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

1

Prof. Dr. rer. nat. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 13-1.1, 13.04.13

1) Objekte
2) Klassen
3) Allgemeines über Objektorientierte Programmierung

Obligatorische Literatur

- Zuser Kap 7, Anhang A
- Störrle, einleitende Kapitel
- Java:
  - Balzert, LE 3-5
  - Boles, Kap. 1-6
- Optional:
  - A. Kay. The History of Smalltalk. Second Conference on the History of Programming Languages. ACM. http://portal.acm.org/citation.cfm?id=1057828

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

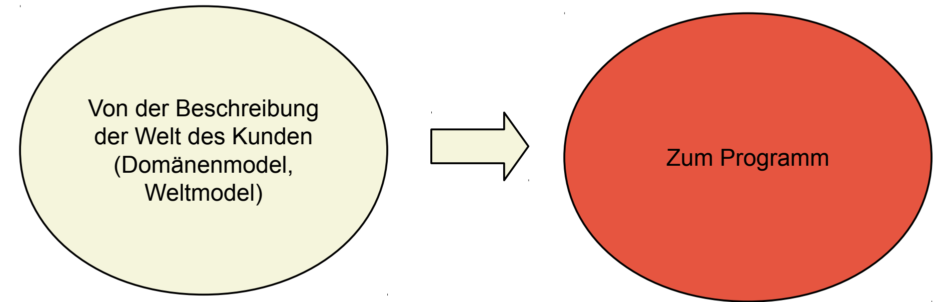
10.1. Objekte: Die Idee

Die zentralen Frage der Softwaretechnologie

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

Die zentralen Frage der Softwaretechnologie

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



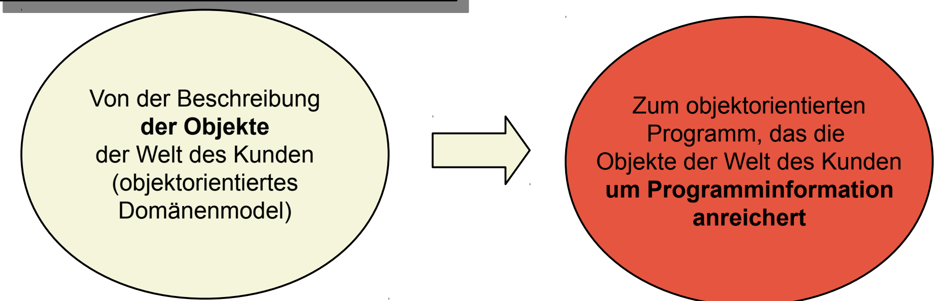
Die zentralen Fragen des objektorientierten Ansatzes

Wie können wir die Welt möglichst einfach beschreiben?

Wie können wir diese Beschreibung im Computer realisieren?

Die zentralen Fragen des objektorientierten Ansatzes

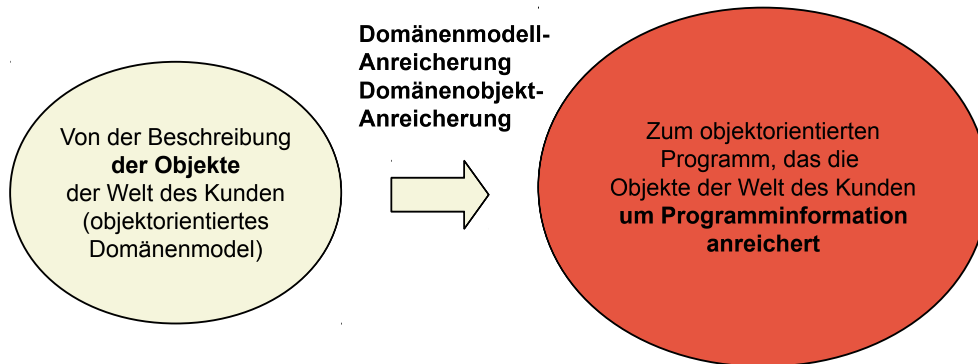
Wie können wir die Welt möglichst einfach beschreiben?



Wie können wir diese Beschreibung im Computer realisieren?

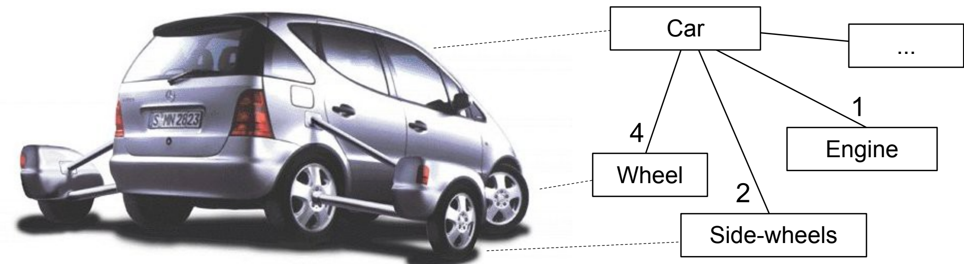
Die zentralen Fragen des objektorientierten Ansatzes

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

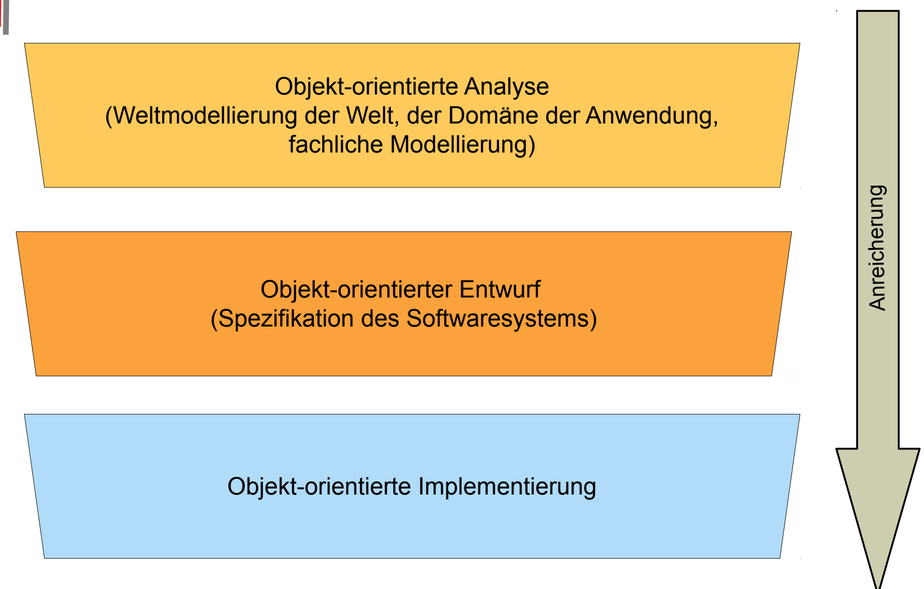
Herkunft:

Simula 1967 (Nygaard, Dahl). Ziel: technische Systeme simulieren
Smalltalk 1973 (Kay, Goldberg)

Verschiedene Antworten auf "Wie komme ich zum Programm?"

