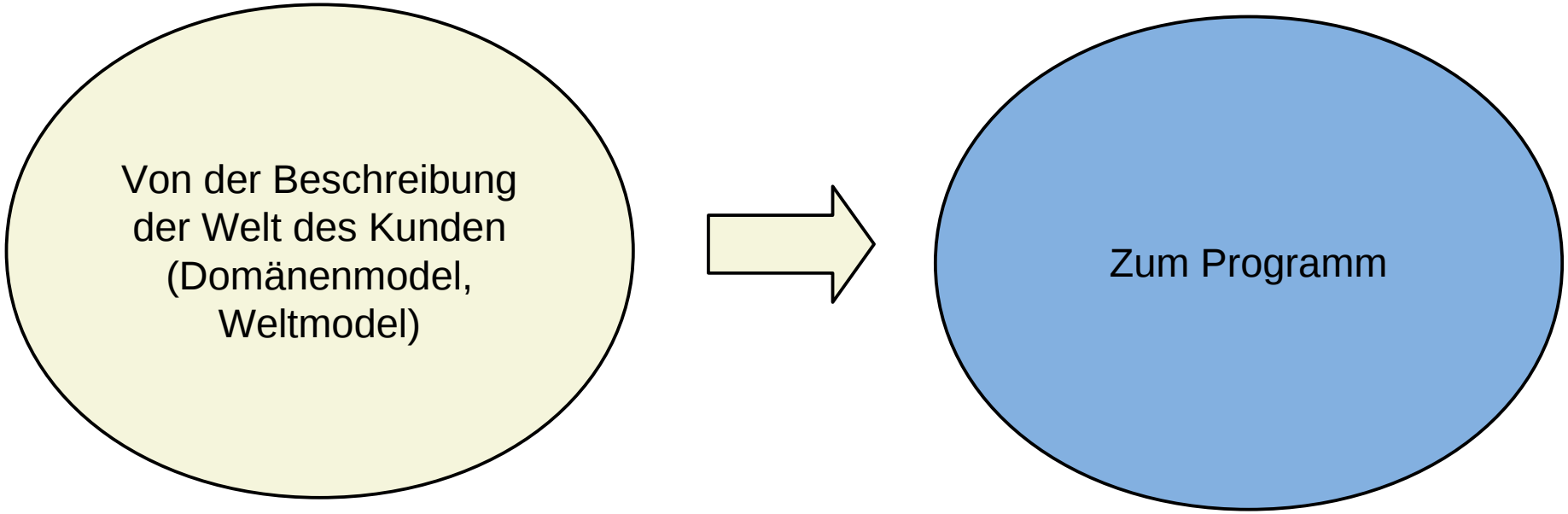
- Objekte verstehen als identitätstragende Entitäten mit Methoden und Zustand
- Klassen als Modellierungskonzepte verstehen
  - Mengen-, Schablonensemantik
- Verstehen, wie Objekte und Klassen im Speicher dargestellt werden

10.1. Objekte: die Idee

Objektorientierung bietet eine Grundlage für die Erstellung von Software

Die zentralen Frage der Softwaretechnologie

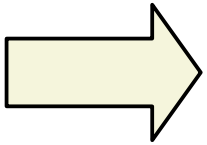
Wie kommen wir vom Problem des Kunden zum Programm (und zum Software-Produkt)?



Die zentralen Fragen des objektorientierten Ansatzes

Wie können wir die Welt möglichst einfach beschreiben?

Von der Beschreibung **der Objekte** der Welt des Kunden (objektorientiertes Domänenmodel)

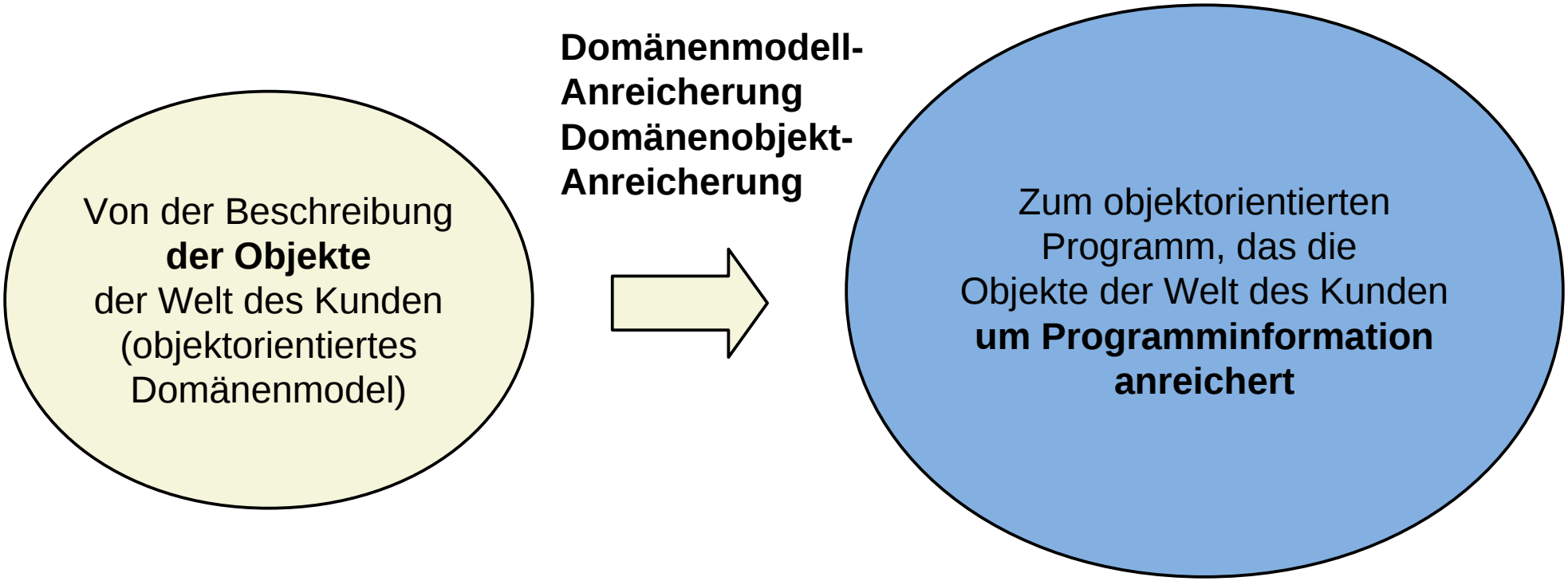


Zum objektorientierten Programm, das die Objekte der Welt des Kunden **um Programminformation anreichert**

Wie können wir diese Beschreibung im Computer realisieren?

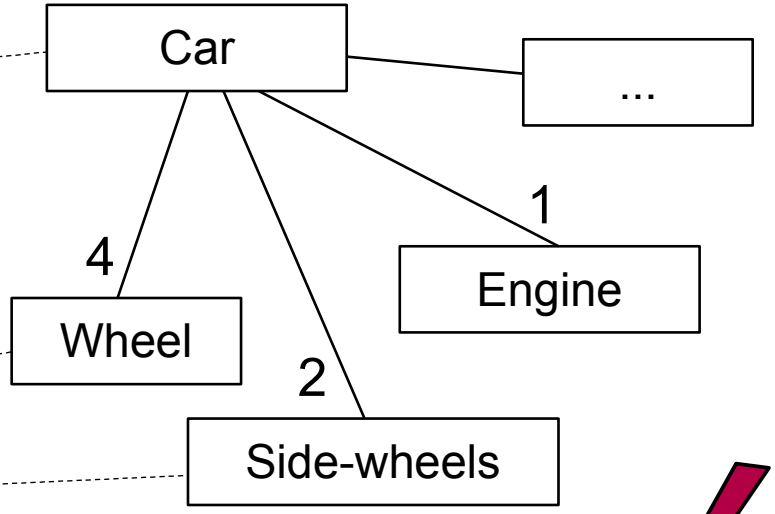
Die zentralen Fragen des objektorientierten Ansatzes

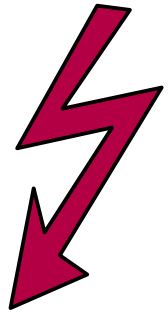
Wie kommen wir vom Problem des Kunden zum Programm (oder Produkt)?



Anreicherung/Verfettung („object fattening“): Anreicherung von Objekten des Domänenmodells durch technische Programminformation hin zu technischen Objekten

Modeling the Real World



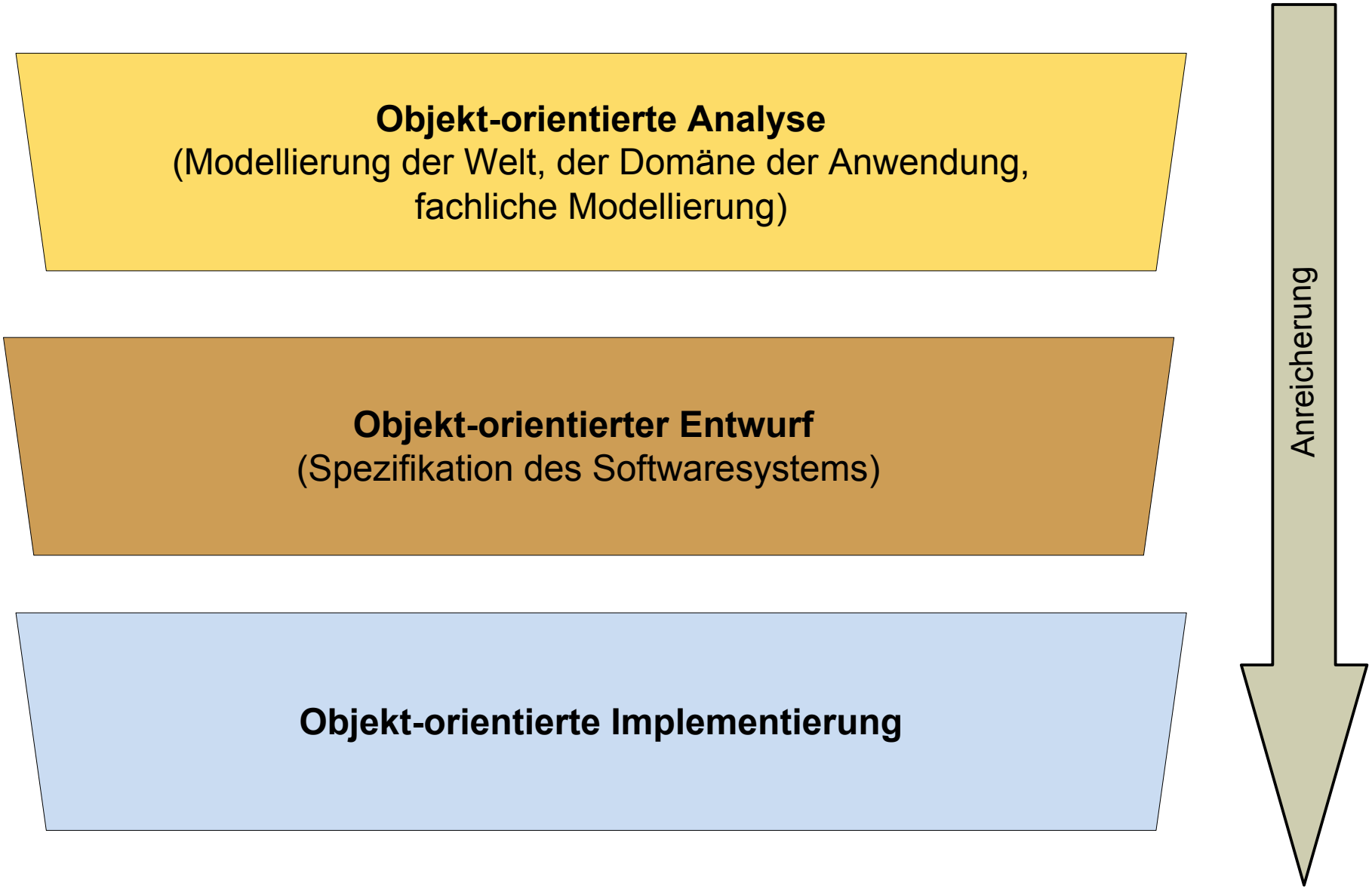


Objektorientierte Entwicklung ***modelliert*** die reale Welt, um sie im Rechner zu ***simulieren***
Ein Teil eines objektorientierten Programms simuliert immer die reale Welt, Indem es auf einem *angereicherten Domänenmodell* arbeitet

Herkunft:

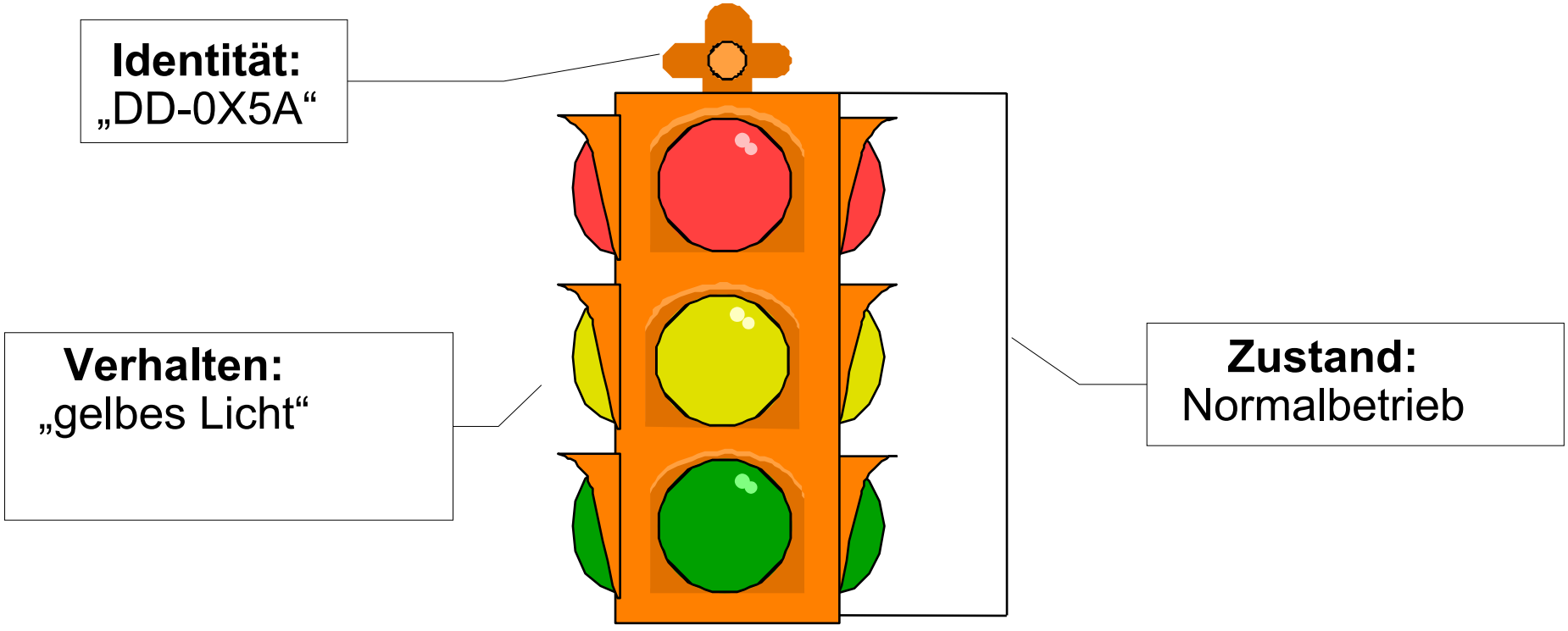
- Simula 1967 (Nygaard, Dahl). Ziel: technische Systeme simulieren
- Smalltalk 1973 (Kay, Goldberg)
- BETA 1973-1990 (Nygaard, Lehrman-Madsen, et.al.)

Der objekt-orientierte Softwareentwicklungsprozess in 3 Schritten



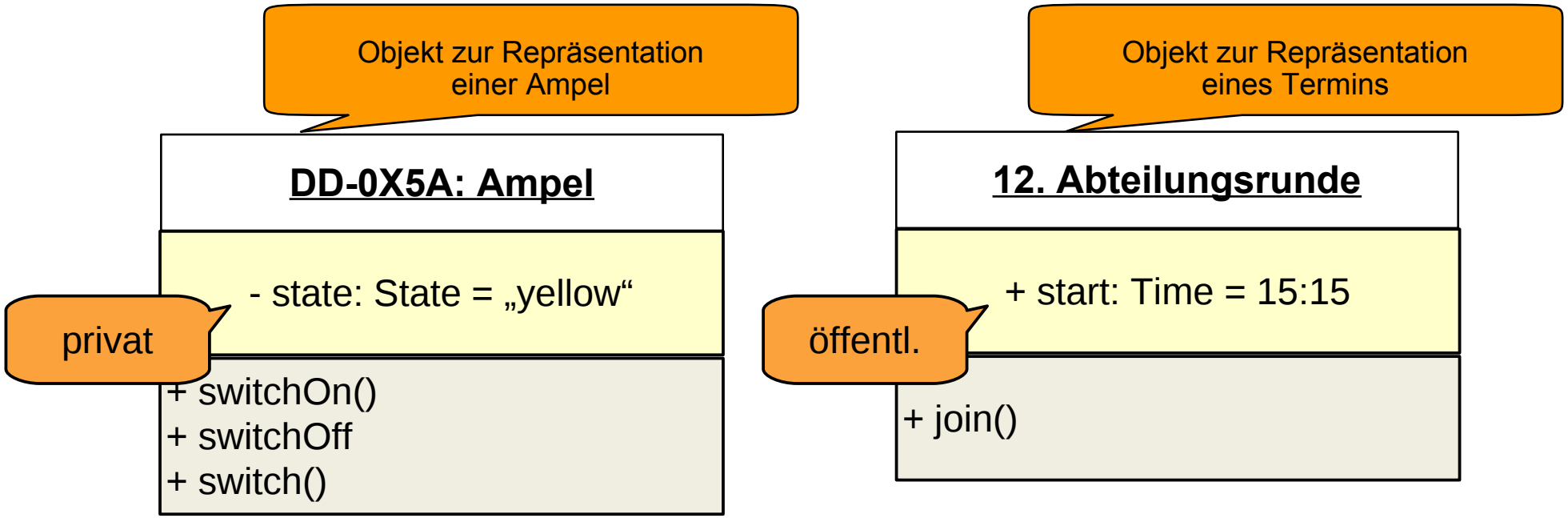
10.1.1 Grundkonzepte der Objektorientierung: Die Objekte einer Domäne

- Wie ist das Objekt bezeichnet (Name)?
- Wie verhält es sich zu seiner Umgebung? (Relationen)
- Welche Informationen sind „Privatsache“ des Objekts?
- Ein **System** besteht aus vielen **Objekten,** die miteinander interagieren und die aus einer Domäne übernommen sind



Grundkonzepte der Objektorientierung

- Ein Objekt hat eine eindeutige ***Identität*** (Schlüssel, key, object identifier)
- Ein Objekt hat einen inneren ***Zustand,*** ausgedrückt durch sog. *Attribute*, die Werte annehmen können (auch *Instanzvariablen* genannt, i.G. zu globalen Variablen wie in C)
- Ein Objekt hat ein definiertes ***Verhalten*** mit ***Schnittstelle***,
- Ein Objekt lebt in einem ***Objektnetz*** (in Relationen, "noone is an island")

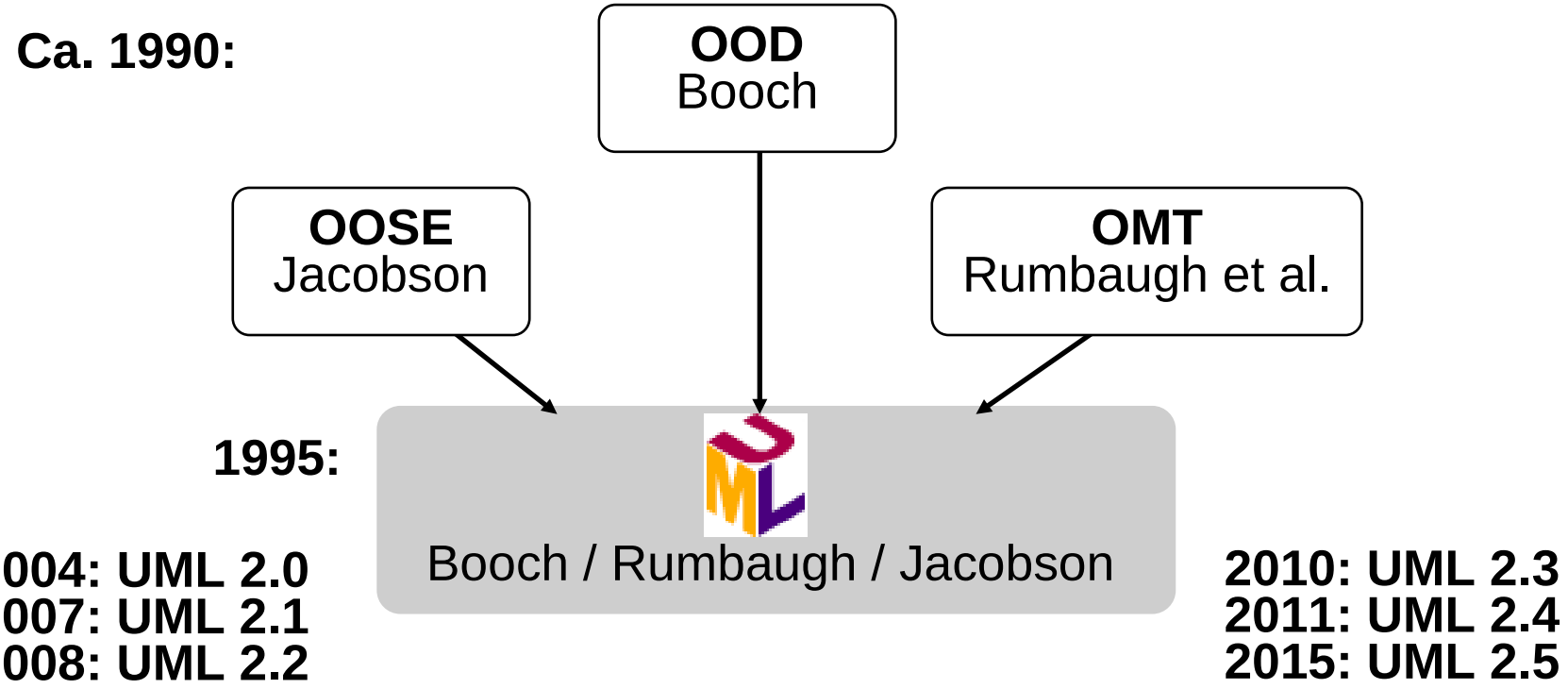


Zustandswechsel einer Ampel in Java

- Achtung: “class” ist ein spezielles “object”

```
class TrafficLight {
  private int state = { green, yellow, red, redyellow,
     blinking };
  public switch () {
    if (state == green) state = yellow;
    if (state == yellow) state = red;
    if (state == red) state = redyellow;
    if (state == redyellow) state = green;
  }
  public switchOn () {
    state = red;
  }
  public switchOff () {
    state = blinking;
  }
}
```