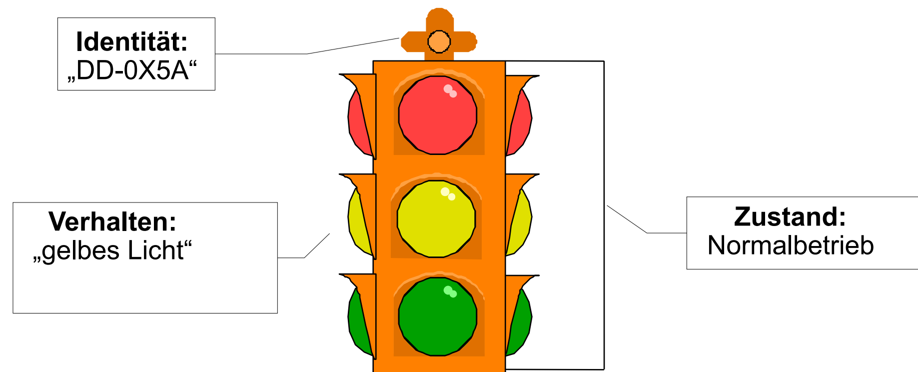
- **Funktionale Sprachen (wie Haskell, ML)**
  - Eingabe wird durch Funktionen in Ausgaben umgesetzt; f.Pr. berechnen Werte *ohne* Zustand. Keine Referenzen auf Werte möglich
- **Imperative modulare Sprachen (wie Modula, C, Ada 83)**
  - Funktionen berechnen Werte *mit Zustand*.
  - Auf Ereignisse kann 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)

Der objekt-orientierte Softwareentwicklungsprozess ganz grob



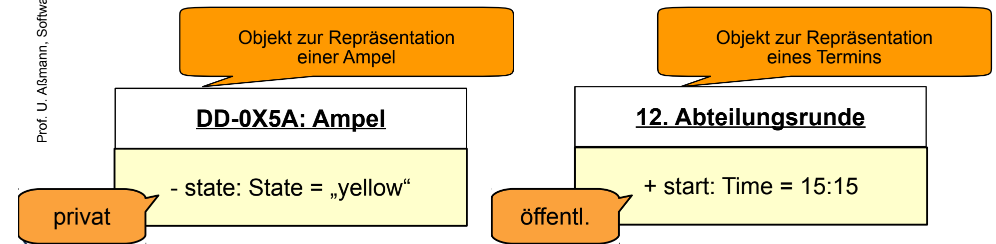
10.1.1 Grundkonzepte der Objektorientierung: Objekte einer Domäne

- 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 aus einer Domäne übernommen sind



Grundkonzepte der Objektorientierung

- Ein Objekt hat eine eindeutige ***Identität*** (Schlüssel, key, object identifier)
  - Die Identität ist unabhängig von anderen Eigenschaften.
  - Es können mehrere verschiedene Objekte mit identischem Verhalten und identischem inneren Zustand im gleichen System existieren.
  - Das Objekt ist kein *Wert (value)*, sondern es kann mehrere Namen haben (Aliase, Referenzen)
- 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 verwaltet den Zustand als ***Geheimnis*** (*Geheimnisprinzip*, *Kapselung* des Zustandes)
  - Zustand des Objekts ist zunächst Privatsache; niemand soll sich Annahmen auf den Zustand machen, ausser denen, die das Objekte *freiwillig* angibt
  - Der Zustand kann auch von außen sichtbar sein



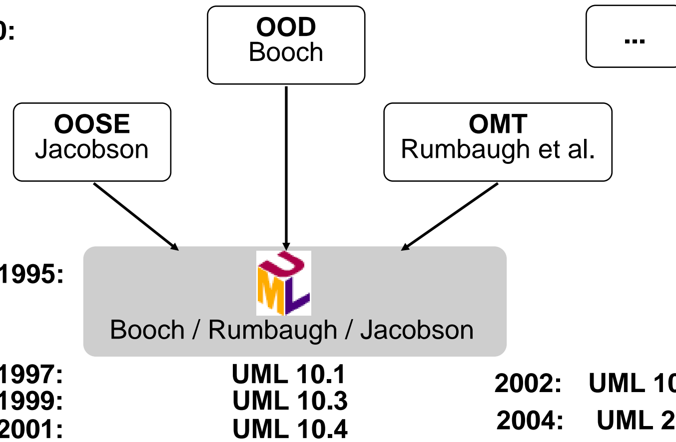
Grundkonzepte der Objektorientierung

- Ein Objekt hat ein definiertes ***Verhalten***
  - Es besitzt eine Menge genau definierter *Operationen (Funktionen, Methoden, Prozeduren)*
  - Eine Operation wird beim Empfang einer *Nachricht (Message)* ausgeführt.
  - Operationen drücken die *Zuständigkeit* eines Objektes für gewisse Aufgaben des Systems aus
  - Das Resultat einer Operation hängt vom aktuellen Zustand ab. Operationen sind nicht idempotent
- Ein Objekt besteht also aus einer Menge von ***Operationen* auf dem (lokalen) *Zustand***
  - Objekte gruppieren Operationen, die über einen Zustand, den Werten der Attribute, kommunizieren
- Ein Objekt lebt in einem ***Objektnetz*** (in Relationen, “noone is an island”)
  - Objekte müssen zu Objektnetzen verknüpft werden

Zustandswechsel einer Ampel


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

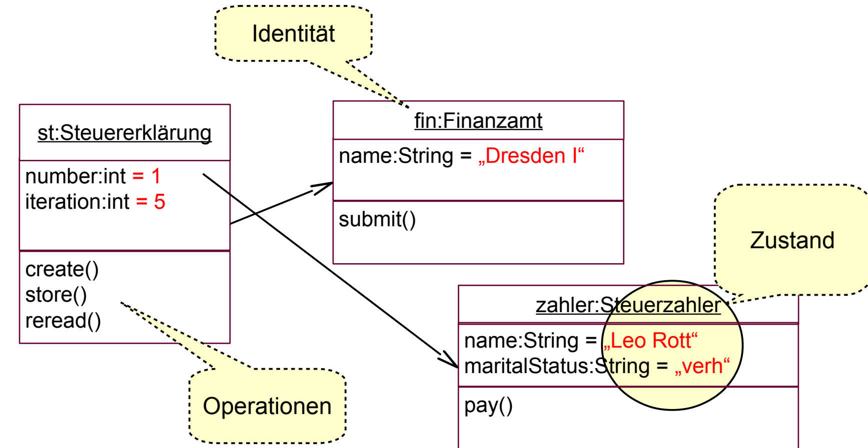
Unified Modeling Language UML



- UML ist Industriestandard der OMG (Object Management Group) http://www.omg.org/uml
- Achtung: wir verwenden hier jUML (Java-äquivalent), aUML (für Analyse), dUML (für Design).

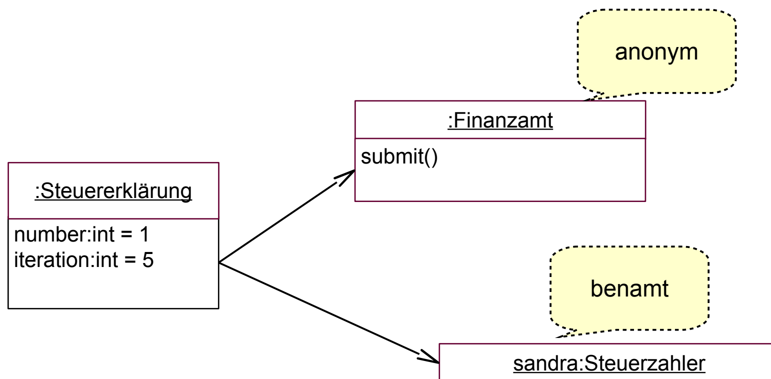
Wie sieht ein Objekt in (j)UML aus?

- 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*
- UML Objektdiagramm:



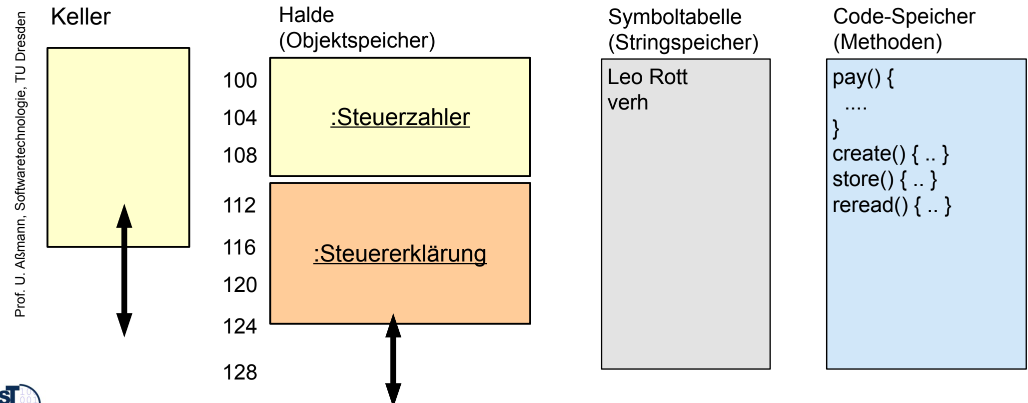
Wie sieht ein Objekt in UML aus?

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



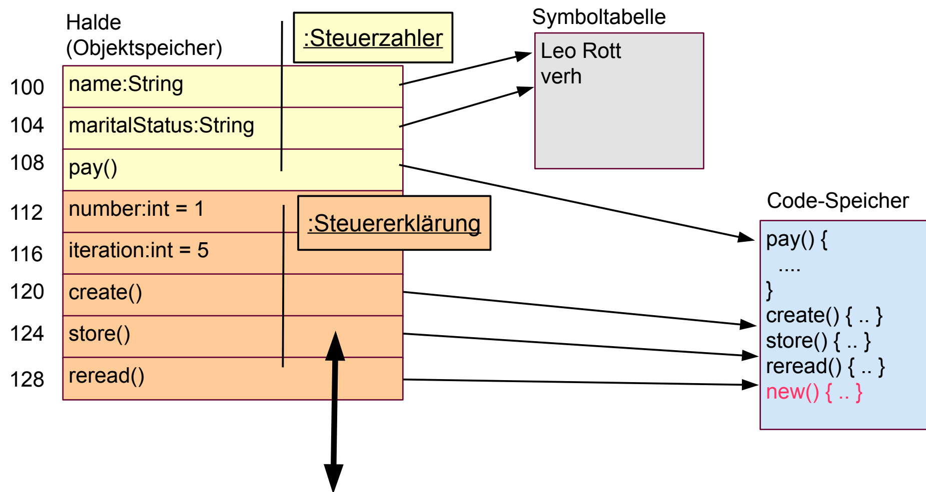
Wie sieht ein Objekt im Speicher aus? (vereinfacht)

- 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)



Wie sieht ein Objekt im Speicher aus? (vereinfacht)

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



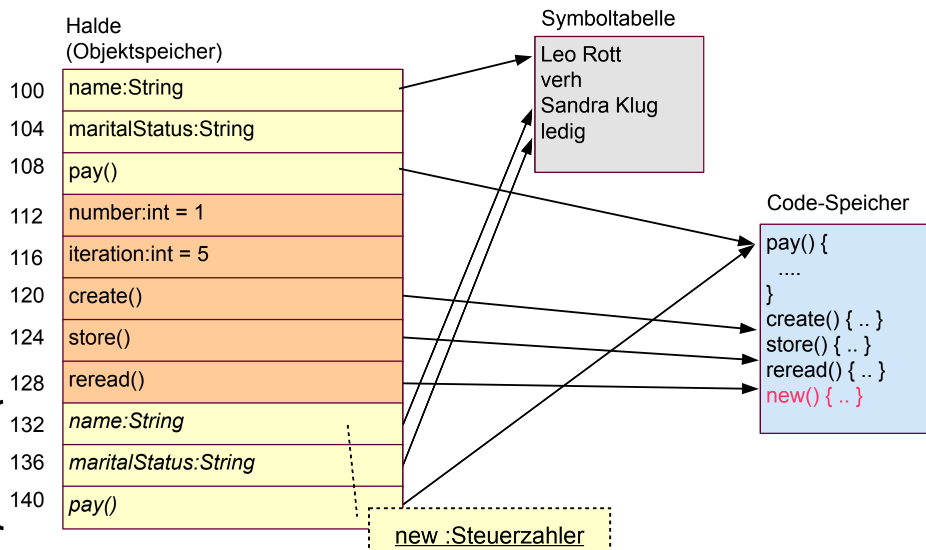
Objekterzeugung (Allokation)

- Eine Methode im Code-Speicher spielt eine besondere Rolle: der *Allokator (Objekterzeuger)* `new()`
  - Erzeugt in der Halde ein neues Objekt, mit
    - Platz für die Attribute. Der Allokator arbeitet spezifisch für den Typ eines Objekts und kennt die Platzbedarf eines Objekts
    - Platz für die Zeiger auf die Methoden
    - Initialisierung: Der Allokator erhält die Initialwerte von Attributen als Parameter
- In Java:

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


Was passiert bei der Objekterzeugung im Speicher?