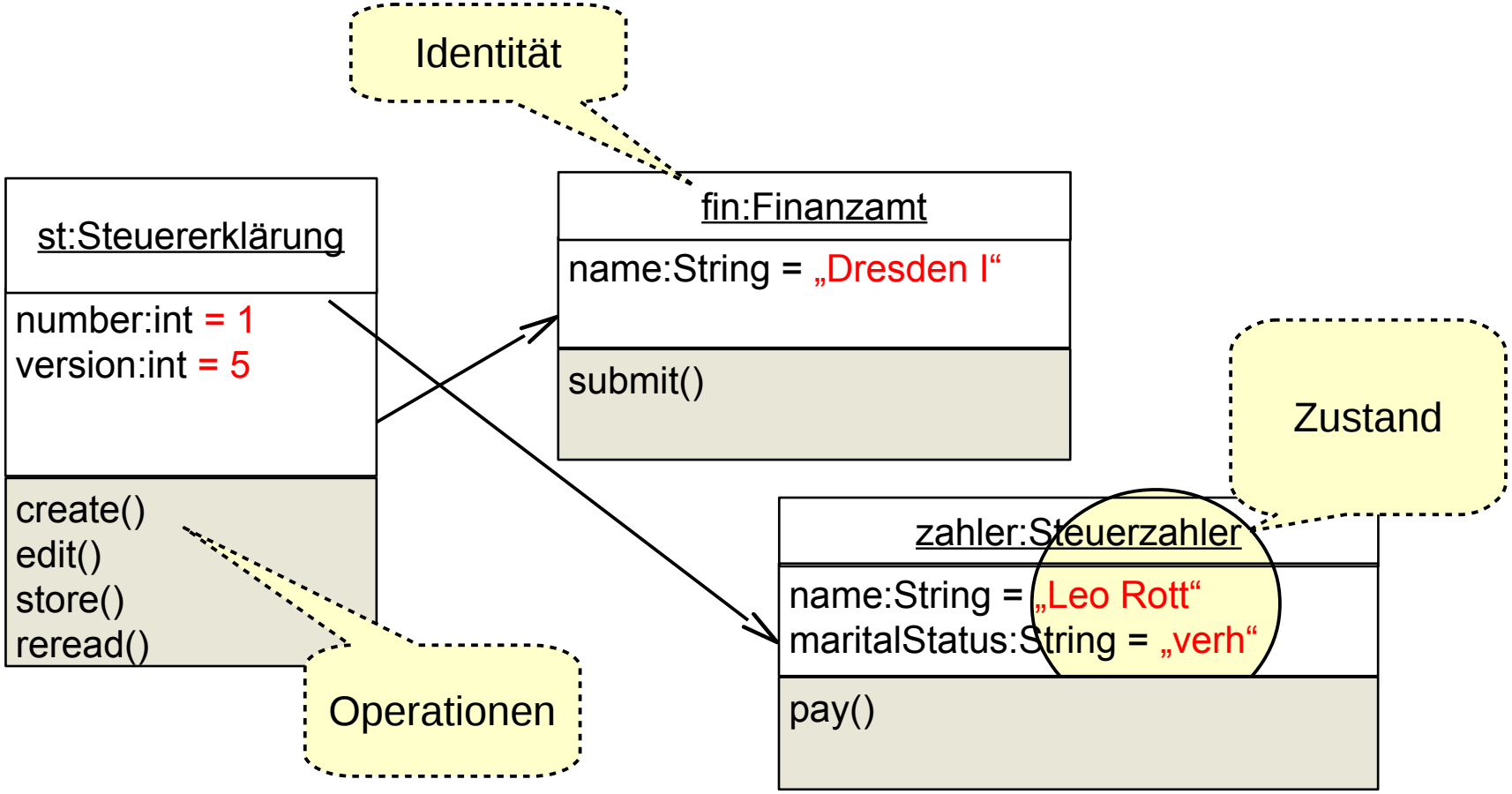
Modellierung mit der Unified Modeling Language UML



- UML ist eine objekt-orientierte diagrambasierte Modellierungssprache
  - Programmfragmente werden in Diagramm-Modellen ausgedrückt
- Industriestandard der OMG (Object Management Group)
  - http://www.omg.org/uml
  - http://en.wikipedia.org/wiki/Unified_Modeling_Language
- Achtung: wir verwenden hier jUML (Java-äquivalent), aUML (für Analyse), dUML (für Design).

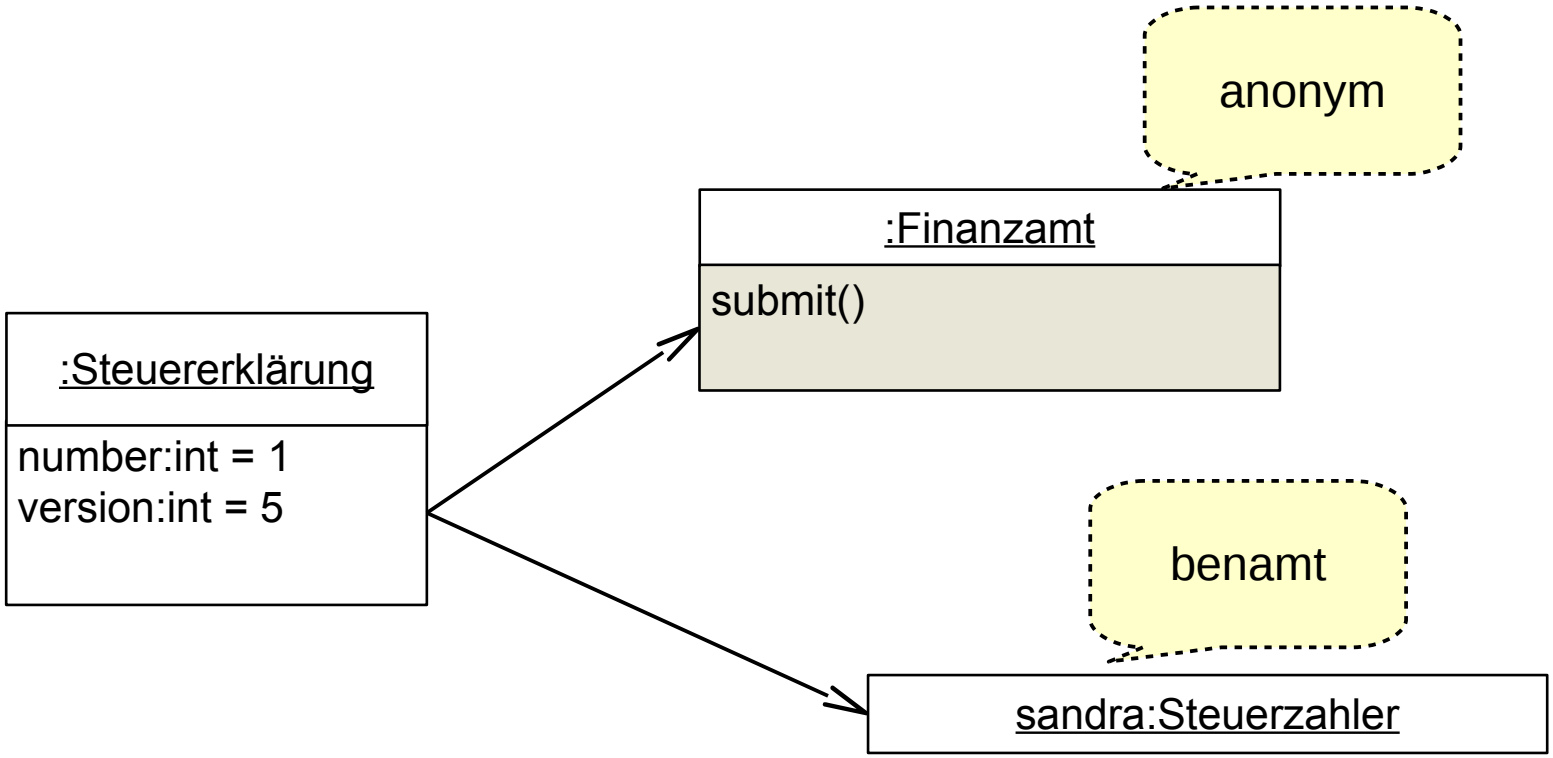
Wie sieht ein Objekt in (j)UML aus?

- Bsp.: Eine Einkommenssteuererklärung wird von einem Steuerzahler ausgefüllt. Sie wird zuerst mit Standardwerten initialisiert, dann in Iterationen ausgefüllt: abgespeichert und wiedergelesen. Zuletzt wird sie beim Finanzamt eingereicht.
- Die Werte aller *Attribute* bildet den *Objektzustand*
- Nutze das **UML Objektdiagramm**, um das Zusammenspiel von Objekten zu zeigen:



Wie sieht ein Objekt in UML aus?

- Attribute und/oder Operationen (Methoden, Funktionen, Prozeduren) können weggelassen sein
- Objekte können *anonym* sein oder einen Variablennamen (Identifikator) tragen



Das Hauptprogramm “Main”

- Eine Operation spielt eine besondere Rolle:
  `public static void main(String args[])`
  - wird vom Laufzeitsystem (Betriebssystem) des Computers immer als erste aufzurufende Operation identifiziert
  - Parameter: ein Array von Strings, das die Argumente, die auf der Kommandozeile übergeben werden, übergibt
- Aufruf von Shell:

```
$ java CommandLineDemo this is my first program
this
is
my
first
program
$
```

```
public class CommandLineDemo {
  public static void main(String args[]) {
     System.out.println(args[0]);
     System.out.println(args[1]);
     System.out.println(args[2]);
     System.out.println(args[3]);
     System.out.println(args[4]);
   }
}
```

Analyse: Was ist gefährlich bei diesem Programm?


```
$ java CommandLineDemo this is my first
this
is
my
first
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
at CommandLineDemo.main(CommandLineDemo.java:22)
$
```

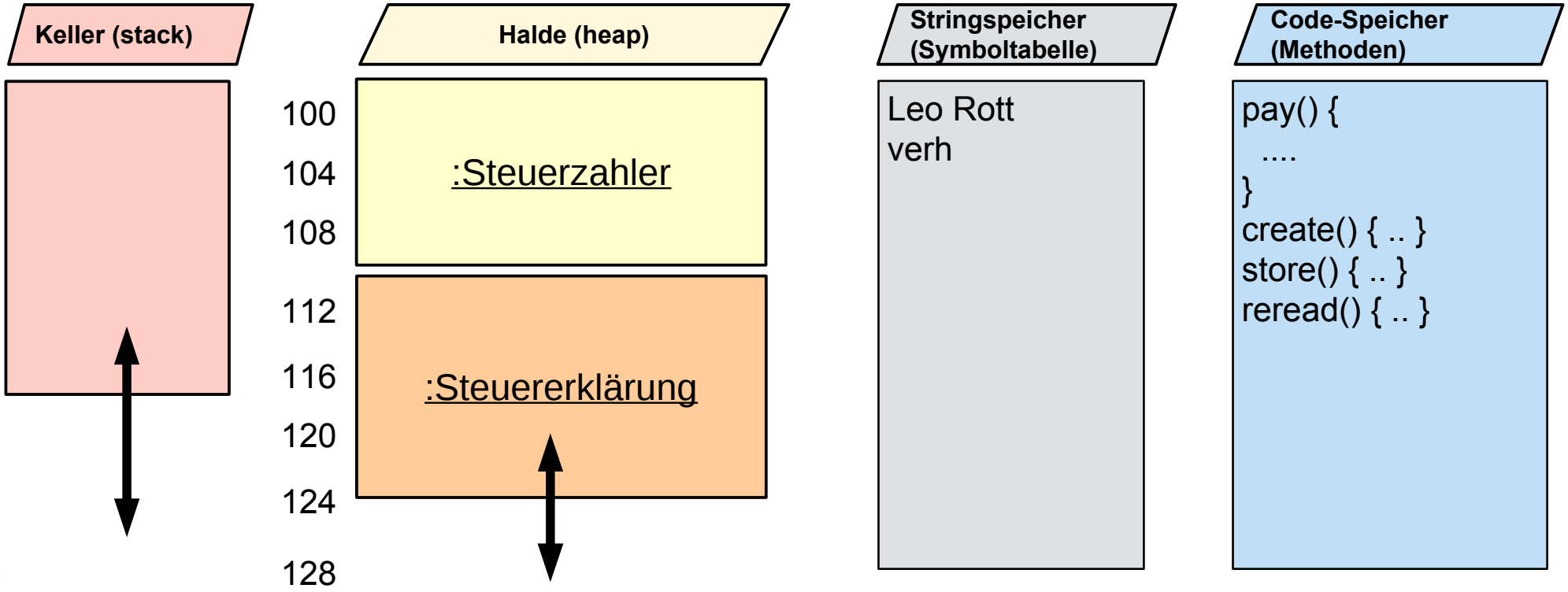
```
public class CommandLineDemo {
  public static void main(String args[]) {
     System.out.println(args[0]);
     System.out.println(args[1]);
     System.out.println(args[2]);
     System.out.println(args[3]);
     System.out.println(args[4]);
   }
}
```

Heim-Übung: Erstes Programm

- Lade Datei **`CommandLineDemoLoop.java`** von der Webseite
- Lese und analysiere die Datei:
  - Wie stellt man fest, dass ein Array eine bestimmte Länge hat?
  - Wie wird etwas auf dem Terminal ausgegeben?
  - Wozu der import von `java.util.*`?
- Übersetze die Datei mit javac
- Führe das Programm aus (Vorsicht, nur den Klassennamen benutzen)

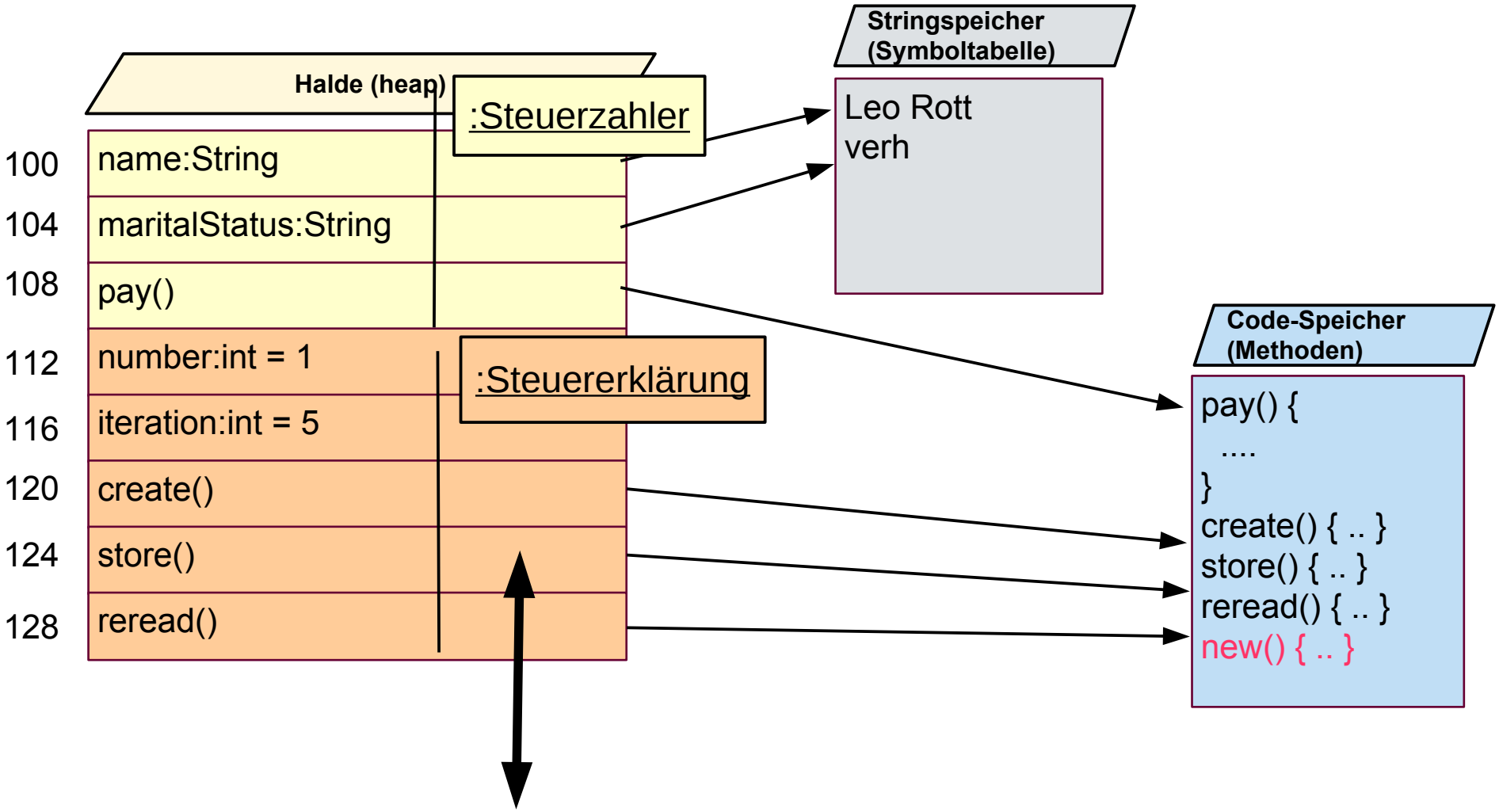
Wie sieht ein Objekt im Speicher aus? (vereinfacht)

- Der Speicher setzt sich zusammen aus
  - **Halde** (heap, dynamisch wachsender Objektspeicher)
  - **Keller** (stack, Laufzeitkeller für Methoden und ihre Variablen)
  - **Symboltabelle** (symbol table, halbdynamisch: statischer und dynamischer Teil, enthält alle Zeichenketten *(Strings)*)
  - **Code-Speicher** für Methoden (statisch, nur lesbar)
  - **Klassenprototypen** für Klassen (statisch, nur lesbar, später)



Wie sieht ein Objekt im Speicher aus? (vereinfacht)

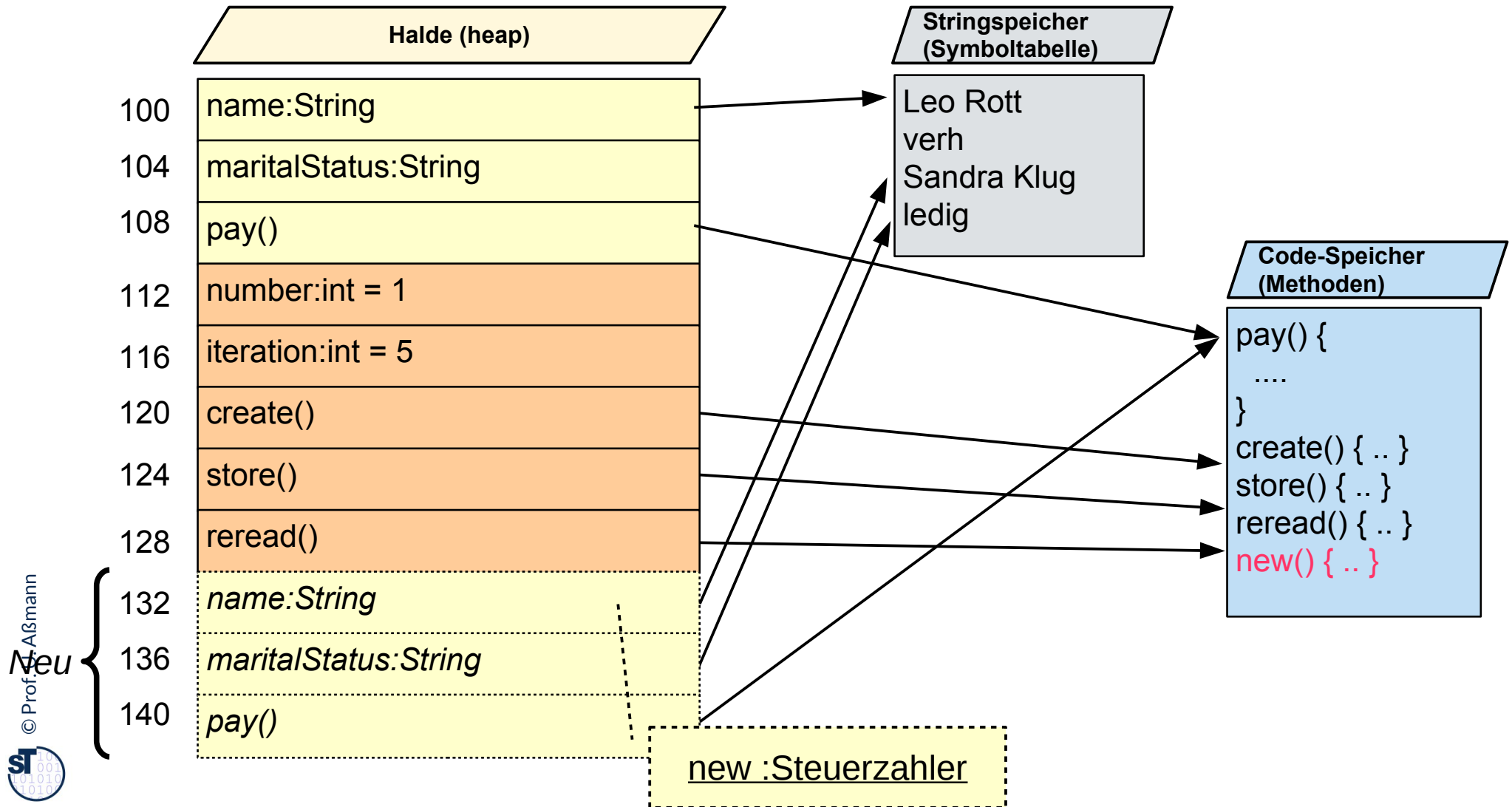
- Die Halde wächst *dynamisch* mit wachsender Menge von Objekten
  - verschiedene Lebenszeiten



Was passiert bei der Objekterzeugung im Speicher?