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

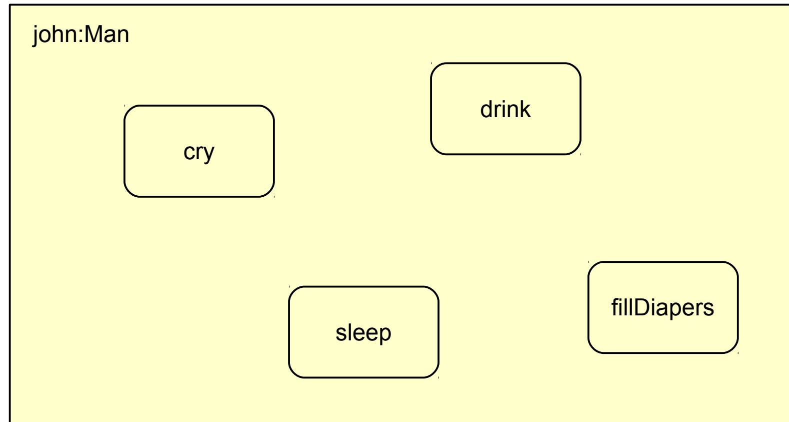


Allokation und Aufruf eines Objektes in Java

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

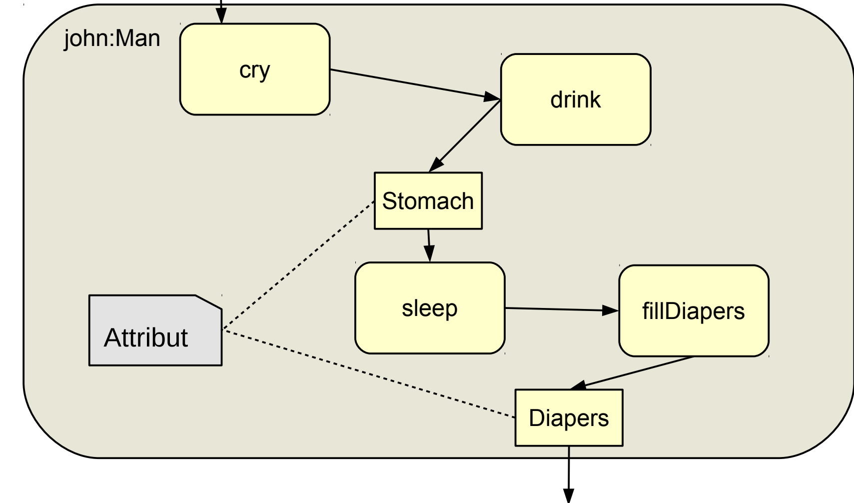
```
// Typical life of a young man
john = new Man();     // Allokation: lege Objekt im Speicher an
while (true) {
  john.cry();         // Yields: john.state == crying
  john.drink();       // Yields: john.state == drinking
  john.sleep();       // Yields: john.state == sleeping
  john.fillDiapers();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

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



Ein Objekt besteht aus einer Menge von Aktivitäten

- 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



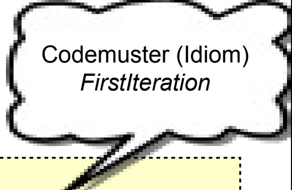
Zustandsänderungen im Ablauf eines Java-Programms


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

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.shipToTaxOffice();
erkl = null;            // object dies
```



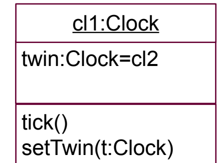
Was können Objekte eines Programms darstellen?

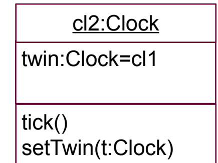
- **Simulierte Objekte** der realen Welt (der *Anwendungsdomäne*)
  - Ampeln, Uhren, Türen, Menschen, Häuser, Maschinen, Weine, Steuerzahler, etc.
- **Simulierte abstrakte Dinge** (Anwendungsobjekte, fachliche Objekte, Geschäftsobjekte, business objects)
  - Adressen, Rechnungen, Löhne, Steuererklärungen,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

10.1.2 Objektnetze

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

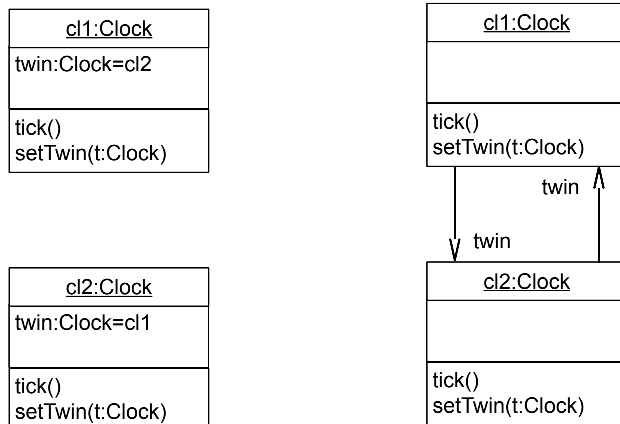
```
class TwinClock {
  TwinClock twin;
  public void tick() { minutes++;
    if (minutes == 60) { minutes = 0; hours++; }
    twin.tick();
  }
  void setTwin(TwinClock t) { twin = t; }
}
...
TwinClock cl1 = new TwinClock();
   // cl10.twin == undefined
TwinClock cl2 = new TwinClock();
   // cl2.twin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





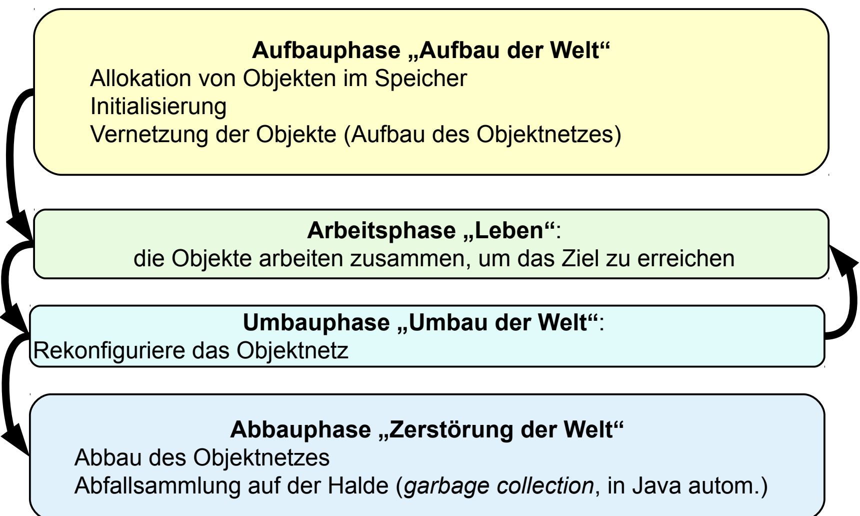
Objektwertige Attribute in Objektnetzen

- ... 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



10.1.3 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 Clock { // assume a combination of previous Clock and TwinClock
  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

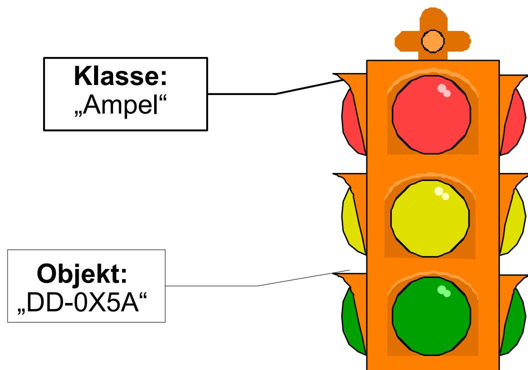
Abbauphase

10.2. Klassen

Softwaretechnologie, © Prof. Uwe Aßmann
Technische Universität Dresden, Fakultät Informatik

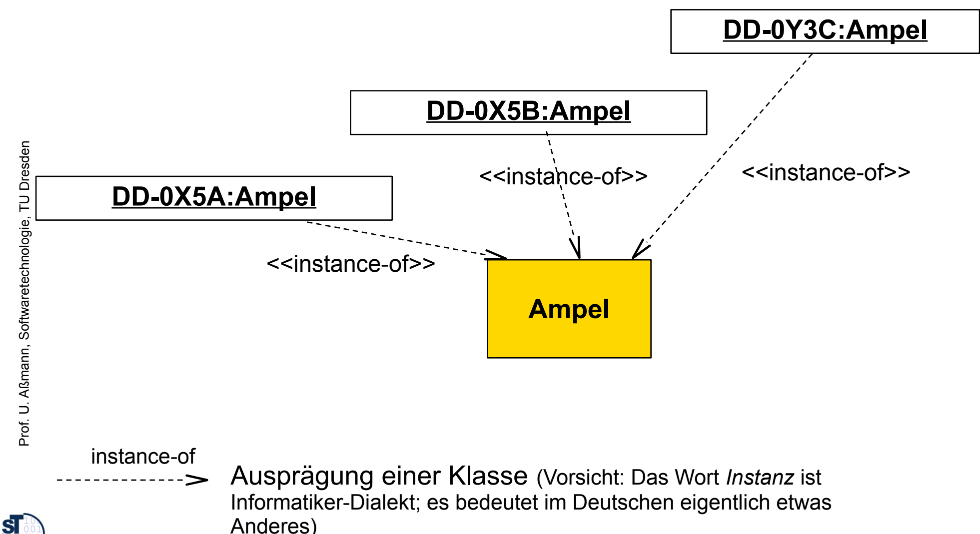
Klasse und Objekt

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



Beispiel: Ampel-Klasse und Ampel-Objekte

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



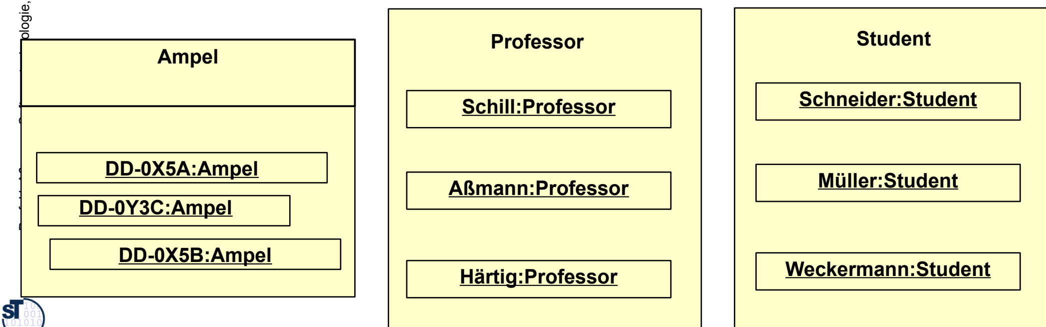
Ausprägung einer Klasse (Vorsicht: Das Wort *Instanz* ist Informatiker-Dialekt; es bedeutet im Deutschen eigentlich etwas Anderes)

Klassen als Begriffe, Mengen und Schablonen

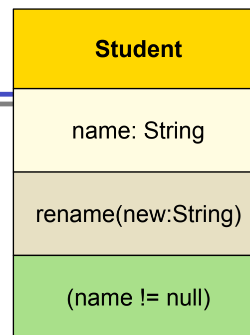
- **Begriffsorientierte Sicht:**
  - Eine Klasse stellt einen Begriff dar. Dann *charakterisiert* die Klasse ein *Konzept*
  - Man nennt das Modell dann eine *Begriffshierarchie*, *Begriffswelt* bzw. *Ontologie* (gr. Lehre von der Welt)
- **Mengenorientierte Sicht:** (z.B. in Datenbanken)
  - Eine Klasse *enthält* eine Menge von Objekten
  - Eine Klasse *abstrahiert* eine Menge von Objekten
- **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)
  - Die Klasse schreibt die innere **Struktur** vor (Strukturäquivalenz)
  - Die Klasse kann auch das **Verhalten** ihrer Objekte vorschreiben (Verhaltensäquivalenz)
- Bildung eines neuen Objektes aus einer Klasse:
  - Eine Klasse enthält einen speziellen Repräsentanten, ein spezielles Objekt, ein Schablonenobjekt (Prototyp) für ihre Objekte.
  - Ein Objekt wird aus einer Klasse erzeugt (Objektallokation),
    - in dem der Prototyp in einen neuen, freien Speicherbereich kopiert wird.
    - Damit erbt das neue Objekt das Verhalten der Klasse (die Operationen), das Prädikat und die Attribute des Prototyps (meistens Nullwerte wie 0 oder null)

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**)

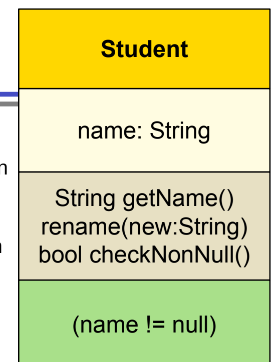


Merkmale von Klassen und Objekten



- 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). Attribute 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**, Bedingungen, die für alle Objekte zu allen Zeiten gelten
- 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.

Arten von Methoden



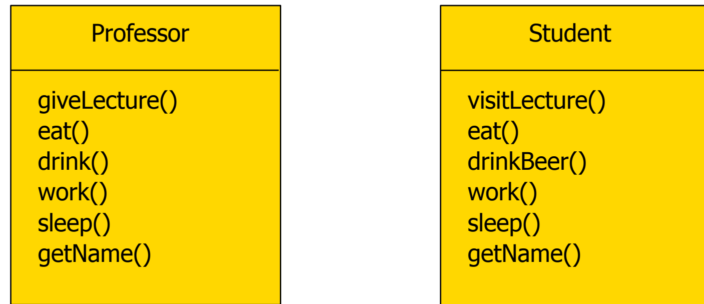
Zustandsinvariante Methoden:

- **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:

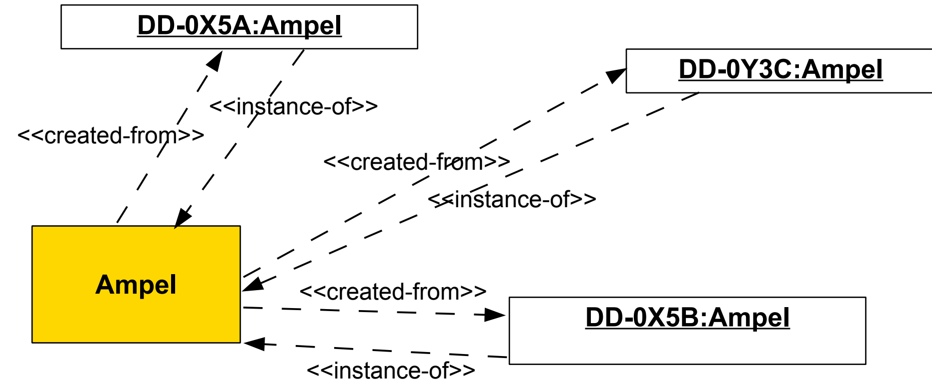
- **Modifikatoren (Mutatoren)**
  - **Attributmodifikatoren** ändern den (lokalen) Zustand des Objekts
  - **Netzmodifikatoren** ändern die Vernetzung des Objektes
- **Repräsentationswechsler (representation changers)** transportieren das Objekt in eine andere Repräsentation
  - z.B. drucken den Zustand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

Klasse in schablonenorientierter Sicht Professoren und Studenten



Klasse in schablonenorientierter Sicht

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

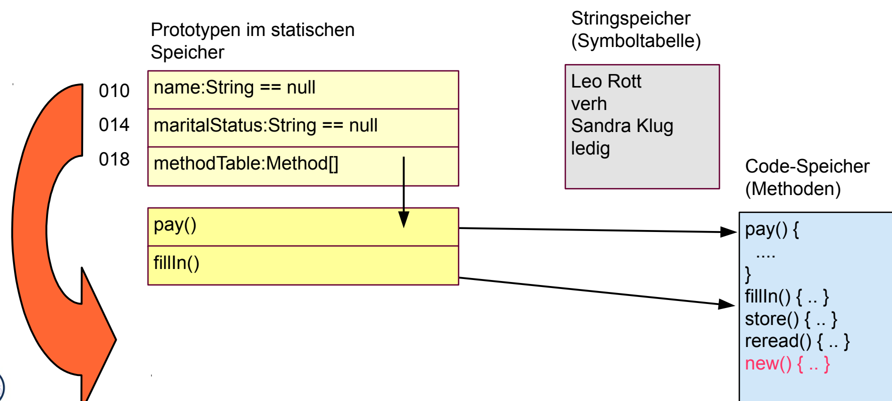


Wie sieht eine Klasse im Speicher aus?

- 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

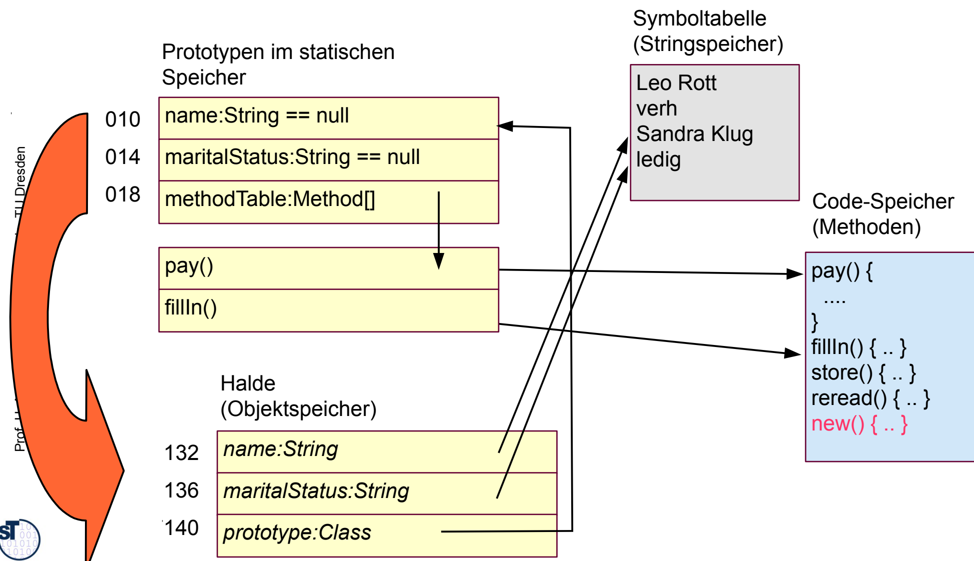
Wie erzeugt man ein Objekt für eine Klasse? (Verfeinerung)

- `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



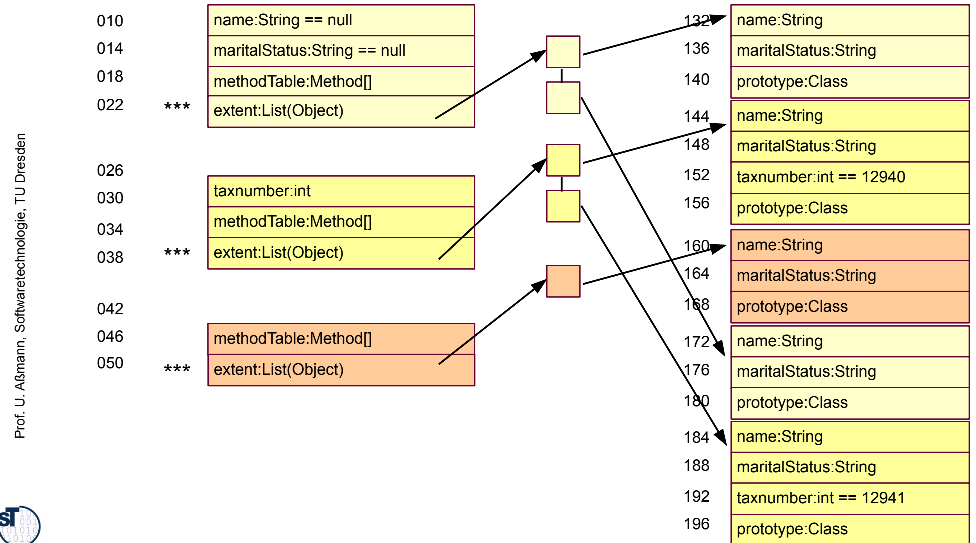
Wie erzeugt man ein Objekt für eine Klasse? (Verfeinerung)

- Ergebnis:



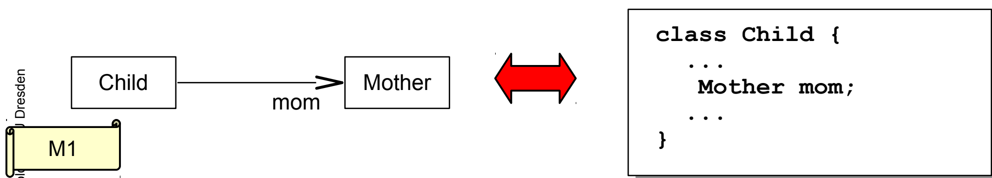
Objekt-Extent im Speicher

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



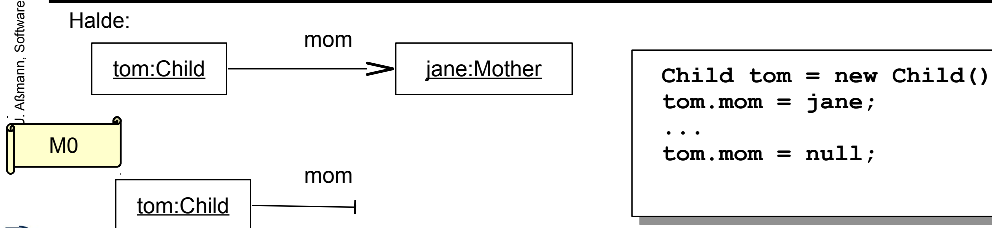
Klassennetze und Objektnetze

- 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*