- Der *Allokator (Objekterzeuger)* `new()` spielt eine besondere Rolle:

```
zahler2 = new Steuerzahler("Sandra Klug", "ledig");
```



Heim-Übung: Ampeln

- Lade Datei `TrafficLightDemo.java` von der Webseite
- Lese und analysiere die Datei:
  - Wie bestimmt man die Zahl der erzeugten Ampeln?
  - Identifiziere den Konstruktur der Klasse
  - Wie wird der Konstruktor eingesetzt?
  - Wie schaltet man eine Ampel weiter?
- Übersetze die Datei mit javac
- Führe das Programm aus (Vorsicht, nur den Klassennamen benutzen)

Allokation und Aufruf eines Objektes in Java

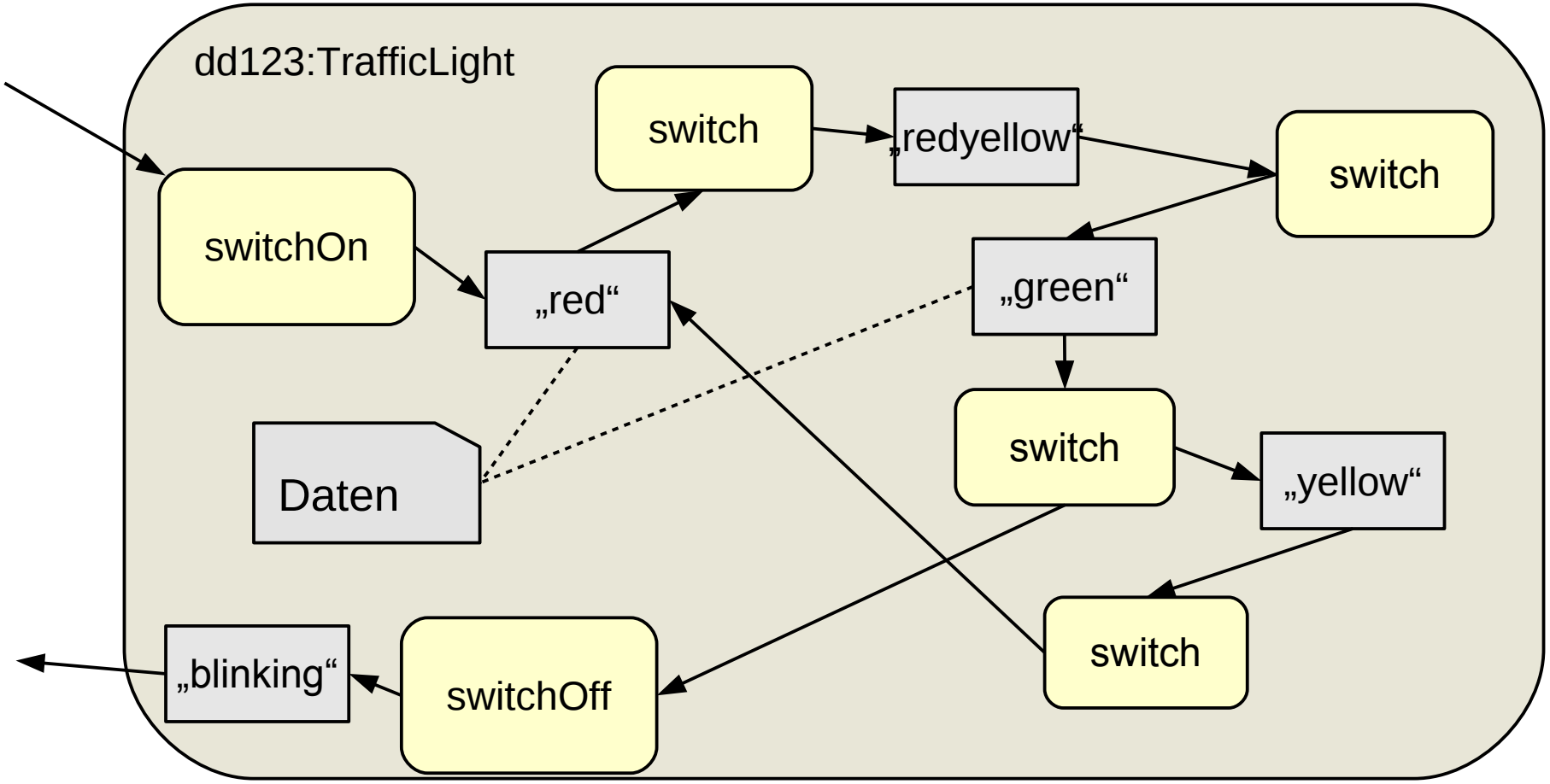
- Objekte durchlaufen im Laufe ihres Lebens viele Zustandsveränderungen, die durch Aufrufe von Operationen verursacht werden
- Das objektorientierte Programm simuliert dabei den Lebenszyklus eines Domänenobjekts

```
// Typical life of a traffic light
TrafficLight dd123 = new TrafficLight();
   // Allokation: lege Objekt im Speicher an, initialer Zustand ist
  // „blinking“

dd123.switchOn();  // Zustand wird „red“
while (true) {
  dd123.switch();       // Yields: john.state == „redyellow“
  dd123.switch();       // Yields: john.state == „green“
  dd123.switch();       // Yields: john.state == „yellow“
  dd123.switch();       // Yields: john.state == „red“
}
```

Ein Objekt besteht aus einer Menge von Aktivitäten auf einem Zustand

- Die Reihe der Aktivitäten (Operationen) eines Objektes nennt man **Lebenszyklus.**
- Reihenfolgen von Aktivitäten kann man in UML mit einem **Aktivitätendiagramm** beschreiben



Was können Objekte eines Programms darstellen?

- **Simulierte Objekte** der realen Welt (der *Anwendungsdomäne*)
  - Ampeln, Uhren, Türen, Menschen, Häuser, Maschinen, Weine, Steuerzahler, Finanzamt, etc.
- **Simulierte abstrakte Dinge** der Gedankenwelt der Anwender (Anwendungsobjekte, fachliche Objekte, Geschäftsobjekte, business objects)
  - Adressen, Rechnungen, Löhne, Steuererklärungen, Bestellungen, etc.
- **Konzepte** und **Begriffe** der Gedankenwelt
  - Farben, Geschmack, Regionen, politische Einstellungen, etc.
  - Dann nennt man das Modell eine *Ontologie (Begriffsmodell)*
- Substantivierte **Handlungen**. Objekte können auch Aktionen darstellen
  - Entspricht der Substantivierung eines Verbs (*Reifikation*, *reification)*
  - sog. Kommandoobjekte, wie Editieren, Drucken, Zeichnen, etc.

10.2. Klassen

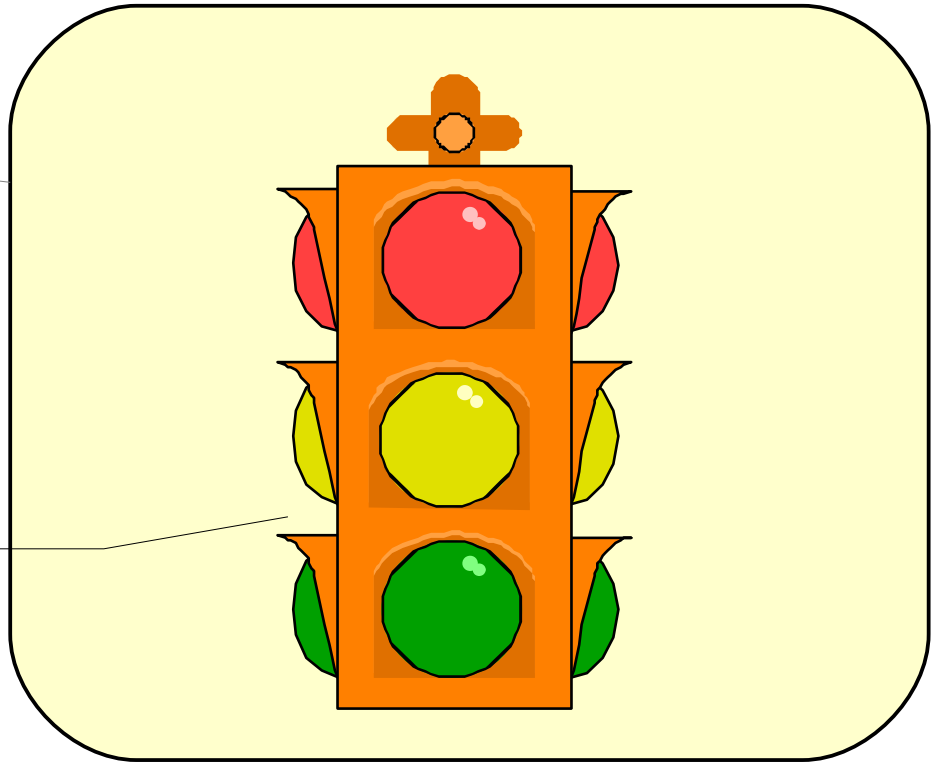
Der Begriff der “Klasse” leitet sich vom Begriff der “Äquivalenzklasse” der Mathematik ab

Objekte und ihre Klassen

- Ein weiterer Grundbegriff der Objektorientierung ist der der *Klasse*.
- *Fragen:*
  - Zu welcher Menge gehört das Objekt?
  - In welche Äquivalenzklasse einer Klassifikation fällt das Objekt?
  - Welcher Begriff beschreibt das Objekt?
  - Welchen Typ hat das Objekt?
  - Welches Verhalten zeigt das Objekt?

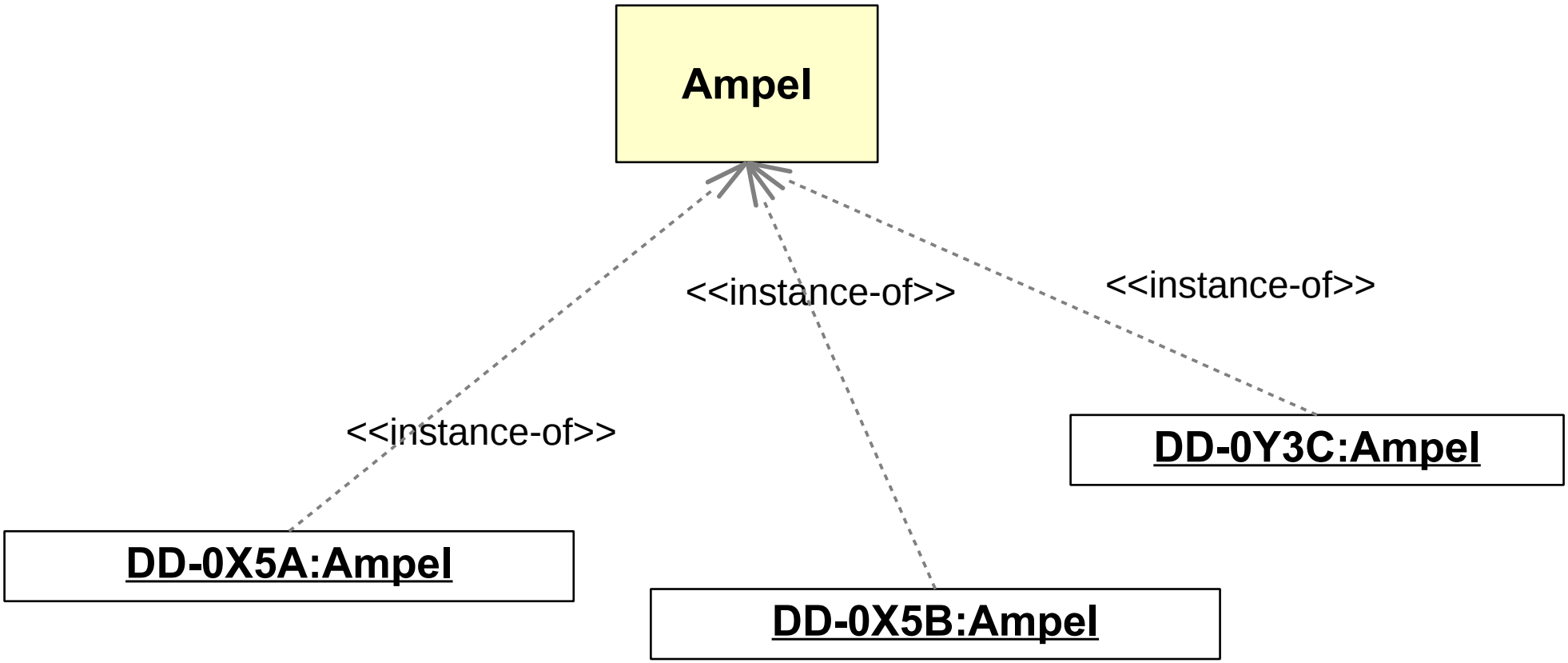
Klasse (Menge):
„Ampel“

Objekt:
„DD-0X5A“



Beispiel: Ampel-Klasse und Ampel-Objekte

- Objekte, die zu Klassen gehören, heißen *Elemente*, *Ausprägungen* oder *Instanzen* der Klasse



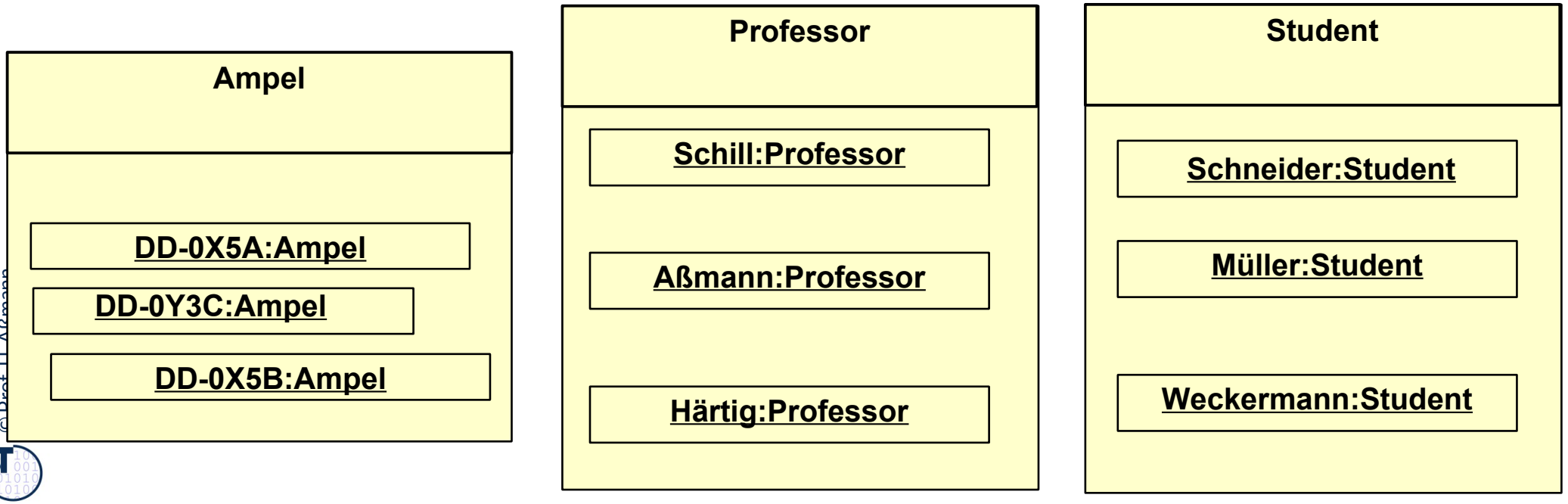
instance-of ----------> Ausprägung einer Klasse (Vorsicht: Das Wort *Instanz* ist Informatiker-Dialekt; es bedeutet im Deutschen eigentlich etwas Anderes)

Klassen als Begriffe, Mengen und Schablonen

- **Begriffsorientierte Sicht:** Eine Klasse stellt einen *Begriff* dar. Dann *charakterisiert* die Klasse ein *Konzept*
  - Man nennt das Modell eine *Begriffshierarchie*, *Begriffswelt* (*Ontologie,* gr. Lehre von der Welt)
- **Mengenorientierte Sicht:** (z.B. in Datenbanken) Eine Klasse *enthält* eine Menge von Objekten
  - Eine Klasse *abstrahiert* eine Menge von Objekten
  - `instance-of` bedeutet hier Enthaltensein
- **Schablonenorientierte (ähnlichkeitsorientierte) Sicht:**
  - Eine Klasse bildet eine *Äquivalenzklasse* für eine Menge von Objekten
  - Eine Äquivalenzklasse ist eine spezielle Menge, die durch ein gemeinsames Prädikat charakterisiert ist (Klassenprädikat, Klasseninvariante, Aquivalenzprädikat)
- **Strukturorientierte Sicht:** Die Klasse schreibt die innere **Struktur** vor (**Strukturäquivalenz**)
  - `instance-of` bedeutet hier Strukturgleichheit
  - Objekt erbt das Prädikat und die die Attribute der Klasse
  - Bildung eines neuen Objektes aus einer Klasse in der strukturorientierten Sicht:
    - Eine Klasse enthält einen speziellen Repräsentanten, ein spezielles Objekt, ein Schablonenobjekt (Prototyp) für ihre Objekte. Ein Objekt wird aus einer Klasse erzeugt (Objektallokation), indem der Prototyp in einen freien Speicherbereich kopiert wird.
- **Verhaltensorientierte Sicht:** Die Klasse kann zusätzlich das **Verhalten** ihrer Objekte vorschreiben (**Verhaltensäquivalenz**). Ein Objekt wird aus einer Klasse erzeugt (Objektallokation),
  - **Verhaltensäquivalenz:** Damit erbt das neue Objekt das Verhalten der Klasse (die Operationen), das Prädikat und die Attribute des Prototyps (meistens Nullwerte wie 0 oder null)
  - `instance-of` bedeutet also Verhaltensgleichheit mit den anderen Objekten der Klasse

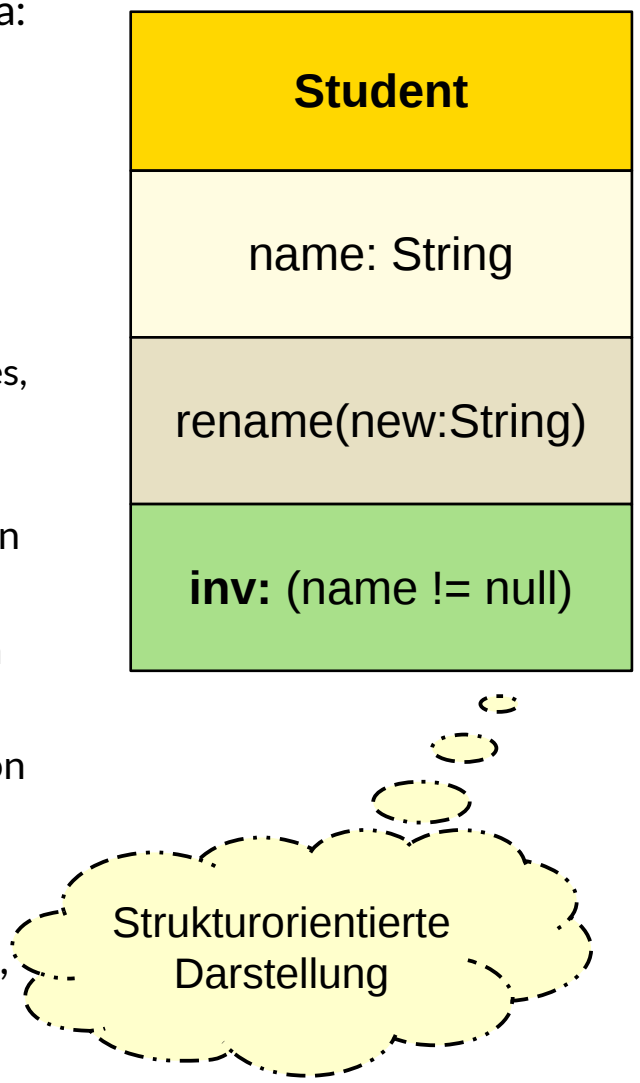
Klasse dargestellt als Venn-Diagramm (mengenorientierte Sicht)

- In der mengenorientierten Sicht bilden Klassen Flächen in einem Venn-Diagramm. Objekte werden durch Enthaltensein in Flächen ausgedrückt
  - Merke die Verbindung zu Datenbanken: Objekte entsprechen Tupeln mit eindeutigem Identifikator (OID, surrogate)
  - Klassen entsprechen Relationen mit Identifikator-Attribut
- In UML kann Schachtelung von Klassen und Objekten durch "UML-Komponenten (Blöcke)" ausgedrückt werden
  - Objekte bilden eine eigene Abteilung der Klasse (**Objekt-Extent**)



Strukturorientierte Sicht: Merkmale von Klassen und Objekten

- Eine Klasse hat ***Struktur- und Verhaltensmerkmale (features***, in Java: ***members)***, die sie auf die Objekte überträgt.
  - Damit haben alle Objekte der Klasse die gleichen Merkmale
  - Die Merkmale sind in ***Abteilungen (compartments)*** gegliedert:
  - ***Attribute*** (*attributes*, Daten) haben in einem Objekt einen objektspezifischen Wert
    - Die Werte bilden in ihrer Gesamtheit den *Zustand* des Objektes, der sich über der Zeit ändert.
  - ***Operationen*** (Methoden, Funktionen, Prozeduren, *functions*, *methods*) sind Prozeduren, die den Zustand des Objektes abfragen oder verändern können.
  - ***Invarianten*** sind Bedingungen, die für alle Objekte zu allen Zeiten gelten
    - Die Menge der Invarianten bildet eine Äquivalenzrelation
- Da ein Objekt aus der Schablone der Klasse erzeugt wird, sind anfänglich die Werte seiner Attribute die des Klassenprototyps.
- Durch Ausführung von Methoden ändert sich jedoch der Zustand, d.h., die Attributwerte.



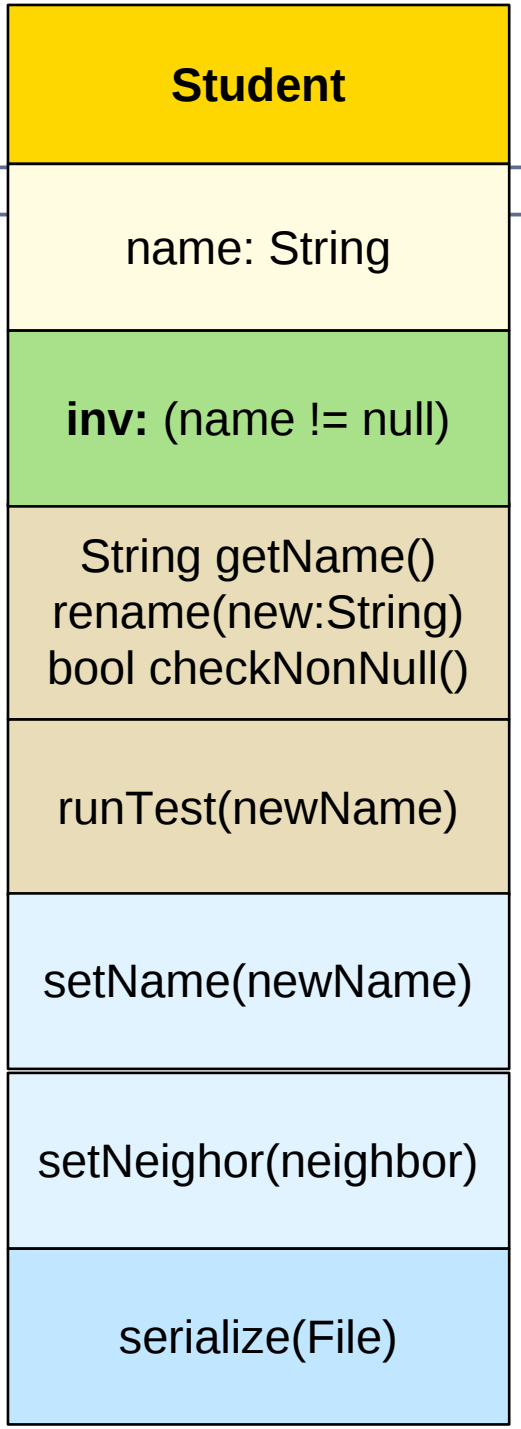
Strukturorientierte Sicht:
Arten von Methoden

Zustandsinvariante Methoden:

- **Anfragen (*queries*, Testbefehle).** Diese Prozeduren geben über den Zustand des Objektes Auskunft, verändern den Zustand aber nicht.
- **Prüfer (*checker*, Prüfbefehle)** prüfen eine Konsistenzbedingung (Integritätsbedingung, integrity constraint) auf dem Objekt, verändern den Zustand aber nicht.
  - Diese Prozeduren liefern einen booleschen Wert.
  - Prüfer von Invarianten, Vorbedingungen für Methoden, Nachbedingungen
- **Tester (Zustandstester)** rufen einen Prüfer auf und vergleichen sein Ergebnis mit einem Sollwert. Tester prüfen z.B. Invarianten

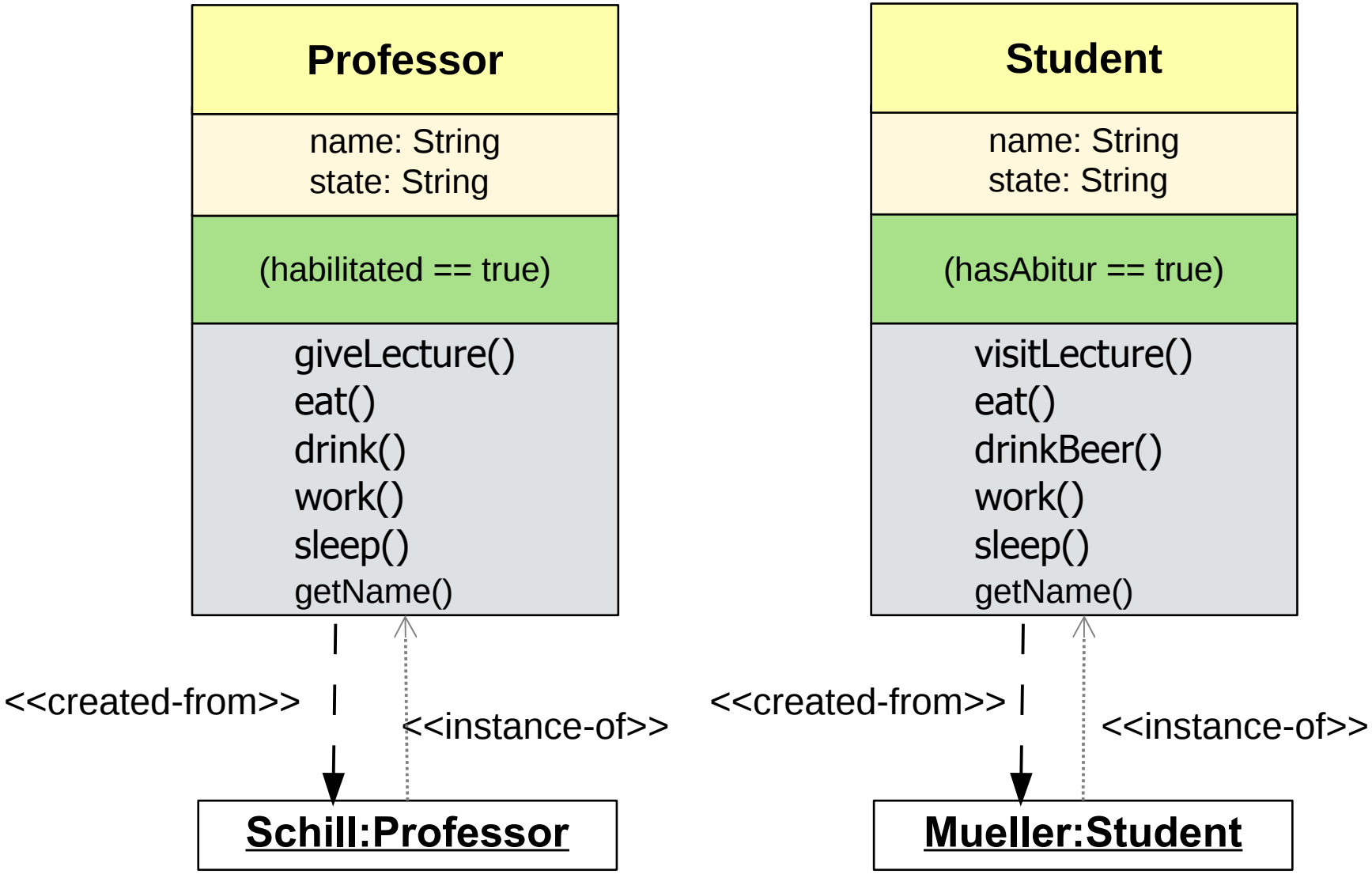
Zustandsverändernde Methoden:

- **Modifikatoren (Mutatoren)**
  - **Attributmodifikatoren** ändern den (lokalen) Zustand des Objekts
  - **Netzmodifikatoren** ändern die Vernetzung des Objektes
- **Repräsentationswechsler (representation changers)** transportieren das Objekt in eine andere Repräsentation
  - z.B. drucken den Zustand des Objekts in einen String, auf eine Datei oder auf einen Datenstrom.
- **Allgemeine Methoden**



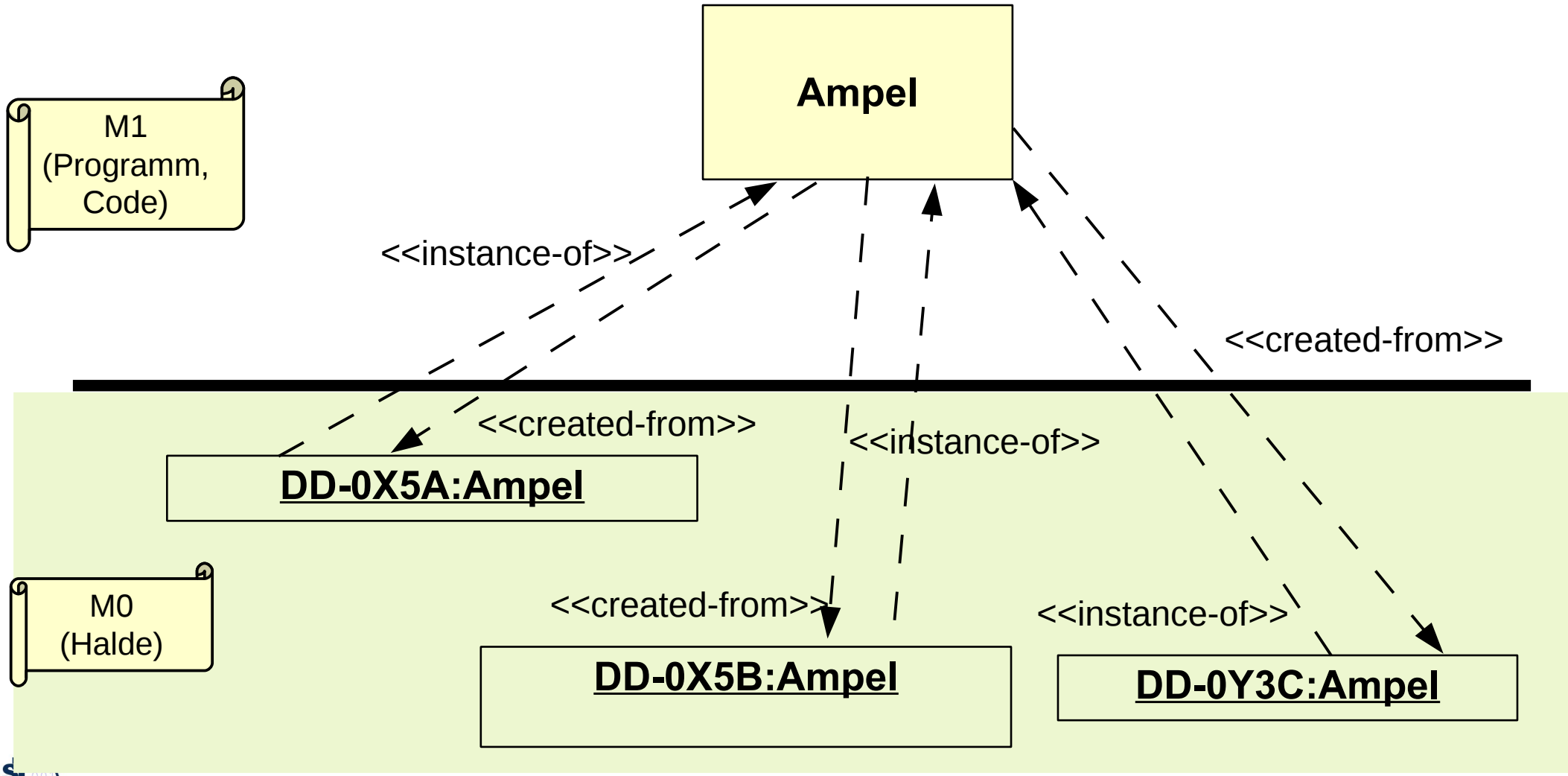
Klasse in strukturorientierter Sicht
Professoren und Studenten

- Ausprägungen werden durch Klassen-Objekt-Assoziationen ausgedrückt
- *created-from* ist die inverse Relation zu *instance-of*



Klasse in strukturorientierter Sicht

- Objekte befinden sich auf Modellierungsschicht 0 (M0)
- Klassen befinden sich auf Modellierungsschicht 1 (M1): Klassen sind Äquivalenzklassen und Schablonen für Objekte



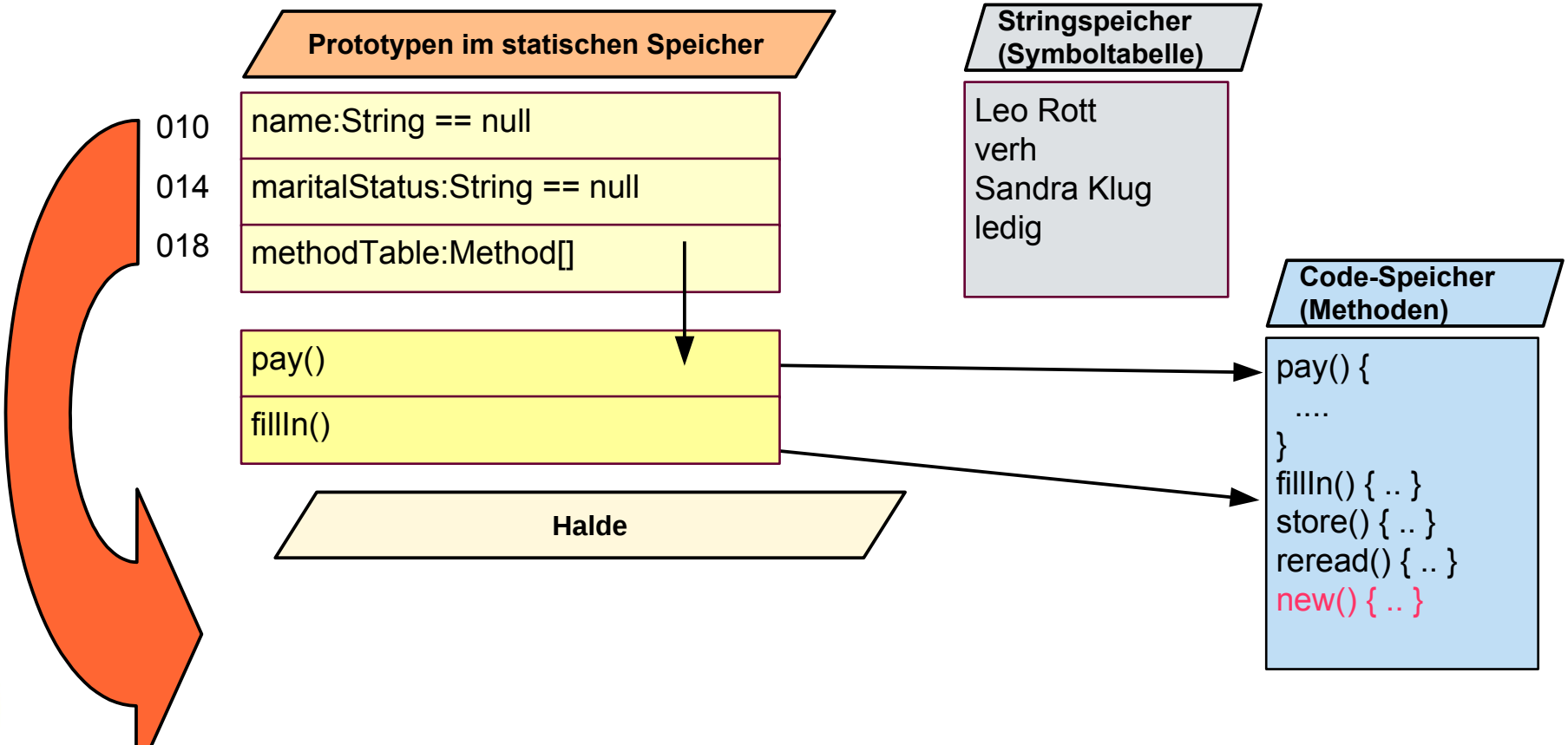
Strukturorientierte Sicht
Wie sieht eine Klasse im Speicher aus?

- Um die unterschiedlichen Sichtweisen auf Klassen zusammen behandeln zu können, gehen wir im Folgenden von folgenden Annahmen aus:
- Jede Klasse hat ein **Prototyp-Objekt** (eine Schablone)
  - Der Prototyp wird vor Ausführung des Programms angelegt (im statischen Speicher) und besitzt
    - eine Tabelle mit Methodenadressen im Codespeicher (die *Methodentabelle)*. Der Aufruf einer Methode erfolgt immer über die Methodentabelle des Prototyps
    - Eine Menge von statische Attributwerten (Klassenattribute)
- und einen **Objekt-Extent (Objektmenge),** eine Menge von Objekten (vereinfacht als Liste realisiert), die alle Objekte der Klasse erreichbar macht
  - In einer Datenbank entspricht der Objekt-Extent der Relation, die Klasse dem Schema

Wie erzeugt man ein Objekt für eine Klasse? (Verfeinerung)

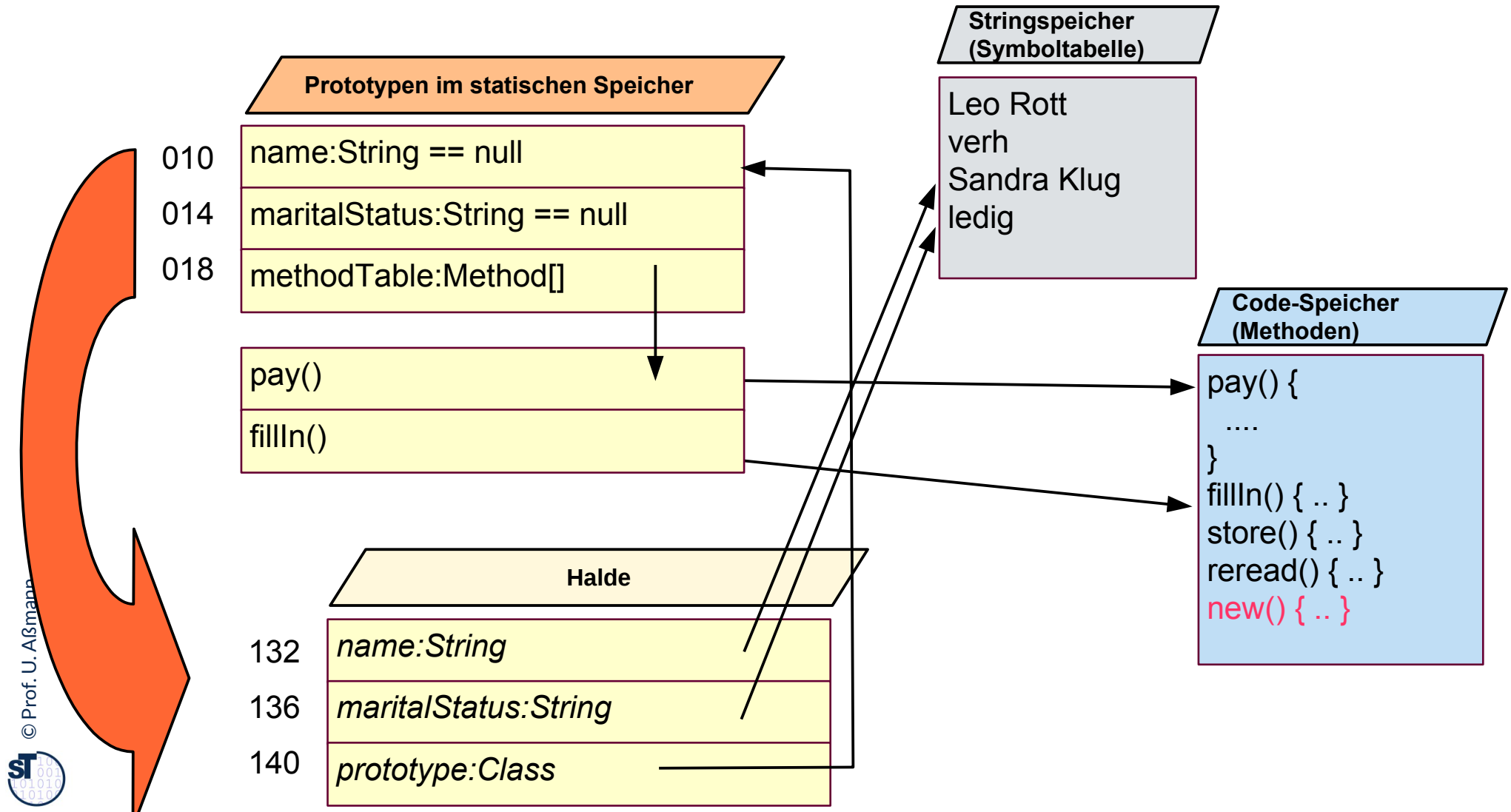

```
zahler2 = new Steuerzahler("Sandra Klug", "ledig");
```

- Man kopiert den Prototyp in die Halde:
  - Man reserviert den Platz des Prototyps in der Halde
  - kopiert den *Zeiger* auf die Methodentabelle (Vorteil: Methodentabelle kann von allen Objekten einer Klasse benutzt werden)
  - und füllt die Prototypattributwerte in den neuen Platz



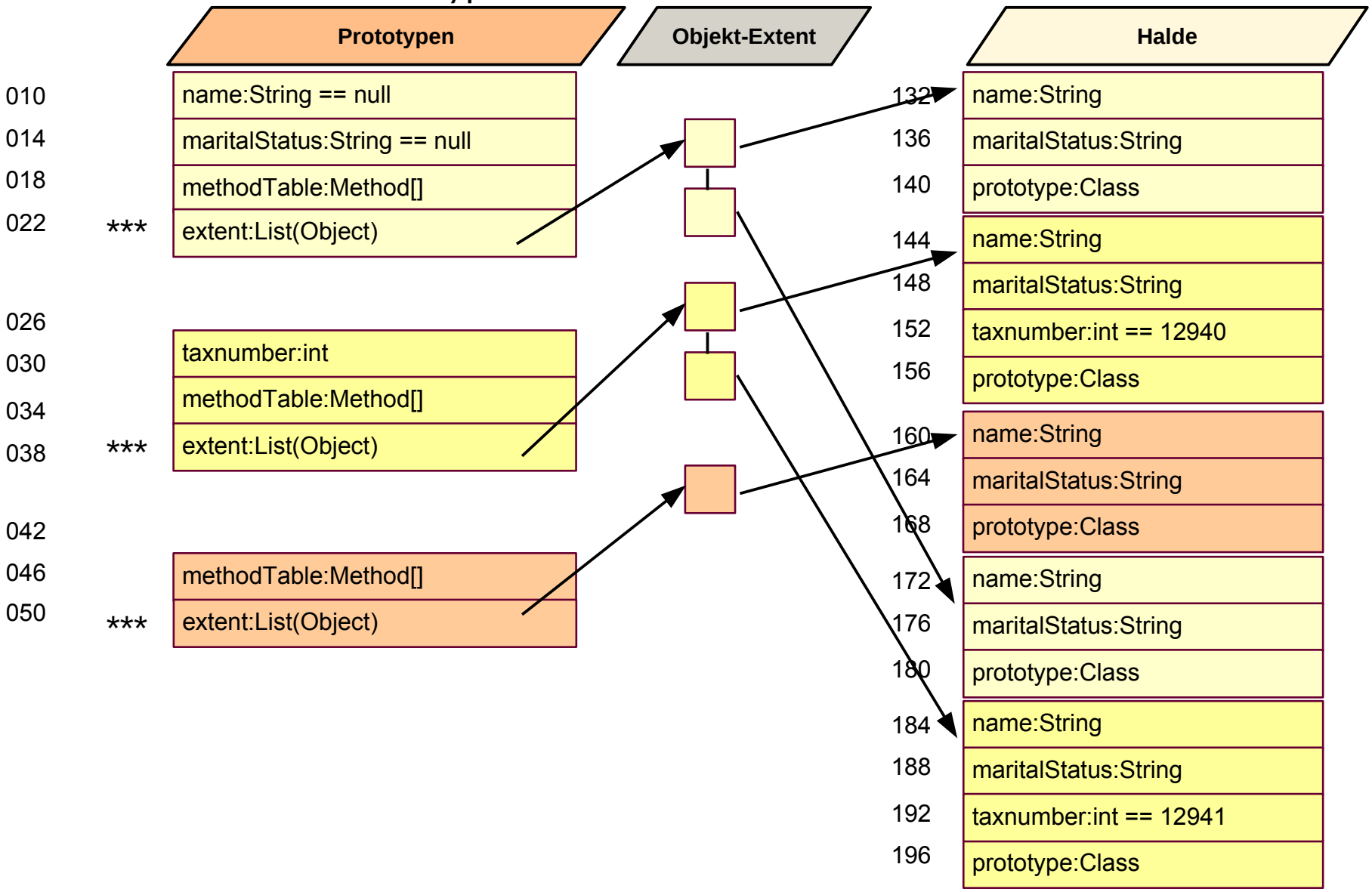
Wie erzeugt man ein Objekt für eine Klasse? (Verfeinerung)

- Ergebnis:



Objekt-Extent im Speicher

- Der **Objekt-Extent** ist eine Liste der Objekte einer Klasse. Wir benötigen dazu ein neues Attribut des Prototyps



Heim-Übung: Steuererklärungen

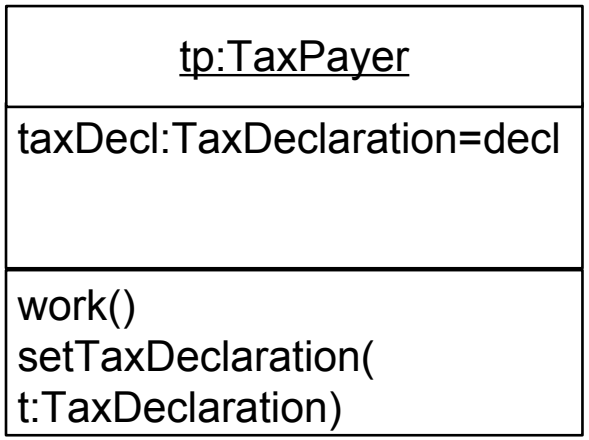
- Lade Datei **`TaxDeclarationDemo.java`** von der Webseite
- Lese und analysiere die Datei:
  - Welche Klassen sind English benamt und entsprechen den in der Vorlesung präsentierten Klassen?
  - Zeichnen sie ein Klassendiagramm mit Beziehungen zwischen den Klassen
  - Zeichnen Sie ein Snapshot der Anwendung bei Punkt (1) der `main`-Prozedur
  - Welche Attribute sind “doppelt”, d.h. in verschiedenen Klassen mit gleicher Semantik definiert?
  - Warum sind manche Attribute “privat”, d.h. nur über Zugriffsfunktionen (“setter/getter”, accessors) zugreifbar?
  - Wie setzt man den Zustand eines Steuerzahlers von “ledig” auf “verheiratet”?