```
class Child {
  ...
  Mother mom;
  ...
}
```

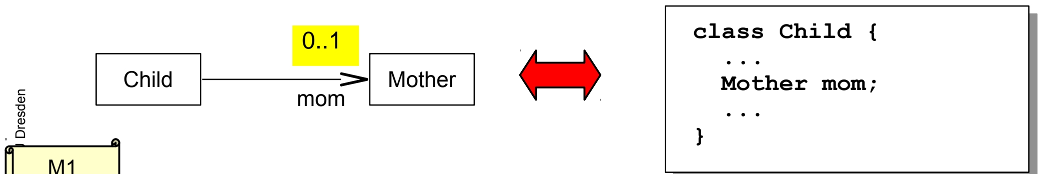
Halde:




```
Child tom = new Child();
tom.mom = jane;
...
tom.mom = null;
```

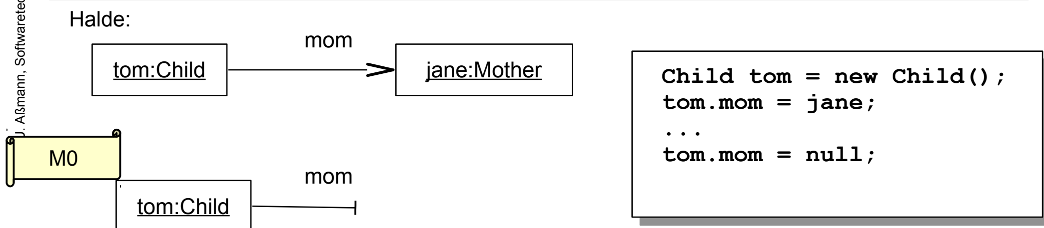
Invarianten von Assoziationen

- Die Obligationen können aus *Multiplizitäten* für die Anzahl der Partner bestehen (*Kardinalitätsbeschränkung* oder *Invariante* der Assoziation)
- *"Ein Kind kann höchstens eine Mutter haben"*




```
class Child {
  ...
  Mother mom;
  ...
}
```

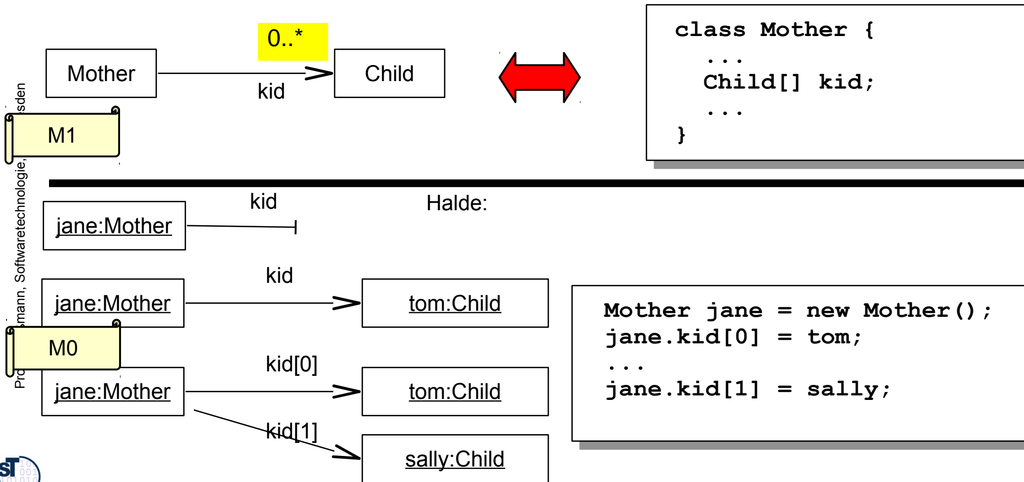
Halde:




```
Child tom = new Child();
tom.mom = jane;
...
tom.mom = null;
```

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;
  ...
}
```

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


Die zentralen Frage der Softwaretechnologie

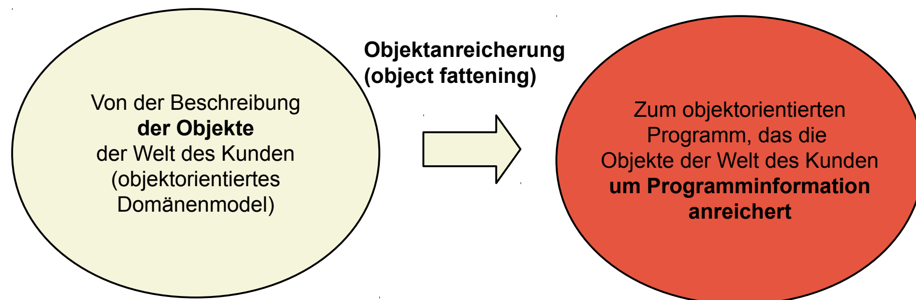
Wie kommen wir vom Problem des Kunden zum Programm (oder Produkt)?

Wie können wir die Welt möglichst einfach beschreiben?

Wie können wir diese Beschreibung im Computer realisieren?

Antwort der Objektorientierung: Durch "Objektanreicherung (object fattening)"

Wie können wir diese Beschreibung im Computer realisieren?



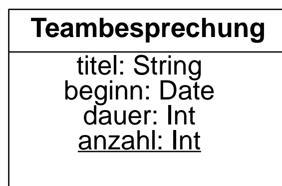
Objekt der Domäne
Verfettung: Anreicherung durch technische Programminformation

10.2.1 Merkmale von Klassen

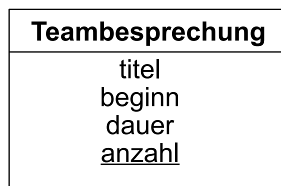
Klassenattribut (Statisches Attribut)

- 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:

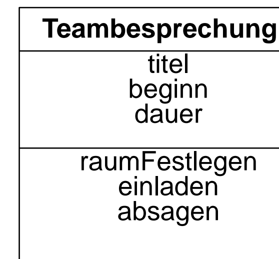


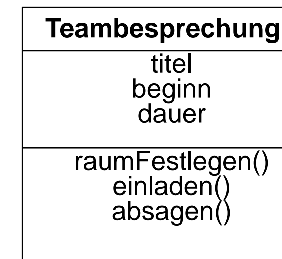
aUML für Analyse:



Operationen

- **Def.:** Eine ***Operation (Instanzoperation)*** einer Klasse *K* ist die Beschreibung einer Aufgabe, die jedes Objekt der Klasse *K* ausführen kann.





- "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

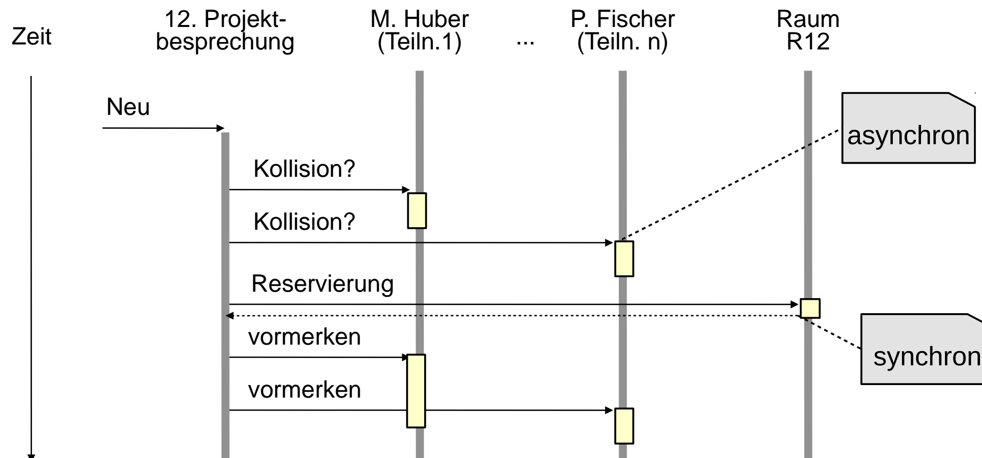
- ***Operation***: "a service that can be requested from an object to effect behaviour" (UML-Standard)
- ***Message (Botschaft, Nachricht):*** eine Nachricht an ein Objekt,
  - um eine Operation auszuführen
  - um ein externes Ereignis mitzuteilen
- ***Methode:*** "the implementation of an operation (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.