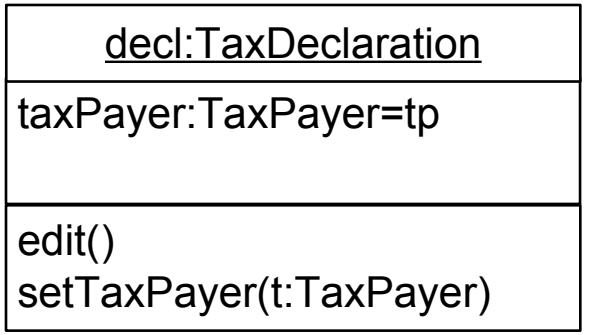
10.2.1. Objektnetze

Objektnetze

- Objekte existieren selten alleine; sie werden zu ***Objektnetzen*** verflochten (Objekte und ihre Relationen)
  - Ein Link von einem Objekt zum nächsten heisst ***Referenz*** *(Objekt-Assoziation)*
  - Die Beziehungen der Objekte in der Domäne müssen abgebildet werden

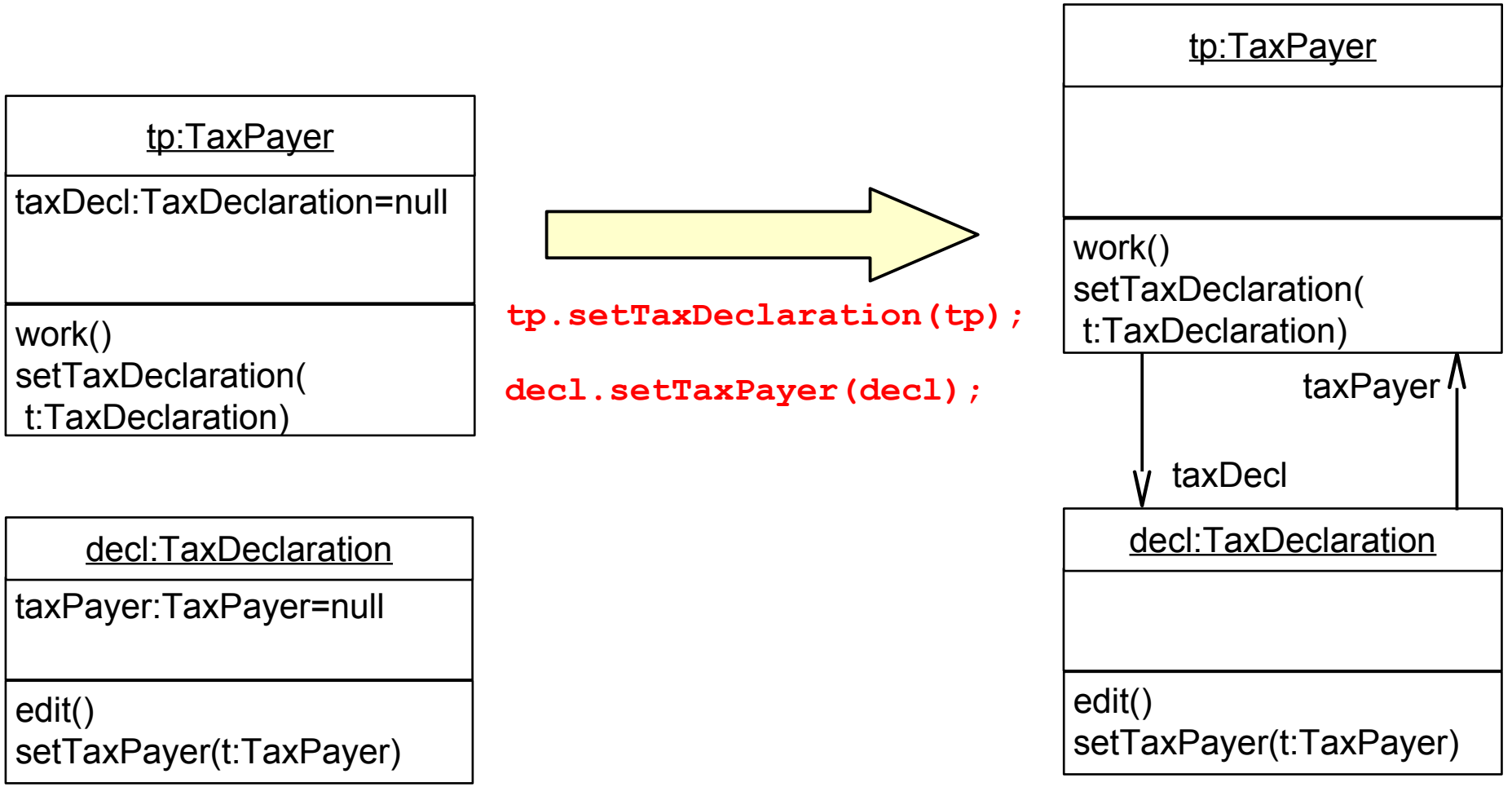
```
class TaxDeclaration {
  TaxPayer taxPayer;
  void setTaxPayer(TaxPayer t) { taxDecl = t; }
}
class TaxPayer {
  TaxDeclaration taxDecl;
  void setTaxDeclaration(TaxDeclaration t) {
     taxDecl = t; }
  void work() {..}
}
…
TaxPayer tp = new TaxPayer();
    // tp0.taxDecl == undefined
TaxDeclaration decl = new TaxDeclaration();
   // decl.taxPayer == undefined
tp.setTaxDeclaration(decl); // tp.taxDecl == decl
decl.setTaxPayer(tp);       // decl.taxPayer == tp
tp.work();
```





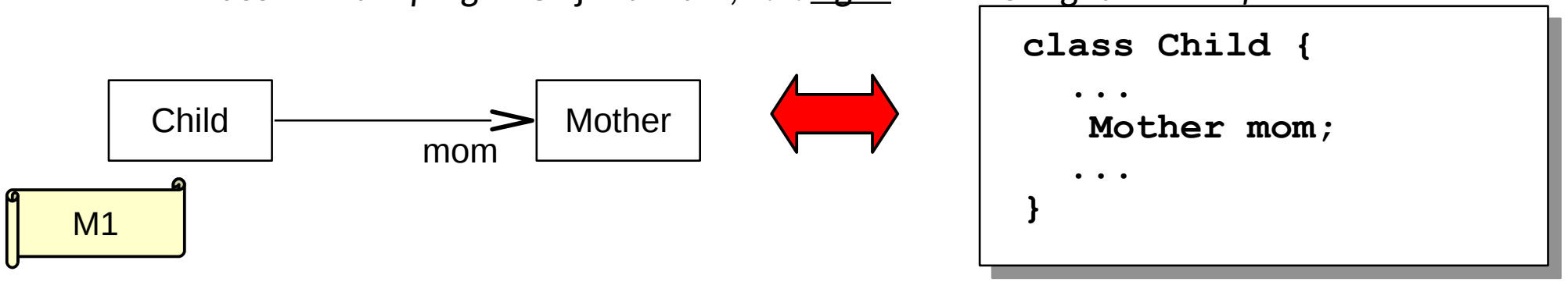
Objektwertige Attribute in Objektnetzen

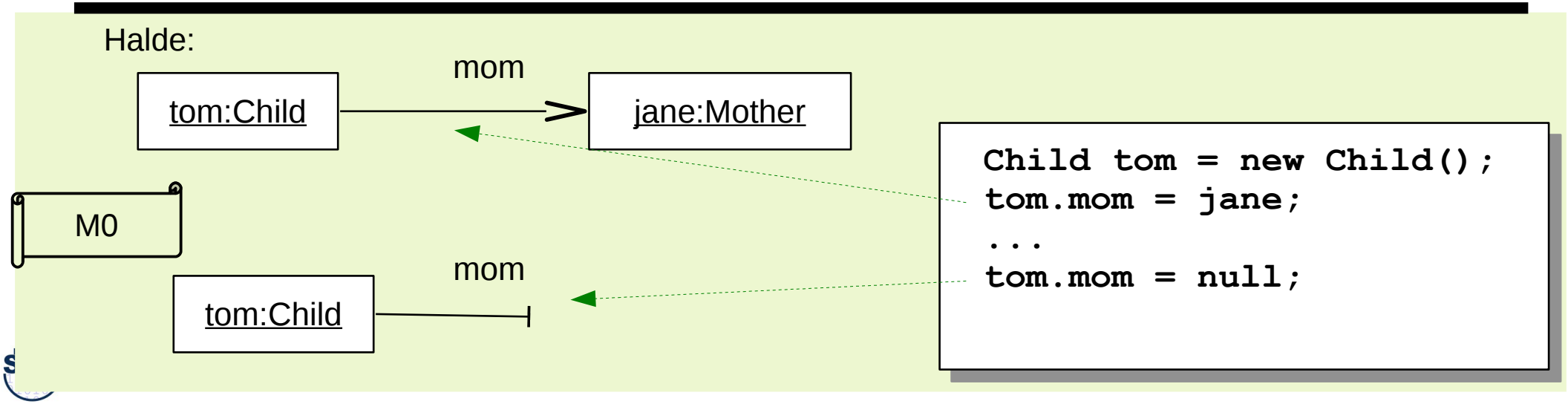
- ... können als Objektnetze in UML dargestellt werden
- In Java werden also Referenzen durch Attribute mit dem Typ eines Objekts (also kein Basistyp wie int); in UML durch Pfeile



Klassennetze und Objektnetze

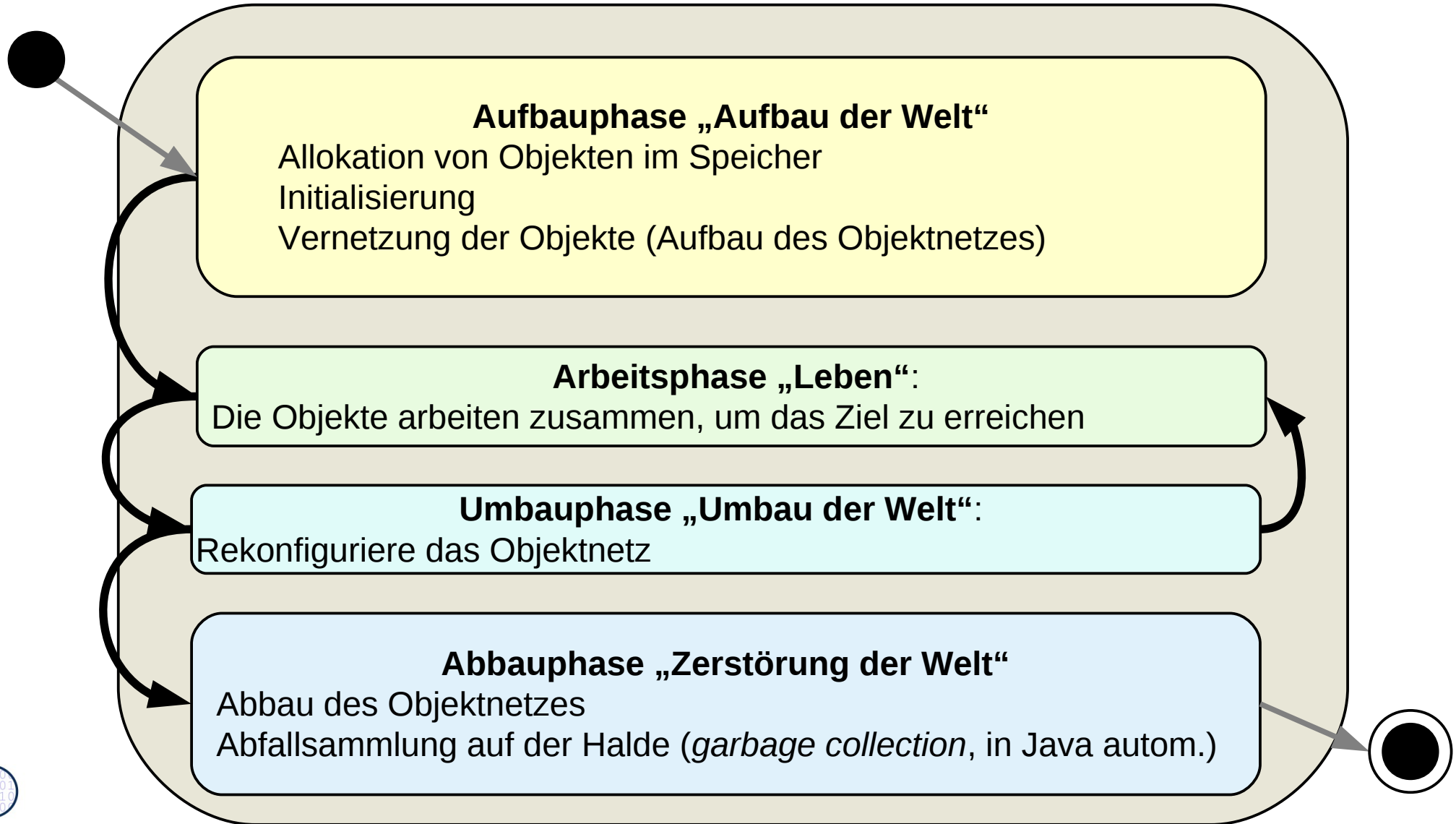
- Klassendiagramme (M1) sind "Zwangsjacken" für Objektdiagramme (M0)
- Objekte und Klassen können zu **Objektnetzen** verbunden werden:
  - Klassen durch *Assoziationen* (gerichtete Pfeile)
  - Objekte durch *Links* (*Zeiger*, *pointer*, gerichtete Pfeile)
- Klassennetze *prägen* Objektnetze, d.h. *legen* ihnen *Obligationen auf*





Phasen eines objektorientierten Programms

- Die folgende Phasengliederung ist als Anhaltspunkt zu verstehen; oft werden einzelne Phasen weggelassen oder in der Reihenfolge verändert



Phasen eines objektorientierten Programms


```
class TaxDeclaration {  TaxPayer taxPayer;  static int number = 0;
  void setTaxPayer(TaxPayer t) { taxDecl = t; };
  void setNumber() {  number++; };
  void edit() { .. };
}
class TaxPayer {   TaxDeclaration taxDecl;
  void setTaxDeclaration(TaxDeclaration t) { taxDecl = t; }
  void work() {..}
}
TaxPayer tp = new TaxPayer();    // tp0.taxDecl == undefined
TaxDeclaration decl = new TaxDeclaration(); // decl.taxPayer == undefined
```

Allokation

```
tp.setName(„John Silver");
decl.setNumber();
```

Initialisierung

```
tp.setTaxDeclaration(decl); // tp.taxDecl == decl
decl.setTaxPayer(tp);       // decl.taxPayer == tp
```

Vernetzung

```
tp.work();
decl.edit();
```

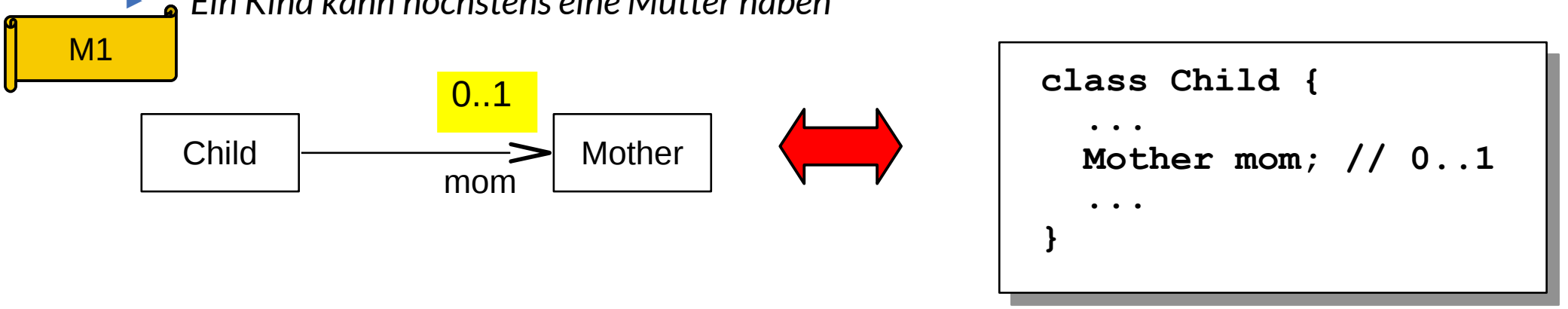
Arbeitsphase

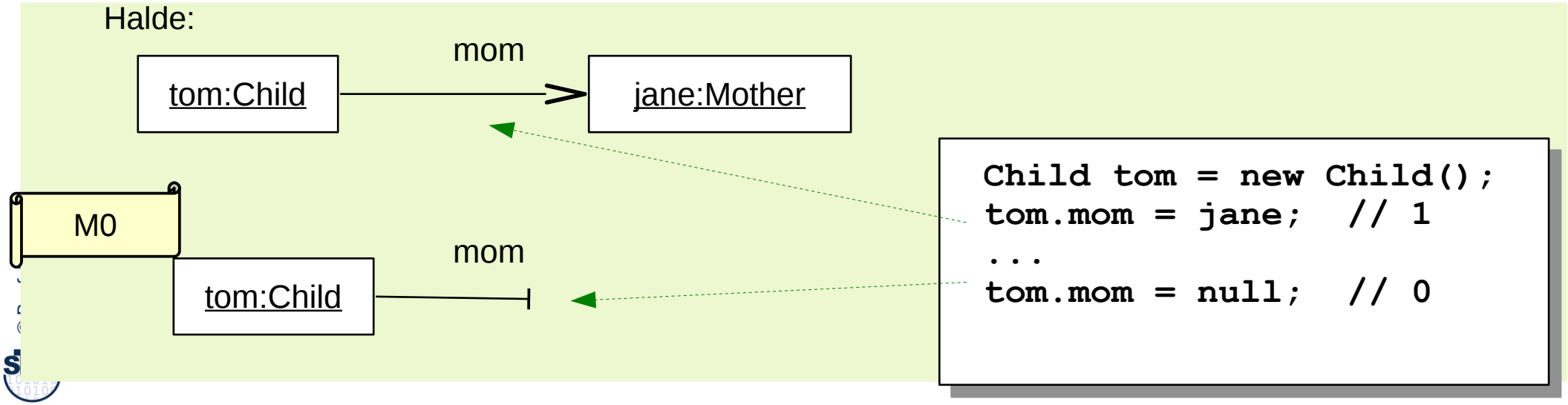
```
tp = null;
decl = null;
```

Abbauphase

Invarianten auf Links und Assoziationen

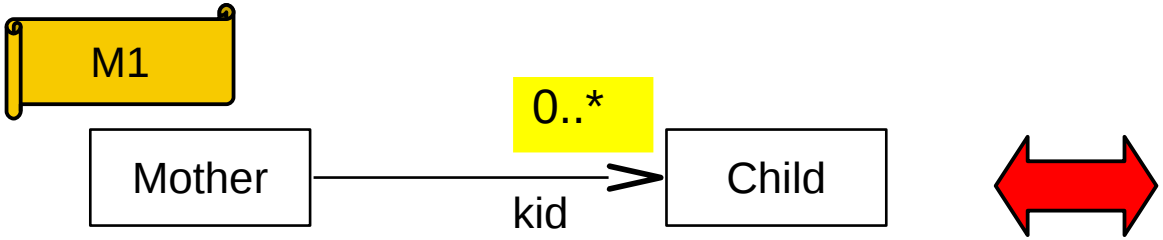
- Invarianten können in UML auch für Relationen spezifiziert werden
  - *Multiplizitäten* für die Anzahl der Partner (*Kardinalitätsbeschränkung* der Assoziation)
- *"Ein Kind kann höchstens eine Mutter haben"*



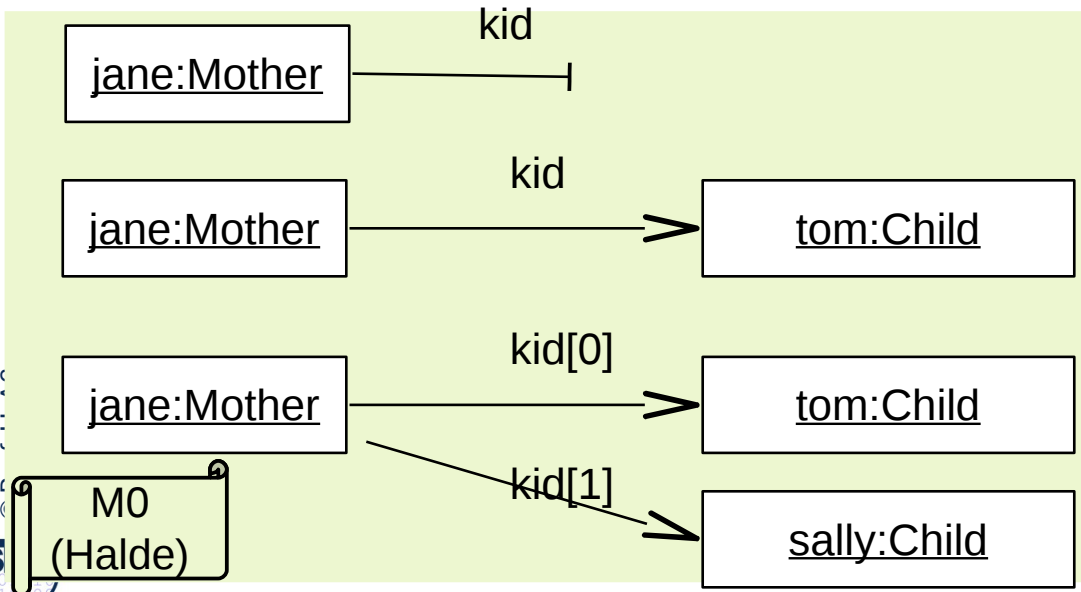


Mehrstellige Assoziationen

- Eine Mutter kann aber viele Kinder haben
  - Implementierung in Java durch integer-indizierbare Felder mit statisch bekannter Obergrenze (*arrays*). Allgemeinere Realisierungen im Kapitel “Collections”.
- Assoziationen und ihre Multiplizitäten prägen also die Gestalt der Objektnetze.




```
class Mother {
  ...
  Child[] kid; // 0..*
  ...
}
```



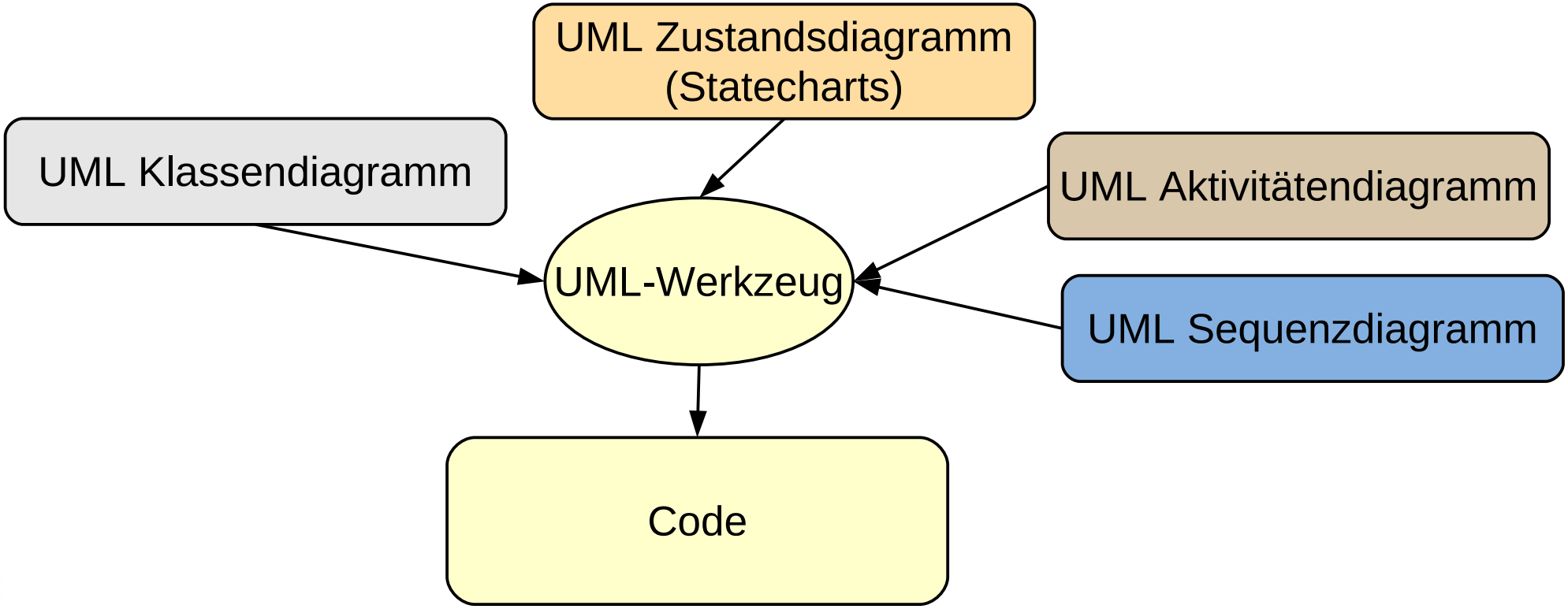

```
Mother jane = new Mother();
jane.kid[0] = tom;
...
jane.kid[1] = sally;
```

10.3 Objektorientierte Modellierung

- Programmierung spezifiziert ein System vollständig
- Modellierung spezifiziert fragmentarische Sichten auf ein System
  - Wird oft in der Anforderungsanalyse und im Entwurf gebraucht

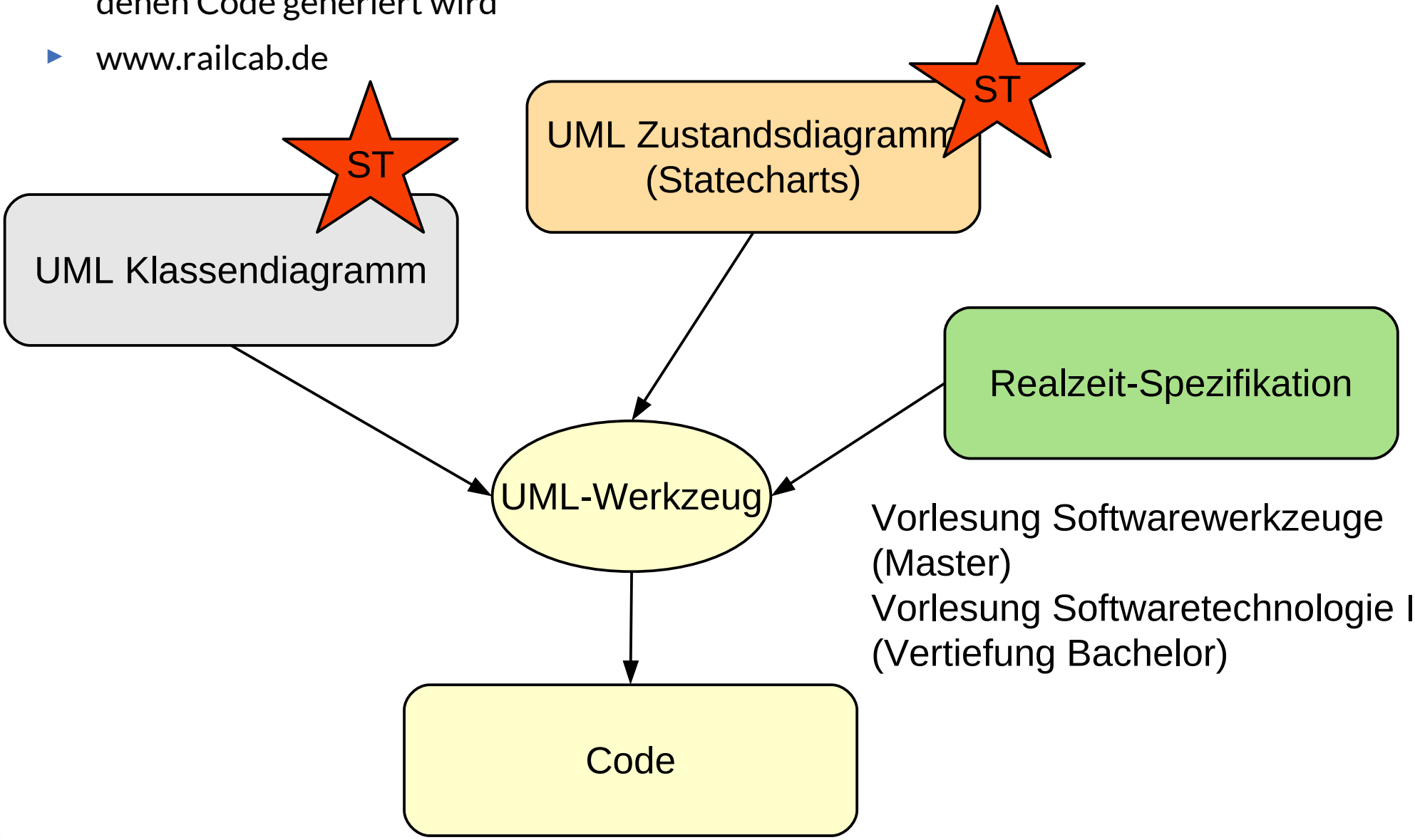
System-Modellierung mit UML

- UML besteht aus einer Sprachfamilie, deren Sprachen unterschiedliche Sichten auf das System modellieren
- **Modellierung** spezifiziert mehrere fragmentarische Sichten eines Systems, die an sich einfacher sind als das gesamte System
  - Vorteil: Sichten können oft verifiziert werden
  - Aus Modellen kann per Werkzeug Code generiert werden
- Beispielhaftes Szenario:



Bsp.: Modellierung von sicherheitskritischen Systemen

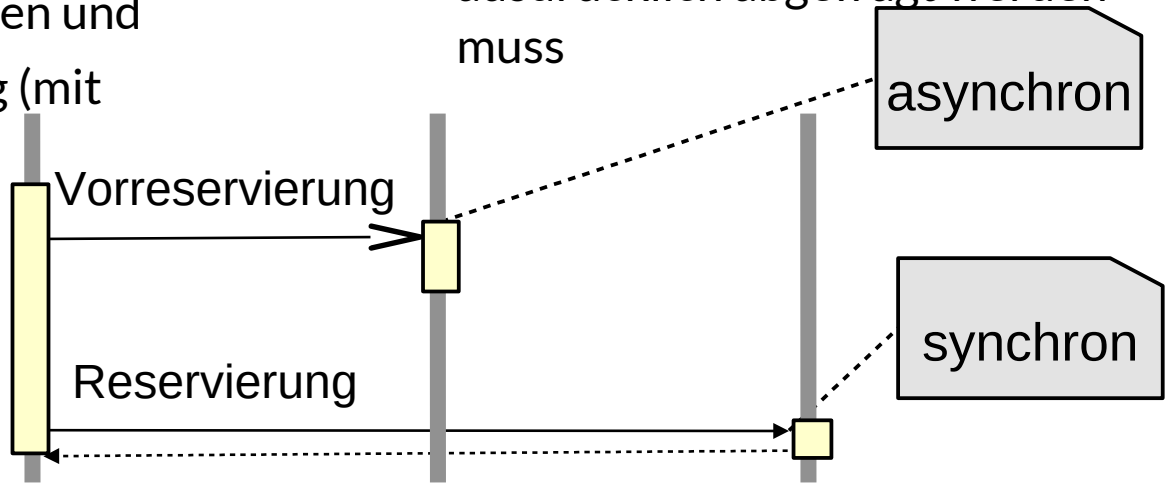
- Die Paderborner Railcabs werden modelliert mit Real-time statecharts, aus denen Code generiert wird
- www.railcab.de



Vorlesung Softwarewerkzeuge (Master)
Vorlesung Softwaretechnologie II (Vertiefung Bachelor)

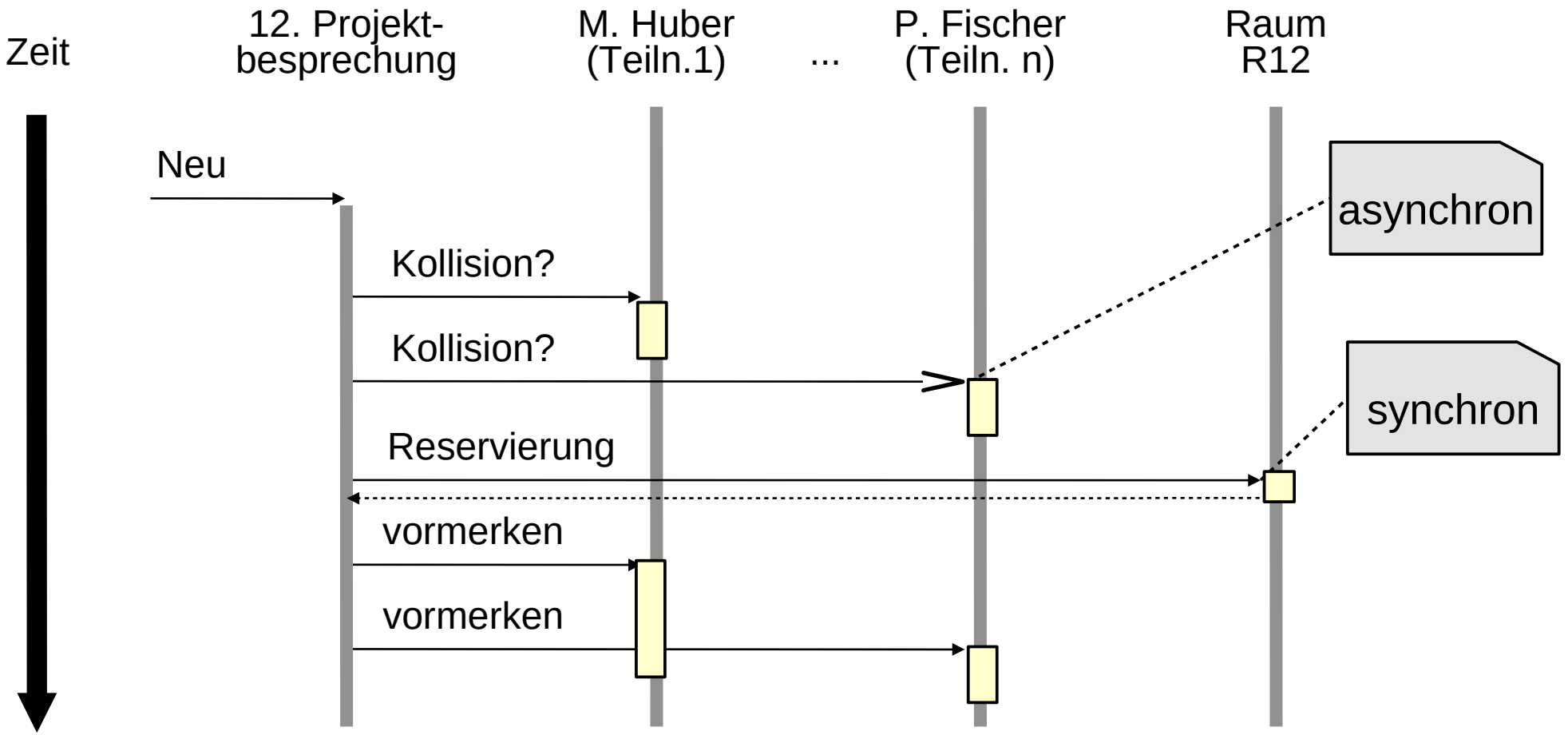
Sequentielle und parallele OO Sprachen

- Objekte kommunizieren durch Austausch von Botschaften und reagieren auf den Empfang einer Nachricht mit dem Ausführen einer (oder mehrerer) Methode(n)
- In einer **sequentiellen objekt-orientierten Sprache** setzt sich ein Aufruf an ein Objekt mit der Anfrage, eine Operation (einen Dienst) auszuführen, zusammen aus:
  - einer Aufruf-Nachricht (Botschaft, message),
  - einer **synchronen** Ausführung einer (oder mehrerer) Methoden und
  - einer Aufruf-Fertigmeldung (mit Rückgabe)
- In einer **parallelen objekt-orientierten Sprache** setzt sich ein Aufruf an ein Objekt mit der Anfrage, eine Operation (einen Dienst) auszuführen, zusammen aus:
  - einer Aufruf-Nachricht (Botschaft, message),
  - einer **asynchronen** Ausführung von Methoden (der Sender kann parallel weiterlaufen)
  - einer Aufruf-Fertigmeldung (mit Rückgabe), die vom Sender ausdrücklich abgefragt werden muss



UML-Sequenzdiagramm: Kooperative Ausführung in Szenarien paralleler Objekte

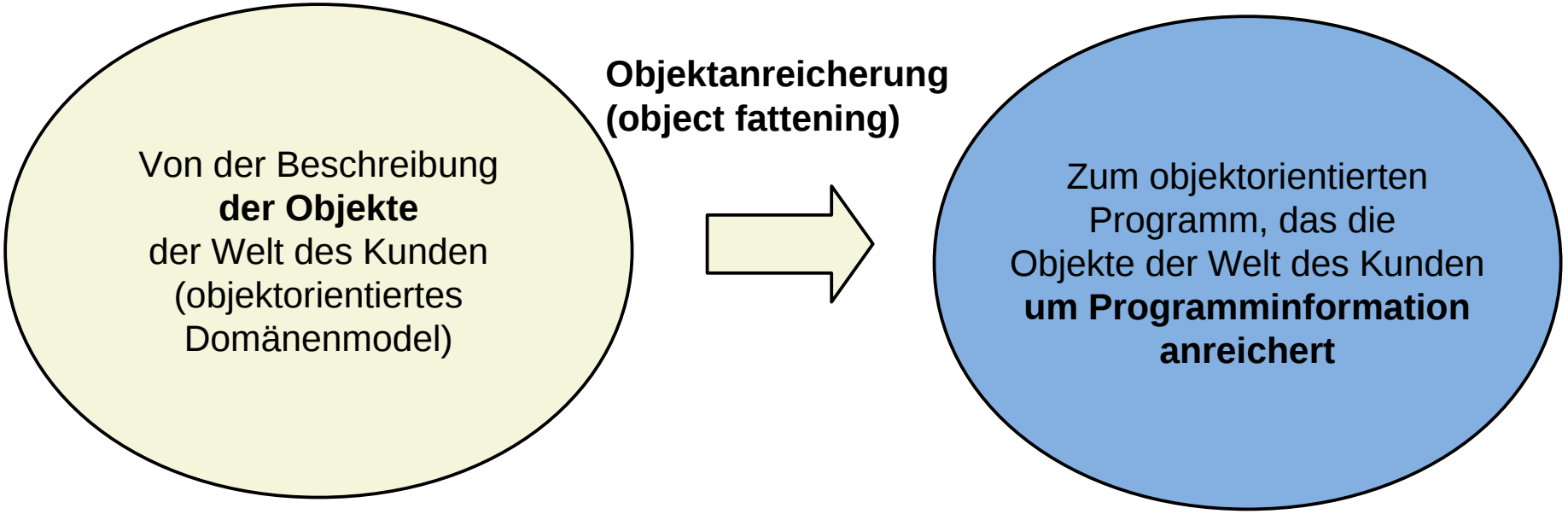
- In UML kann man sequentielle oder auch parallele Operationen spezifizieren
- Kooperierende Objekte mit lokaler Datenhaltung
- Nachrichten starten synchrone oder asynchrone Methoden



10.4 Objektorientierte Programmierung

Antwort der Objektorientierung: Durch “Objektanreicherung (object fattening)”

Wie können wir diese Beschreibung im Computer realisieren?

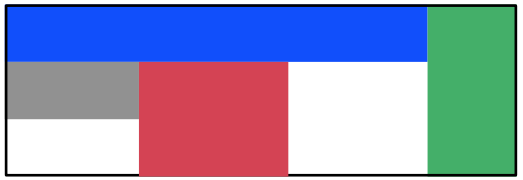


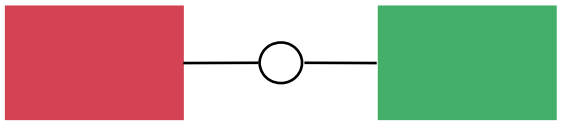
Objekt der Domäne
Verfettung: Anreicherung durch technische Programminformation

Die Welt *einfach* modellieren

Die Antwort der Objektorientierung:
durch Software-Objekte

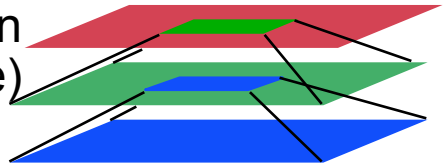
Identifikation von Objekten in Systemen
Kapselung von Funktionen und Zuständen in Objekten
(Verantwortungsbereiche)

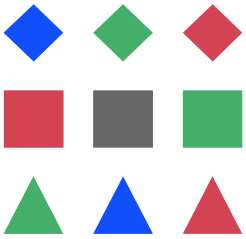




Klare Schnittstellen zwischen Teilsystemen
(Benutzungsprotokolle zwischen Objekten)

Verschiedene Abstraktionsebenen
(Hierarchie)





Vorgefertigte Teile - Wiederverwendung
(Baukastenprinzip, Rahmenwerke)

Programmierparadigmen

- ***Funktionale*** Programmierung:
  - Funktionsdefinition und -komposition, Rekursion
  - Werte, keine Zustände, keine Objekte mit Identität

```
fac(n) = if n = 0 then 1 else n * fac(n-1) end
```

- ***Imperative*** Programmierung:
  - Variablen, Attribute (Zustandsbegriff)
  - Steuerfluß: Anweisungsfolgen, Schleifen, Prozeduren

```
k := n; r := 1; while k > 0 do r := r*k; k := k-1 end;
```

- ***Logische*** Programmierung:
  - Werte, keine Zustände, keine Objekte mit Identität
  - Formallogische Axiomensysteme (Hornklauseln)
  - Fakten, Prädikate, Invarianten
  - Deduktionsregeln (Resolution)

```
fac(0,1).
fac(n+1, r) :- fac(n, r1),
  mult(n+1, r1, r).
```

At the end of the day, you have to put your data and operations somewhere.
Martin Odersky

Strukturparadigmen

- ***Unstrukturierte*** Programmierung:
  - Lineare Folge von Befehlen, Sprüngen, Marken (*spaghetti code*)

```
k:=n; r:=1; M: if k<=0 goto E; r:=r*k; k:=k-1; goto M; E:...
```

- ***Strukturierte*** Programmierung:
  - Blöcke, Prozeduren, Fallunterscheidung, Schleifen
  - Ein Eingang, ein Ausgang für jeden Block

```
k := n; r := 1; while k > 0 do r := r*k; k := k-1 end;
```

- ***Modulare*** Programmierung:
  - Sammlung von Typdeklarationen und Prozeduren
  - Klare Schnittstellen zwischen Modulen

```
DEFINITION MODULE F; PROCEDURE fac(n:CARDINAL):CARDINAL; ...
```

- ***Logische*** Programmierung:
  - Darstellung von Daten als Fakten, Mengen, Listen, Bäume (Terme)
  - Wissen als Regeln
  - Deduktion zur Berechnung neuen Wissens

```
public rule if (n == 1) then m = n.
```

Strukturparadigmen (ctd.)

- ***Objektorientierte*** Programmierung:
  - Kapselung von Daten und Operationen in Objekte
  - Klassen, Vererbung und Polymorphie

```
public class CombMath extends Number { int fac(int n) …
```

- ***Rollenorientierte*** Programmierung:
  - Kapselung von kontextbezogenen Eigenschaften von Objekten in *Rollen*
  - Klassen, Vererbung und Polymorphie auch auf Rollen

```
public context ProducerConsumer {
  public role class Producer { void put(int n); }
  public role class Consumer { void get(int n); }
} ...
```

Verschiedene Antworten auf “Wie komme ich zum Programm?”