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

- Nachrichten starten synchrone oder asynchrone Methoden



Kooperierende Objekte mit lokaler Datenhaltung
Einfacher Übergang zu Parallelverarbeitung und Verteilung

Spezifikation von Operationen

- **Definition** Die *Spezifikation* einer Operation legt das Verhalten der Operation fest, ohne einen Algorithmus 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

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

Besprechungsraum
raumNr kapazität
reservieren() freigeben() freienRaumSuchen() new()

```
class Steuererklaerung {

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


10.3 Objektorientierte Programmierung

My guess is that object-oriented programming will be in
the 1980's what structured programming was in the 1970's.
Everyone will be in favor of it.
Every manufacturer will promote his products as supporting it.
Every manager will pay lip service to it.
Every programmer will practice it (differently!).
And no one will know just what it is ..."

Tim Rentsch (1982)

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

- ***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

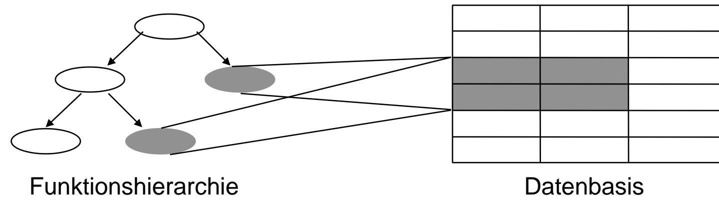
```
public team ProducerConsumer {
  public role class Producer { void put(int n); }
  public role class Consumer { void get(int n); }
} ...
```

Orthogonalität

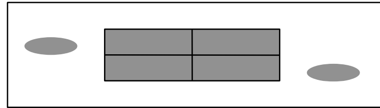
	Strukturiert	Modular	Objekt-orientiert	Rollen-orientiert
Funktional	– –	**ML**	**Haskell**	– –
Imperativ	**Pascal**	**Modula-2 Ada**	**Java C++**	**ObjectTeams**
Logisch Mengenorientiert	– –	**XSB-Prolog**	**OWL**	– –

Funktion und Daten

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

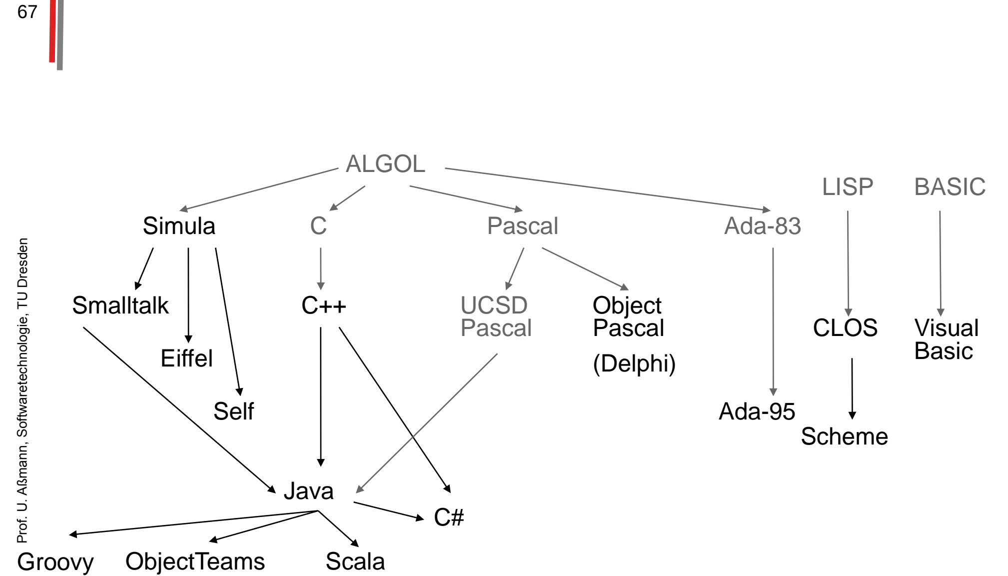


- ***Integration*** von Funktion und Daten in der objektorientierten Programmierung



- 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)

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

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 unabhängig von Hardware und Betriebssystem.
  - Java *Bytecode* in der *Java Virtual Machine*
- Java ist angepaßt an moderne Benutzungsoberflächen.
  - *Java Swing Library*
- Java ermöglicht nebenläufige Programme (*multi-threading)*
- Java bietet die Wiederbenutzung von Klassenbibliotheken (Frameworks, Rahmenwerken) an
  - z.B. Java *Collection Framework*

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

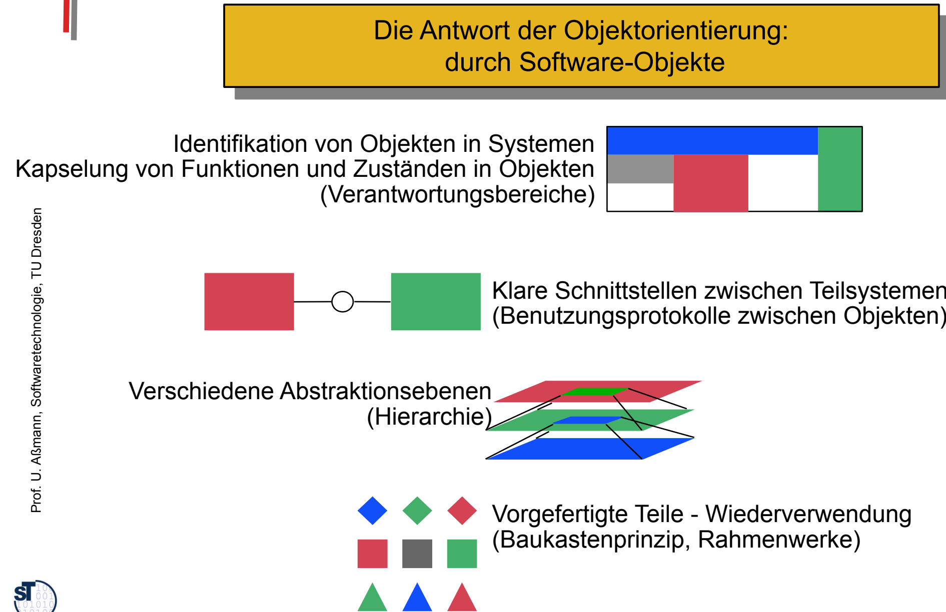
- Keine Verhaltensgleichheit von Klassen garantiert (keine konforme Vererbung):
  - Man kann bei der Wiederbenutzung von Bibliotheksklassen Fehler machen und den Bibliothekscode *invalidieren*
- Basisdatentypen (int, char, boolean, array) sind keine Objekte
  - Anders in C#!
- JVM startet langsam. Beim 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 für Anwendungen, die mit der realen Welt zu tun haben, eine *oftmals adäquate* Entwicklungsmethodik an
  - Die die Software *einfach*, d.h. analog zur Welt organisiert
  - Und daher viele Probleme vermeidet und die Entwicklung erleichtert

Appendix

Die Welt *einfach* modellieren



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

Beispiel: Operationen im Analysemodell