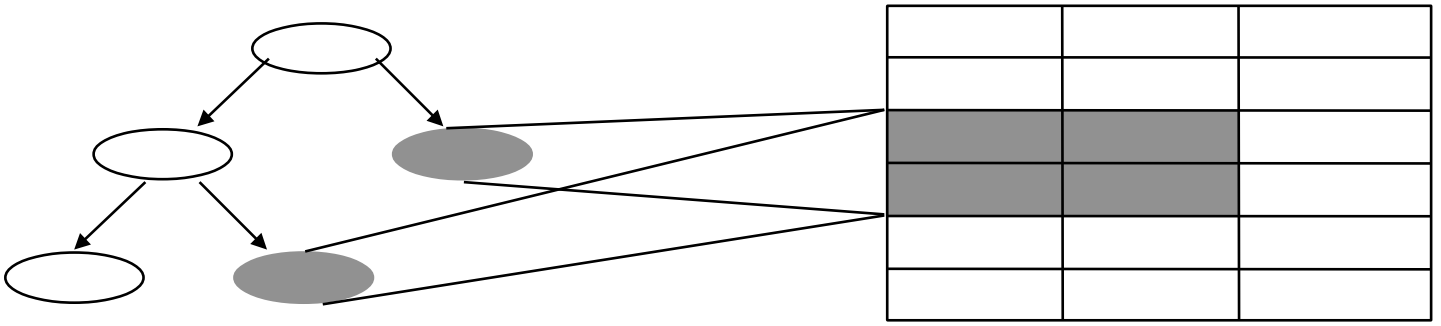
- **Funktionale Sprachen (wie Haskell, ML)**
  - Eingabe wird durch Funktionen in Ausgaben umgesetzt; f.Pr. berechnen Werte *ohne* Zustand. Keine Referenzen auf Werte möglich
- **Imperative modulare Sprachen (wie Modula, C, Ada 83)**
  - Funktionen berechnen Werte *mit Zustand*.
  - Auf Ereignisse kann reagiert werden (reaktive Systeme)
  - Statische Aufrufbeziehungen zwischen den Funktionen
- **Logische Sprachen (Prolog, Datalog, F-Logik, OWL)**
  - Prädikate beschreiben Aussagen (Wahrheiten)
  - Deklarative Logik beschreibt inferierbares Wissen auf einer Faktenbasis
  - Kein expliziter Steuer- oder Datenfluß, ist implizit
- **Objektorientierte Sprachen (wie C++, Java, C#, Ada95)**
  - Funktionen berechnen Werte auf Objekten mit einem Zustand
  - **Domänen-Objekte bilden das Gerüst des Programms**
  - Dynamisch veränderliche Aufrufbeziehungen
- **Modellierungssprachen (wie UML oder SysML)**
  - Graphische Modelle, aus denen Code generiert werden kann (plattformunabhängig)

Orthogonalität

	Strukturiert	Modular	Objekt-orientiert	Rollen-orientiert
Funktional	– –	**ML**	**Haskell**	– –
Imperativ	**Pascal**	**Modula-2 Ada**	**Java C++**	**ObjectTeams**
Logisch Mengenorientiert	– –	**XSB- Prolog**	**OWL fUML**	– –

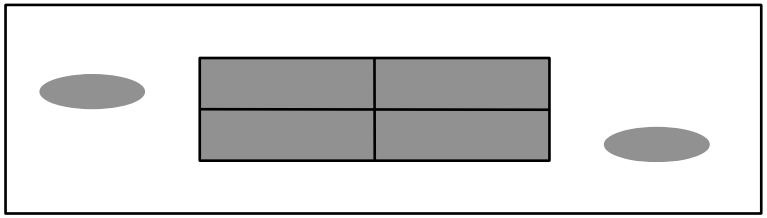
Funktion und Daten

- ***Separation*** von Funktion und Daten in der modularen Programmierung



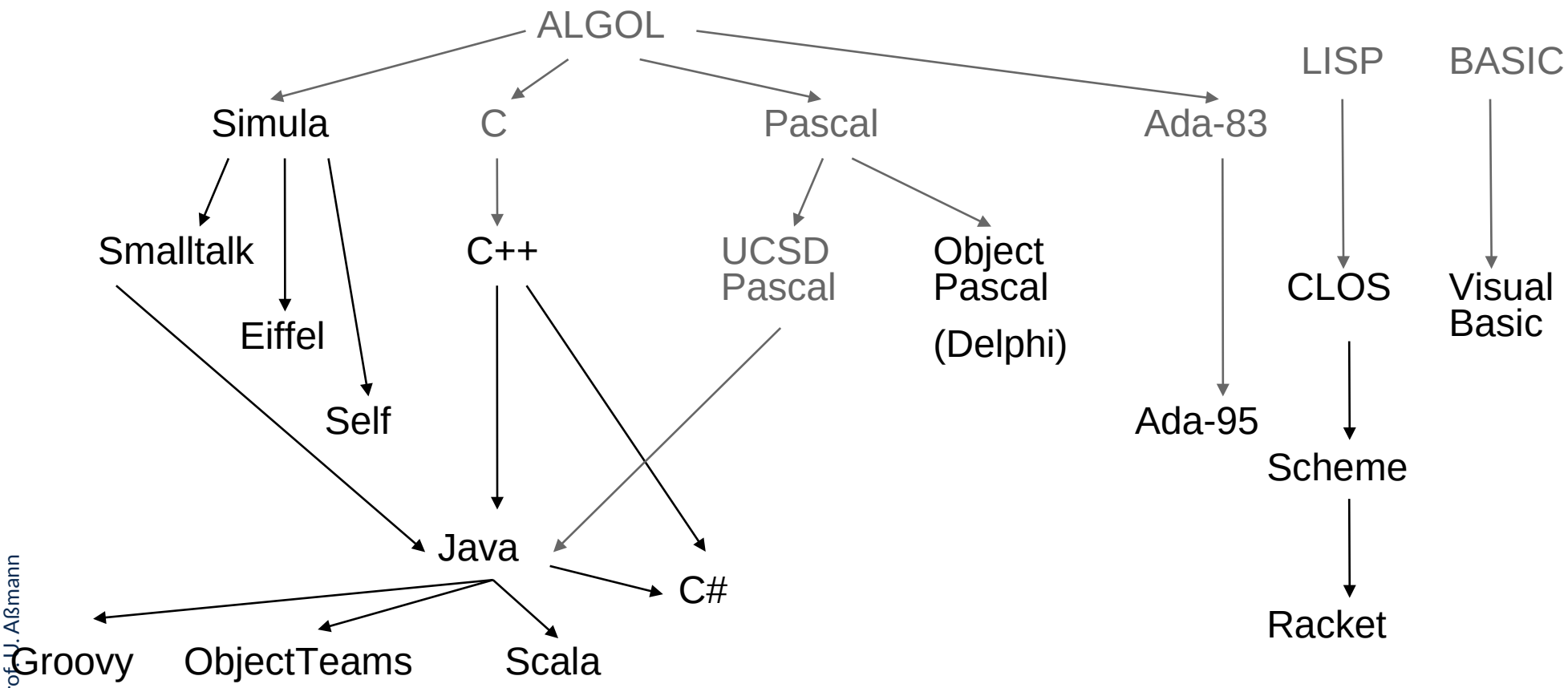
Funktionshierarchie Datenbasis

- ***Integration*** von Funktion und Daten in der objektorientierten Programmierung



- In den Strukturparadigmen:
  - strukturiert: Separation von Funktion und Daten
  - modular: Modul = (grosse) Funktionsgruppe + lokale Daten
  - objektorientiert: Objekt = (kleine) Dateneinheit + lokale Funktionen

Objektorientierte Programmiersprachen



Geschichte der OO-Programmierung

- ***Simula***: Ole-Johan Dahl + Krysten Nygaard, Norwegen, 1967
- Allan Kay: The Reactive Engine, Dissertation, Univ. Utah, 1969
- ***Smalltalk***: Allan Kay, Adele Goldberg, H. H. Ingalls,
  Xerox Palo Alto Research Center (PARC), 1976-1980
- ***C++***: Bjarne Stroustrup, Bell Labs (New Jersey), 1984
- ***Eiffel***: Bertrand Meyer, 1988
- ***Java***: Ken Arnold, James Gosling, Sun Microsystems, 1995
- ***C#***: Anders Heijlsberg, Microsoft (auch Schöpfer von Turbo Pascal)
  http://www.csharp.net
- ***Scala***: Martin Odersky, http://www.scala-lang.org
- ***Racket***: Dialekt von Lisp und Scheme

Die Programmiersprache Java

- **Java™** - Geschichte:
  - Vorläufer von Java: OAK (First Person Inc.), 1991-1992
    - » Betriebssystem/Sprache für Geräte, u.a. interaktives Fernsehen
  - 1995: ***HotJava*** Internet Browser
    - » Java *Applets* für das World Wide Web
    - » 1996: Netscape Browser (2.0) Java-enabled
  - 2005: Java 10.5 mit Generizität
  - Weiterentwicklungen:
    - » Java als Applikationsentwicklungssprache (Enterprise Java)
    - » Java zur Gerätesteuerung (Embedded Java)
    - » Java Beans, Enterprise Java Beans (Software-Komponenten)
    - » Java Smartcards
  - 2014: Java 8
    - » neue Fensterbibliothek JavaFX
    - » Skeletons für paralleles Programmieren

Warum gerade Java?

- Java ist relativ **einfach** und **konzeptionell klar**
  - Java vermeidet “unsaubere” (gefährliche) Konzepte.
  - Strenges Typsystem
  - Kein Zugriff auf Speicheradressen (im Unterschied zu C)
  - Automatische Speicherbereinigung
- Java ist **platform-unabhängig** von Hardware und Betriebssystem.
  - Java *Bytecode* in der *Java Virtual Machine*
- Java ist angepaßt an moderne Benutzungsoberflächen, inkl. Web
  - *Java Swing Library*, *Java FX*, *Java Spring (Web)*
- Java bietet die **Wiederbenutzung** von großen Klassenbibliotheken (Frameworks, Rahmenwerken) an
  - z.B. Java *Collection Framework*
- Java ermöglicht parallele, nebenläufige und verteilte Programme (*parallel*, *multi-threading*, *distributed)*

Objektorientierte Klassenbibliotheken (Frameworks, Rahmenwerke)

- Klassenbibliotheken sind vorgefertigte Mengen von Klassen, die in eigenen Programmen (Anwendungen) benutzt werden können
  - Java Development Kit (JDK)
    - Collections, Swing, ...
  - Test-Klassenbibliothek Junit
  - Praktikumsklassenbibliothek SalesPoint
- Durch Vererbung kann eine Klasse aus einer Bibliothek angepasst werden
  - Eine Anwendung besteht nur zu einem kleinen Prozentsatz aus eigenem Code (Wiederverwendung bringt Kostenersparnis)
- Nachteil: Klassenbibliotheken sind komplexe Programmkomponenten.
  - Man muss eine gewisse Zeit investieren, um die Klassenbibliothek kennenzulernen
  - Man muss eine Menge von Klassenbibliotheken kennen

Probleme von Java

- Keine Verhaltensgleichheit von Klassen garantiert (keine konforme Vererbung):
  - Man kann bei der Wiederbenutzung von Bibliotheksklassen Fehler machen und den Bibliothekscode *invalidieren*
- Basisdatentypen (int, char, boolean, array) sind keine Objekte
  - Anders in C#!
- JVM startet langsam. Beim Start lädt sie zuerst alle Klassen (dynamic class loading), anstatt sie statisch zu einem ausführbaren Programm zu binden
  - Übung: Starte die JVM mit dem –verbose flag
- Grosse Bibliothek benötigt grossen Einarbeitungsaufwand

Warum ist das alles wichtig?

- Anfangsfrage: *Wie können wir die Welt möglichst einfach beschreiben?*
- Antwort: *durch Objekte*
  - ..und ihre Klassen, die sie abstrahieren
    - ... die Beziehungen der Klassen (Vererbung, Assoziation)
  - .. ihre Verantwortlichkeiten
- Daher bietet Objektorientierung eine Entwicklungsmethodik an, die die Welt des Kunden mit der technischen Welt des Programms *brückt*
  - Die Software ist *einfach* modelliert, d.h. analog zur Welt organisiert
  - Die Software ergibt sich als “Erweiterung” der Welt
  - Daher wird die Entwicklung erleichtert
- Objektorientierte Modellierung führt Sichten ein, die die Spezifikation eines Systems erleichtern

Appendix

10.A.1 Merkmale von Klassen

Operationen

- **Def.:** Eine ***Operation (Instanzoperation)*** einer Klasse *K* ist die Beschreibung einer Aufgabe, die jedes Objekt der Klasse *K* ausführen kann.
- ***Operation****:* "a service that can be requested from an object to effect behaviour" (UML-Standard)

aUML für Analyse:

Teambesprechung
titel beginn dauer raumFestlegen einladen absagen

Teambesprechung
titel beginn dauer
raumFestlegen() einladen() absagen()

jUML:

Teambesprechung
String titel Time beginn int dauer
bool raumFestlegen() void einladen() bool absagen()

- "Leere Klammern":
  - In vielen Büchern (und den Unterlagen zur Vorlesung) zur Unterscheidung von Attributnamen: raumFestlegen(), einladen(), absagen() etc.
  - Klammern können aber auch weggelassen werden

Operation, Methode, Botschaften

In objektorientierten Sprachen gibt es neben Operationen weitere Konzepte, die Verhalten beschreiben

- ***Message (Botschaft, Nachricht):*** eine Nachricht an ein Objekt, um eine Operation auszuführen oder ein externes Ereignis mitzuteilen
- ***Methode:*** “the implementation of an operation (the “*how*” of an operation)”
  - "In den Methoden wird all das programmiert, was geschehen soll, wenn das Objekt die betreffende Botschaft erhält." [Middendorf/Singer]
  - ***Prozedur****:* gibt keinen Wert zurück, verändert aber Zustand
  - ***Funktion****:* gibt einen Wert oder ein Objekt zurück
  - ***synchrone Methode:*** der Sender wartet auf die Beendigung des Service
  - ***asynchrone Methode:*** ein Service mit Verhalten aber ohne Rückgabe, d.h. der Sender braucht nicht zu warten
- ***Kanal (channel):*** Ein Objekt hat einen Ein- und einen Ausgabekanal (input, output channel), über den die Botschaften gesendet werden.
  - Das Objekt lauscht also an seinem Eingabekanal auf Nachrichten und führt sie synchron oder asynchron aus.
  - Manche objektorientierten Sprachen erlauben mehrere Kanäle.

Spezifikation von Operationen

- **Definition:** Die *Spezifikation* einer Operation legt das Verhalten der Operation fest („was“), ohne einen Algorithmus („wie“) festzuschreiben.

> Eine Spezif kation beschreibt das ***"Was"*** eines Systems,
> aber noch nicht das ***"Wie"***.

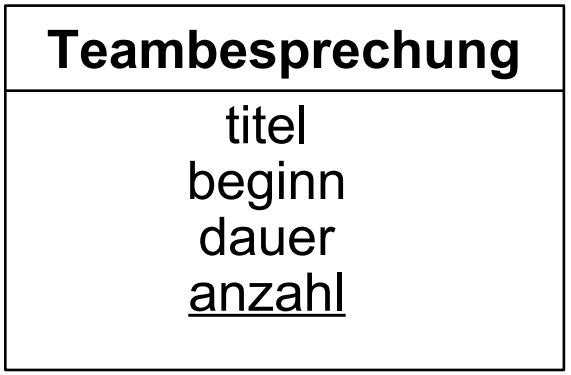
- Häufigste Formen von Spezifikationen:
  - **informell** (in der Analyse, aUML)
    - **Text** in natürlicher Sprache (oft mit speziellen Konventionen), oft in Programmcode eingebettet (Kommentare)
      - Werkzeugunterstützung zur Dokumentationsgenerierung, z.B. "javadoc"
    - Pseudocode (programmiersprachenartiger Text)
    - Tabellen, spezielle Notationen
  - **formal** (im Entwurf und Implementierung, dUML, jUML)
    - **Signaturen** (Typen der Parameter und Rückgabewerte)
    - **Vor- und Nachbedingungen** (Verträge, contracts)
    - **Protokolle** mit **Zustandsmaschinen**, Aktivitätendiagrammen

Klassenattribut (Statisches Attribut)

- Ein ***Klassenattribut*** *A* beschreibt ein Datenelement, das genau einen Wert für die gesamte Klasse annehmen kann.
  - Es ist also ein Attribut des Klassenprototypen, i.G. zu einem Attribut eines Objekts
- **Notation:** Unterstreichung

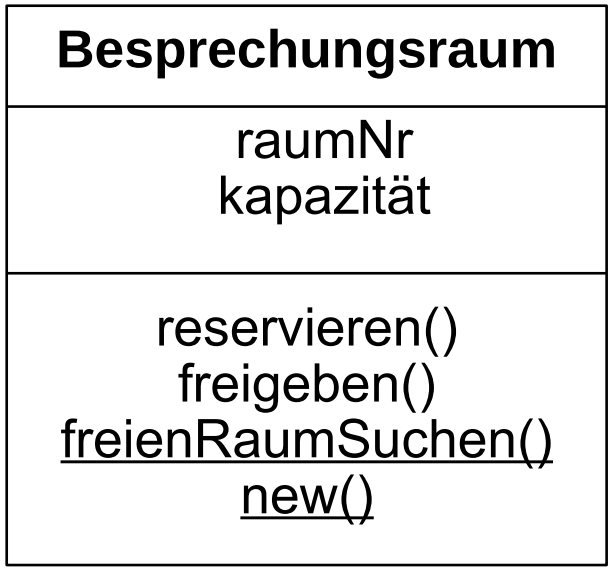
jUML:

aUML für Analyse:



Klassenoperation (Statische Operation)

- **Definition** Eine ***Klassenoperation*** *A* einer Klasse *K* ist die Beschreibung einer Aufgabe, die nur unter Kenntnis der aktuellen Gesamtheit der Instanzen der Klasse ausgeführt werden kann.
  - (Gewöhnliche Operationen heißen auch ***Instanzoperationen.***)
  - Klassenoperationen bearbeiten i.d.R. *nur* Klassenattribute, *keine Objektattribute.*
- **Notation UML:** Unterstreichung analog zu Klassenattributen
- **Java:** Die Methode `main()` ist statisch, und kann vom Betriebssystem aus aufgerufen werden
  - Klassenattribute und -operationen: Schlüsselwort **`static`**
- Jede Klasse hat eine Klassenoperation `new()`, den Allokator




```
class Steuererklaerung {

  public static main (String[] args) {
     Steuerzahler hans =
       new Steuerzahler();
     ...
  }
}
```

Parameter und Datentypen für Operationen

- Detaillierungsgrad in der Analysephase gering
  - meist Operationsname ausreichend
  - Signatur kann angegeben werden
  - Entwurfsphase und Implementierungsmodell: vollständige Angaben sind nötig.
- **jUML Notation:**

  `Operation(Art Parameter:ParamTyp=DefWert, ...): ResTyp`
  - *Art* (des Parameters): **in**, **out**, oder **inout** (weglassen heißt **in**)
  - *DefWert* legt einen Default-Parameterwert fest, der bei Weglassen des Parameters im Aufruf gilt.
- **Beispiel** (Klasse Teambesprechung)**:**

  raumFestlegen (in wunschRaum: Besprechungsraum): Boolean

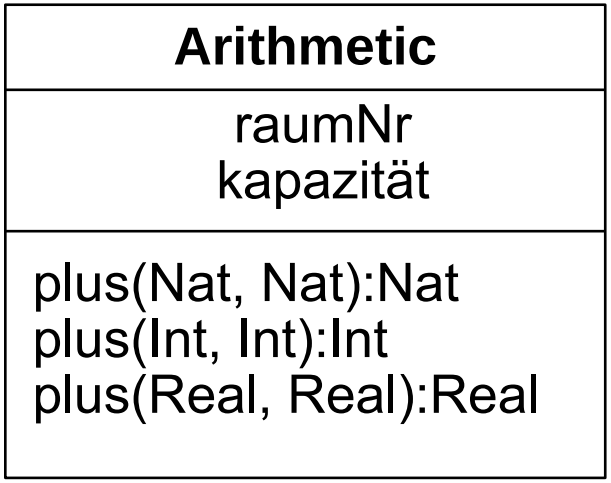
Überladung von Operationen

- Zwei Methoden heissen *überladen*, wenn sie gleich heissen, sich aber in ihrer Signatur (Zahl oder Typisierung der Parameter) unterscheiden
  - Auswahl aus mehreren gleichnamigen Operationen nach Anzahl und Art der Parameter.
- Klassisches Beispiel: Arithmetik

```
+: (Nat,Nat)Nat, (Int,Int)Int, (Real,Real)Real
```

- Java-Operationen:

```
int f1 (int x, y) {...}
int f1 (int x) {...}
int x = f1(f1(1,2));
```



Überladen vs. Polymorphismus

- Überladung wird vom Java-Übersetzer statisch aufgelöst. Aus

```
int f1 (int x, y) {...}
int f1 (int x) {...}
int x = f1(f1(1,2));
```

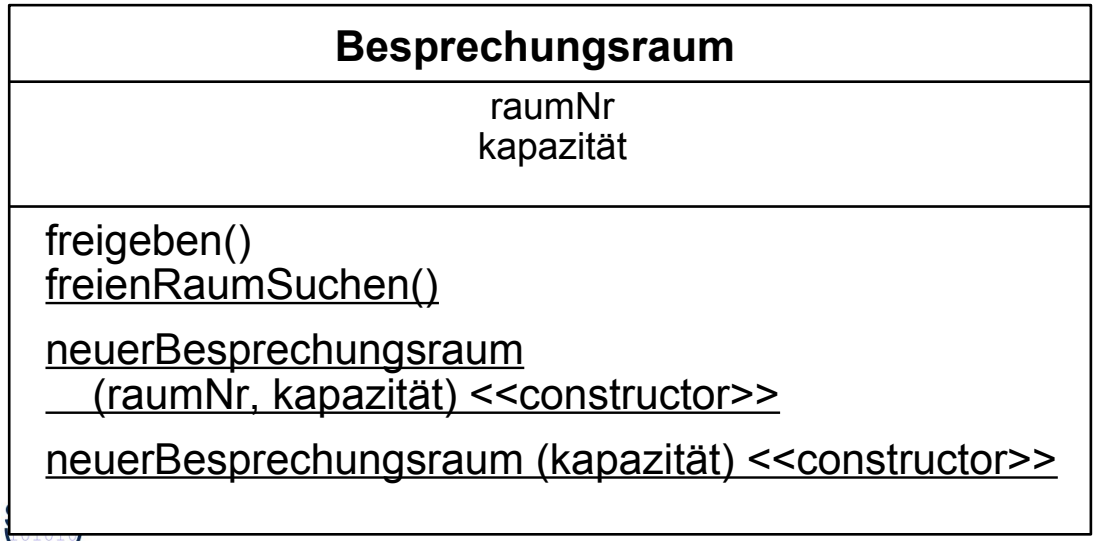
  - macht der Übersetzer

```
int f1_2_int_int (int x, y) {...}
int f1_1_int (int x) {...}
int x = f1_1_int( f1_2_int_int(1,2) );
```

  - indem er die Stelligkeit und die Typen der Parameter bestimmt und den Funktionsnamen in der .class-Datei expandiert
- Polymorphie dagegen kann nicht zur Übersetzungszeit aufgelöst werden
  - Der Merkmalssuchalgorithmus muss dynamisch laufen, da dem Übersetzer nicht klar ist, welchen Unterklassentyp ein Objekt besitzt (es können alle Typen unterhalb der angegebenen Klasse in Frage kommen)

Konstruktor-Operation

- **Definition:** Ein ***Konstruktor (-operation)*** *C* einer Klasse *K* ist eine Klassenoperation, die eine neue Instanz der Klasse erzeugt und initialisiert.
  - Ergebnistyp von *C* ist immer implizit die Klasse *K*.
- Explizite Konstruktoroperationen werden in UML mit einem *Stereotyp* "<<constructor>>" markiert.
- ***Default-Konstruktor:*** Eine Konstruktoroperationen ohne Parameter wird implizit für jede Klasse angenommen
- Konstruktoren sind i.d.R. überladen, d.h. können mit mehreren Parametersätzen definiert werden




```
class Teammitglied {
    ... // ohne Konstruktor
  }
```

```
Teammitglied m = new Teammitglied();
```

Mehr zu Konstruktoren in Java

- Konstruktoren werden meist überladen:

```
class Teammitglied {

  private String name;
  private String abteilung;

  public Teammitglied (String n, String abt) {
    name = n;
    abteilung = abt;
  }

  public Teammitglied (String n) {
    name = n;
    abteilung = "";
  }                                                    ... }
```

```
Teammitglied m = new Teammitglied("Max Müller", "Abt. B");
Teammitglied m2 = new Teammitglied("Mandy Schmitt");
```

Löschen von Objekten

- In Java gibt es ***keine*** Operation zum Löschen von Objekten (keine "Destruktoren").
  - Andere objektorientierte Programmiersprachen kennen Destruktoren (z.B. C++)

- Speicherfreigabe in Java:
  - Sobald keine Referenz mehr auf ein Objekt besteht, *kann* es gelöscht werden.
  - Der konkrete Zeitpunkt der Löschung wird vom Java-Laufzeitsystem festgelegt ("garbage collection").
  - Aktionen, die vor dem Löschen ausgeführt werden müssen:

    `protected void finalize()` (über)definieren

- Fortgeschrittene Technik zur Objektverwaltung (z.B. für Caches):
  - "schwache" Referenzen (verhindern Freigabe nicht)
  - Paket `java.lang.ref`

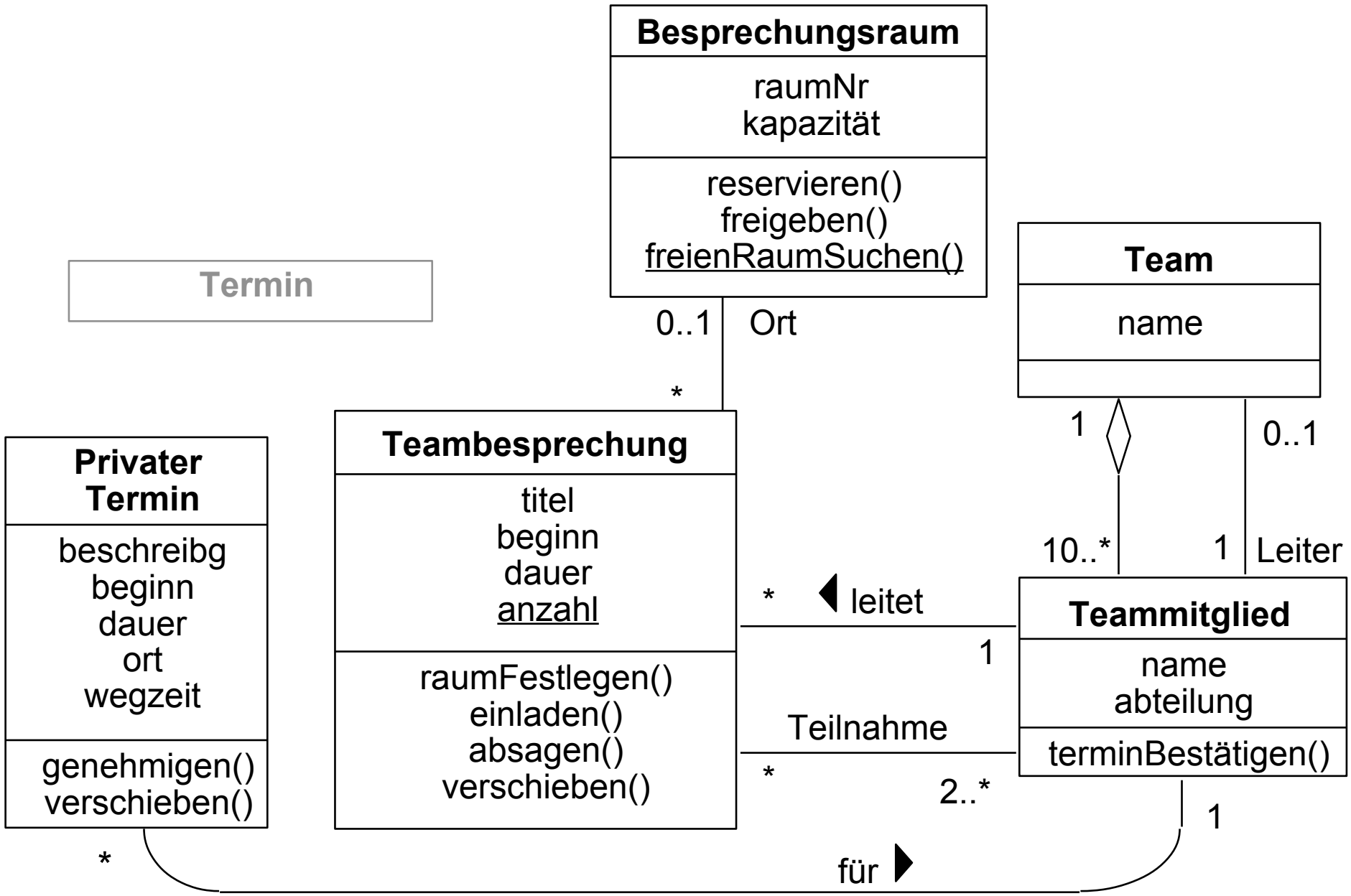
Beispiel Netzaufbau im Java-Programm

- Ein Programm auf höherer Ebene muss zunächst ein Objektnetz verdrahten, bevor die Objekte kommunizieren können (Aufbauphase). Dies kann das Hauptprogramm sein

```
class Terminv {
  public static void main (String argv[]) {
    // Aufbauphase
    Besprechungsraum r1 = new Besprechungsraum("R1", 10);
    Besprechungsraum r2 = new Besprechungsraum("R2", 20);
    Teammitglied mm = new Teammitglied("M. Mueller", "Abt. A");
    Teammitglied es = new Teammitglied("E. Schmidt", "Abt. B");
    Teammitglied fm = new Teammitglied("F. Maier", "Abt. B");
    Teammitglied hb = new Teammitglied("H. Bauer", "Abt. A");
    Hour t1s5 = new Hour(1,5); // Tag 1, Stunde 5
    Teammitglied[] tlB1 = {mm, es};
    Teambesprechung tb1 =
      new Teambesprechung("Bespr. 1", t1s5, 2, tlB1);

    // jetzt erst Arbeitsphase
    tb1.raumFestlegen();
  ...
  }
}
```

Beispiel: Operationen im Analysemodell (aUML)



Die zentralen Frage der Softwaretechnologie

Wie kommen wir vom Problem des Kunden zum Programm (oder Produkt)?

Wie können wir die Welt möglichst einfach beschreiben?

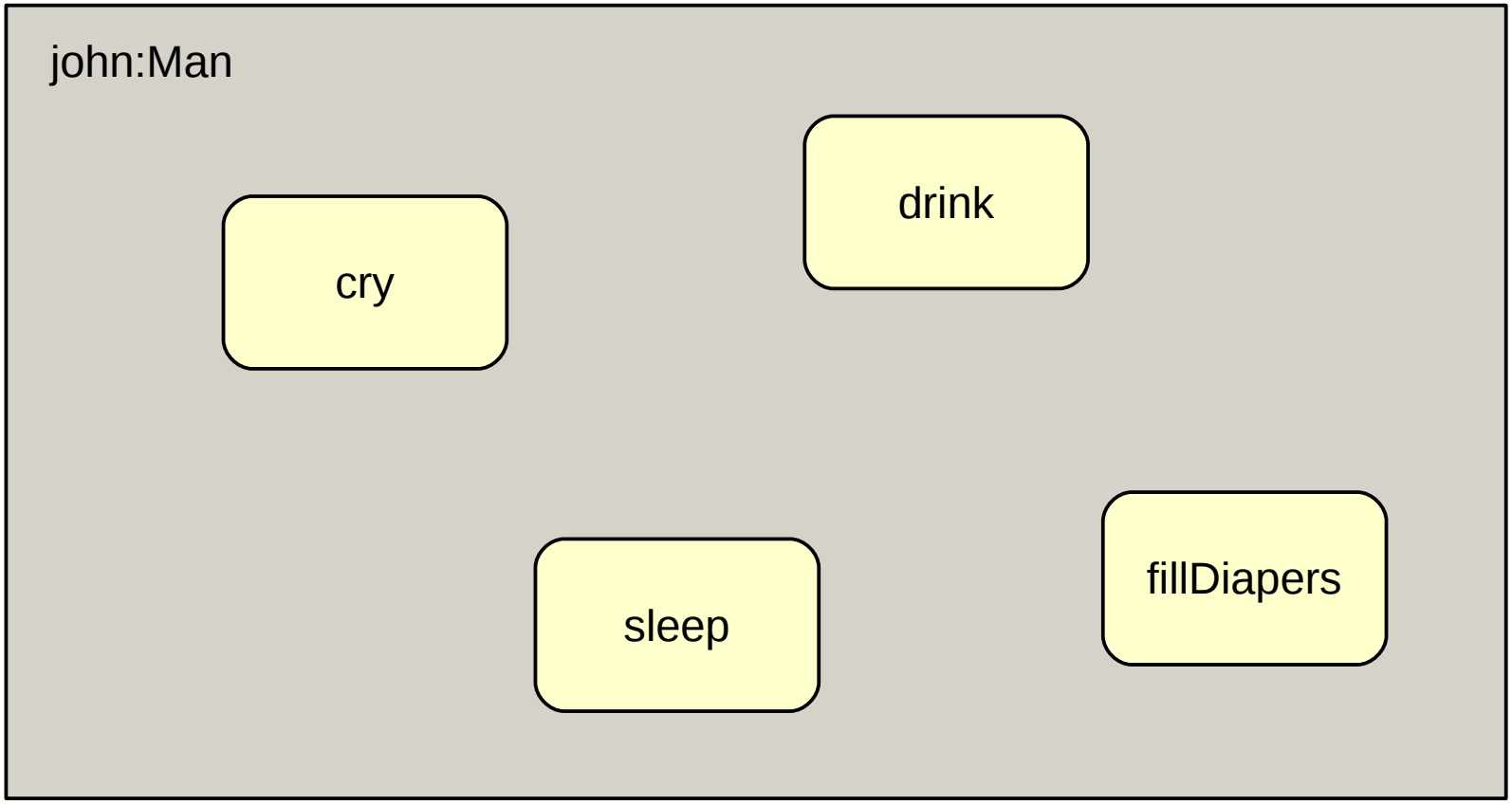
Wie können wir diese Beschreibung im Computer realisieren?

Beispiel: Allokation und Aufruf eines Objektes in Java

- Objekte durchlaufen im Laufe ihres Lebens viele Zustandsveränderungen, die durch Aufrufe verursacht werden
- Das objektorientierte Programm simuliert dabei den Lebenszyklus eines Domänenobjekts

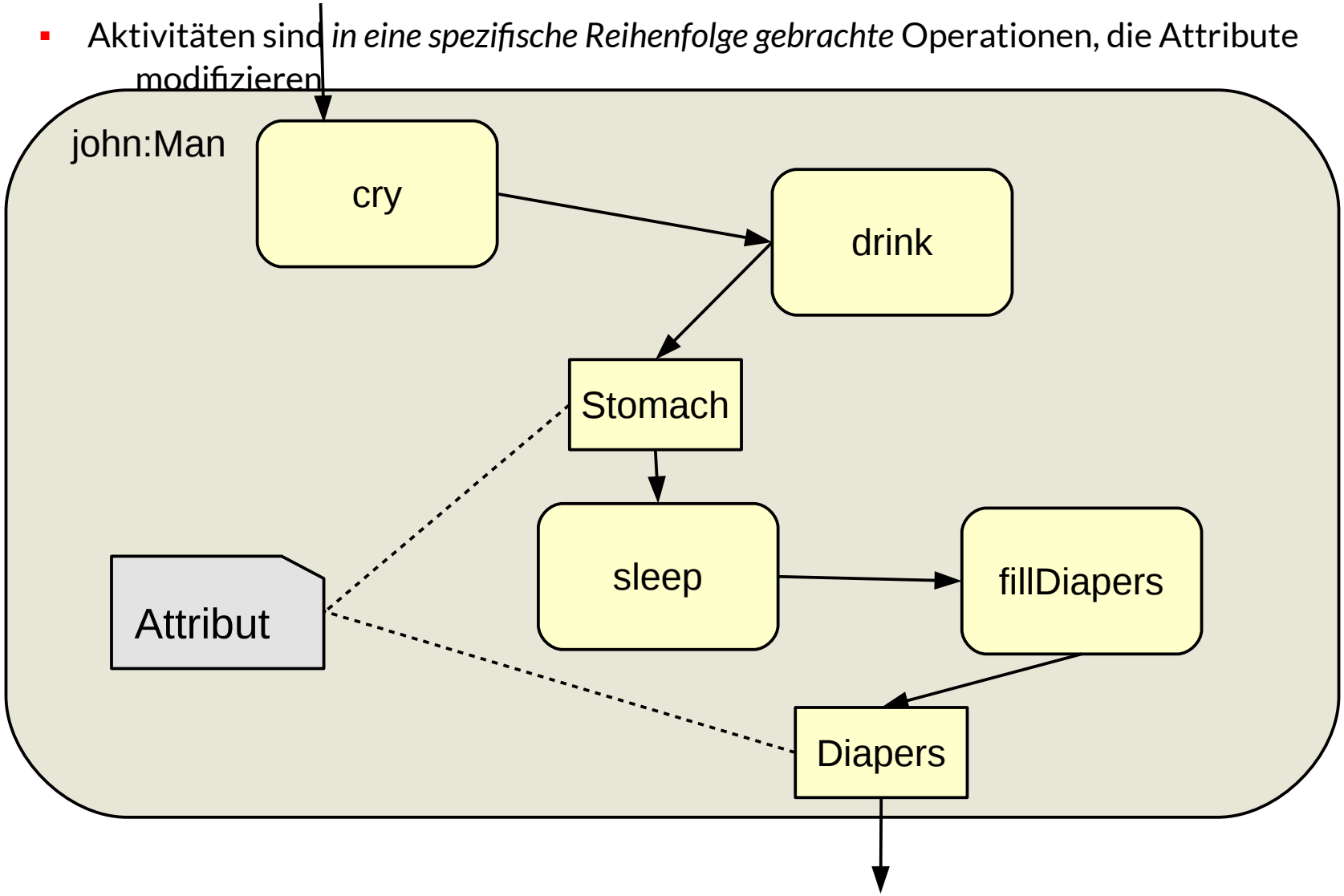
```
// Typical life of a young man
john = new Man();      // Allokation: lege Objekt im Speicher an
while (true) {
  john.cry();          // Yields: john.state == crying
  john.drink();        // Yields: john.state == drinking
  john.sleep();        // Yields: john.state == sleeping
  john.fillDiapers(); // Yields: john.state == fillingDiapers
}
```

Ein Objekt besteht aus einer Menge von Operationen (in UML: Aktivitäten)



Ein Objekt besteht aus einer Menge von Aktivitäten auf einem Zustand

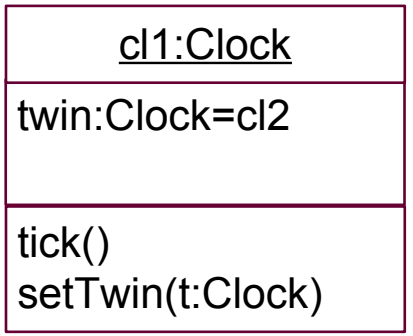
- Die Reihe der Aktivitäten (Operationen) eines Objektes nennt man **Lebenszyklus.**
- Reihenfolgen von Aktivitäten kann man in UML mit einem **Aktivitätendiagramm** beschreiben
  - Aktivitäten sind *in eine spezifische Reihenfolge gebrachte* Operationen, die Attribute modifizieren

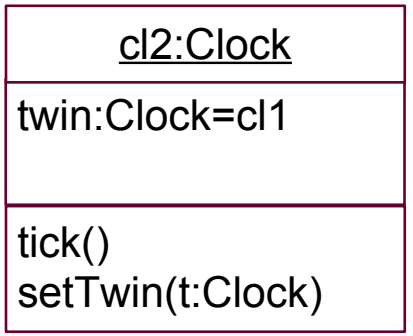


Objektnetze

- Objekte existieren selten alleine; sie müssen zu Objektnetzen verflochten werden (Objekte und ihre Relationen)
  - Ein Link von einem Objekt zum nächsten heisst *Referenz (Objekt-Assoziation)*
  - Die Beziehungen der Objekte in der Domäne müssen abgebildet werden

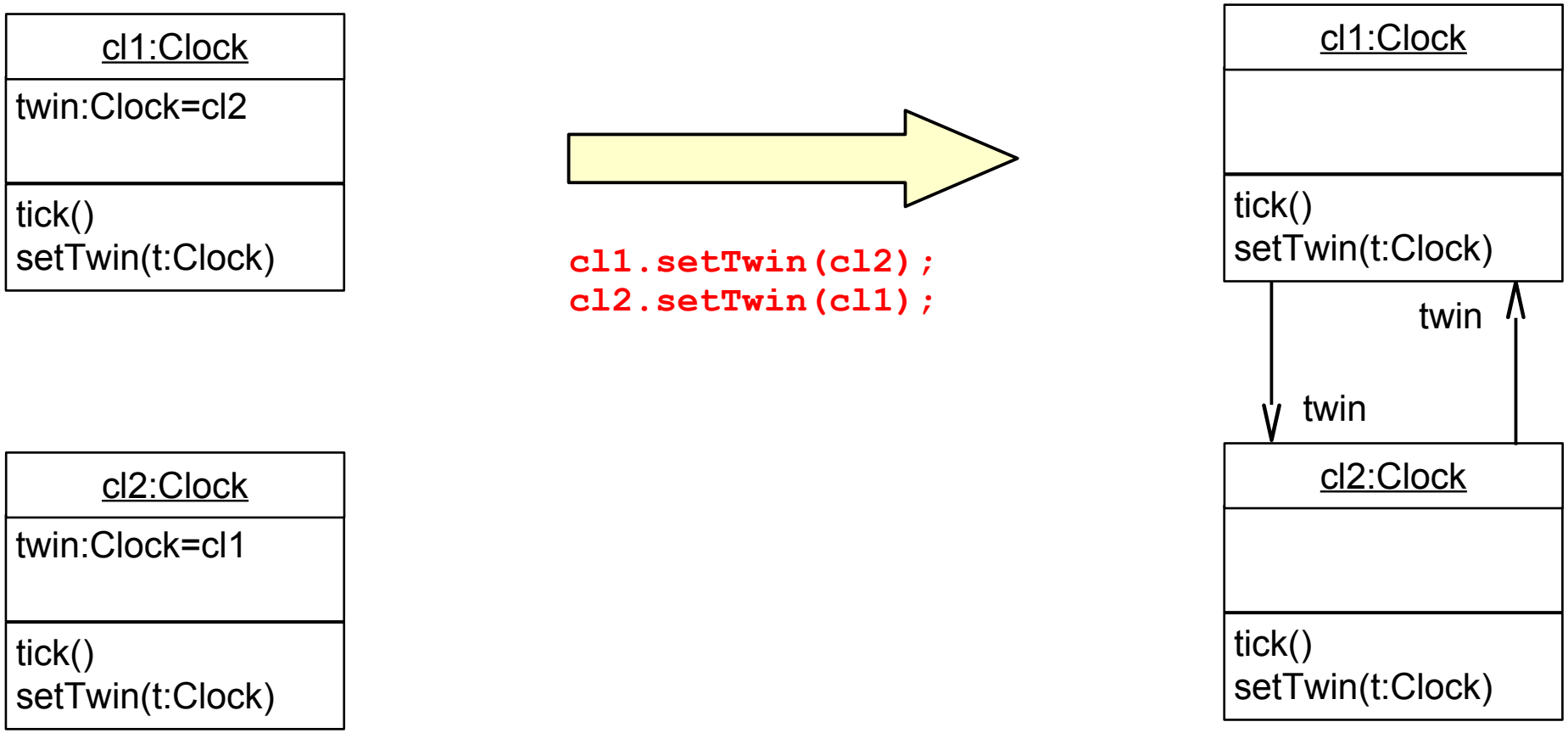
```
class Clock {
  Clock twin;
  public void tick() { minutes++;
    if (minutes == 60) { minutes = 0; hours++; }
    twin.tick();
  }
  void setTwin(Clock t) { twin = t; }
}
...
Clock cl1 = new Clock();
   // cl10.twin == undefined
Clock cl2 = new Clock();
   // cl2.twin == undefined
cl1.setTwin(cl2);          // cl1.twin == cl2
cl2.setTwin(cl1);          // cl2.twin == cl1
```





Objektwertige Attribute in Objektnetzen

- ... können als Objektnetze in UML dargestellt werden
- In Java werden also Referenzen durch Attribute mit dem Typ eines Objekts (also kein Basistyp wie int); in UML durch Pfeile



Zustandsänderungen im Ablauf eines Java-Programms


```
object Clock {
  int hours, minutes;
  public void tick() {
    minutes++; if (minutes == 60) { minutes = 0; hours++; }
  }
  public void tickHour() {
    for (int i = 1; i<= 60; i++) {
      tick();              // object calls itself
    }
  }
  public void reset() { hours = 0; minutes = 0; }
  public void set(int h, int m) { hours = h; minutes = m; }
}

public useClocks() {
  Clock cl1 = new Clock();    // cl1.hours == cl10.minutes == undef
  cl1.reset();                // cl1.hours == cl10.minutes == 0
  Clock cl2 = new Clock();    // cl2.hours == cl2.minutes == undef
  cl2.reset();                // cl2.hours == cl2.minutes == 0
  cl1.set(3,59);              // cl1.hours == 3; cl10.minutes == 59
  cl1.tick();                 // cl1.hours == 4; cl10.minutes == 0
  cl1 = null;                 // object cl1 is dead
}
```

Phasen eines objektorientierten Programms


```
class Clock { // another variant of the Clock
  public void tick() { .. }
  public void tickHour() { .. }
  public void setTwin(Clock c) { .. }
  public void reset() { .. }
  public void set(int h, int m) { .. }
}
...
Clock cl1 = new Clock(); // cl10.hours == cl10.minutes == undefined
Clock cl2 = new Clock(); // cl2.hours == cl2.minutes == undefined

cl10.reset();             // cl10.hours == cl10.minutes == 0
cl2.reset();              // cl2.hours == cl2.minutes == 0

cl10.setTwin(cl2);        // cl10.twin == cl2
cl2.setTwin(cl1);         // cl2.twin == cl1

cl10.set(3,59);           // cl10.hours == 3; cl10.minutes == 59
cl10.tick();              // cl10.hours == 4; cl10.minutes == 0

cl1 = null;               // object cl1 is dead
cl2 = null;               // object cl2 is dead
```

Allokation

Initialisierung

Vernetzung

Arbeitsphase

Abbauphase

Allokation, Aufruf und Deallokation einer Steuererklärung in Java

- Ein *Codemuster (Idiom)* ist ein Lösungsschema für wiederkehrende Programmierprobleme

```
erkl = new Einkommensteuererklärung();
   // Allokation: lege Objekt im Speicher an
erkl.initialize();
   // fill with default values
boolean firstIteration = true;
while (true) {
  if (firstIteration) {
      firstIteration = false;
  } else {
      erkl.rereadFromStore(); // Read last state
  }
  erkl.edit();          // Yields: erkl.state == editing
  erkl.store();         // Yields: erkl.state == stored
}
erkl.submit();
erkl = null;            // object dies
```

Codemuster (Idiom)
FirstIteration