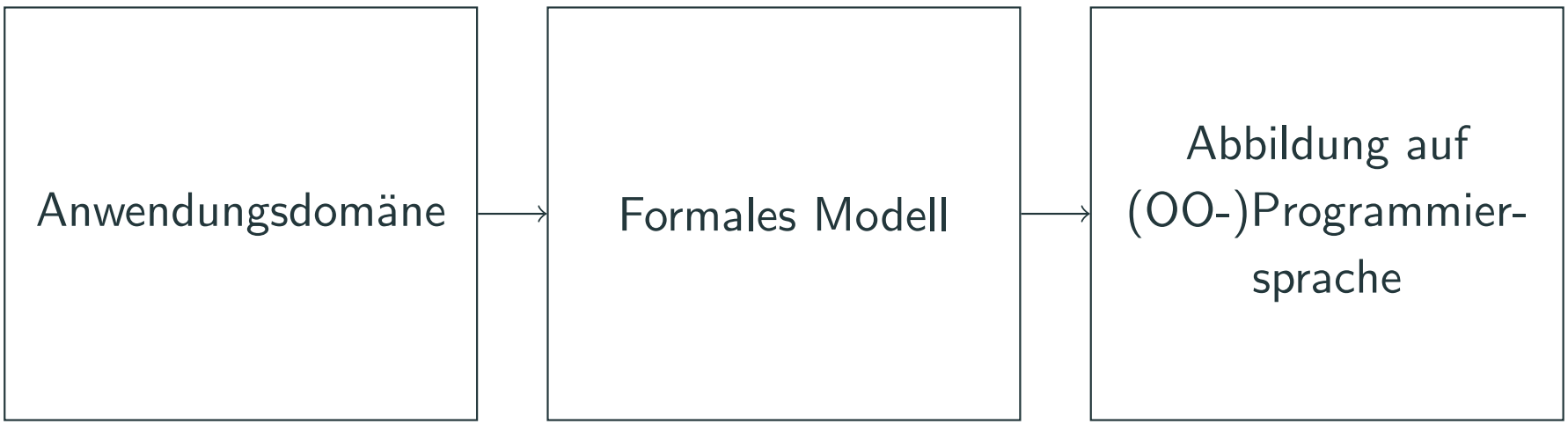
Kapitel 2: Objektbeziehungen und Modellierung

Vom Problem zum Modell

- Probleme liegen in der Anwendungsdomäne vor
- Lösungen werden durch Programmcode umgesetzt
- Dazwischen:
  - (OO-)Analyse $\Rightarrow$ Abstraktes, fomales Modell des Problems
  - (OO-)Design $\Rightarrow$ Abbildung auf konkrete Programmiersprache



OOA und OOD

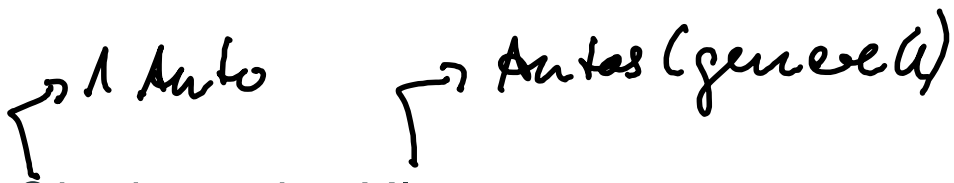
- Analyse und Design schon immer als Schritte vom Problem zum Programm geläufig
- Objekte als Modellierungsmechanismus seit 1960er Jahren
- Spätestens seit Coad/Yourdon (1990) wurden die Analyse- und Design-Methoden formalisiert und vereinheitlicht
  - Object Oriented Analysis (OOA): Finden von Abstraktionen in Form von Klassen, Objekte, deren Methoden, Attribute und Beziehungen als Abbildung des realen Problems
  - Object Oriented Design (OOD): Verfeinerung der Elemente auf abstrakter Ebene und Abbildung auf in einer konkreten Programmiersprache realisierbare Modellierungstechniken

OO-Modelle

- Abstrakte Notation, oft als Grafiken oder Dokumente
- Verschiedene Aspekte der Modellierung werden in unterschiedlichen Diagrammtypen parallel betrachtet
  $\Rightarrow$z.B. Aspekte von Struktur und Verhalten
- Anspruch: (Programmier-)sprachneutral
- In den 1990ern: Vielzahl konkurrierender OOA/OOD-Methoden mit unterschiedlichen Notationen und Konzepten
- 1994-97 Vereinigung der Methoden von Grady Booch, James Rumbaugh (OMT) und Ivar Jacobson (OOSE) zu UML

Anforderungsphase eines Projekts: Textanalyse

- Oft: Anforderungen des Kunden als *natürlichsprachlicher Text* (nicht formalisiert, aber verbindliche Vorgabe)
- Mögliche Strategie für Anforderungstext⇒formales Modell: finden von *Wortklassen*, die sich auf Konzepte von (OO-)Programmiersprachen abbilden lassen, z.B.



Substantive ⇒ Objekte oder Klassen

Verben ⇒ Methoden (C++: Member Functions)

Adjektive ⇒ Attribute (C++: Data Members)

Beispiel: Text-to-Model

Vorgabe:

Car and boat are specialized kinds of vehicles. A vehicle can move by a certain distance, which increases its mileage accordingly. Cars have Wheels. Vehicles can carry passengers.

⇒ **Kandidaten für**

Klassen: . Car, Boat, Vehicle, Distance, Mileage, Wheel, Passenger

Methoden: . Vehicle::move(distance) Mileage::increase(distance) Vehicle::carry(passenger)

Attribute: . Car::wheels Vehicle::passenger Vehicle::mileage

Beispiel: Text-to-Model

Vorgabe:

Car and boat are specialized kinds of vehicles. A vehicle can move by a certain distance, which increases its mileage accordingly. Cars have Wheels. Vehicles can carry passengers.

⇒ **Kandidaten für**

Klassen: `Car`, `Boat`, `Vehicle`

Methoden: `Car::move(distance)`, `Car::carry(passenger)`, `Mileage::increase(distance)`

Attribute: `Wheel`, `Passenger` , `Mileage`

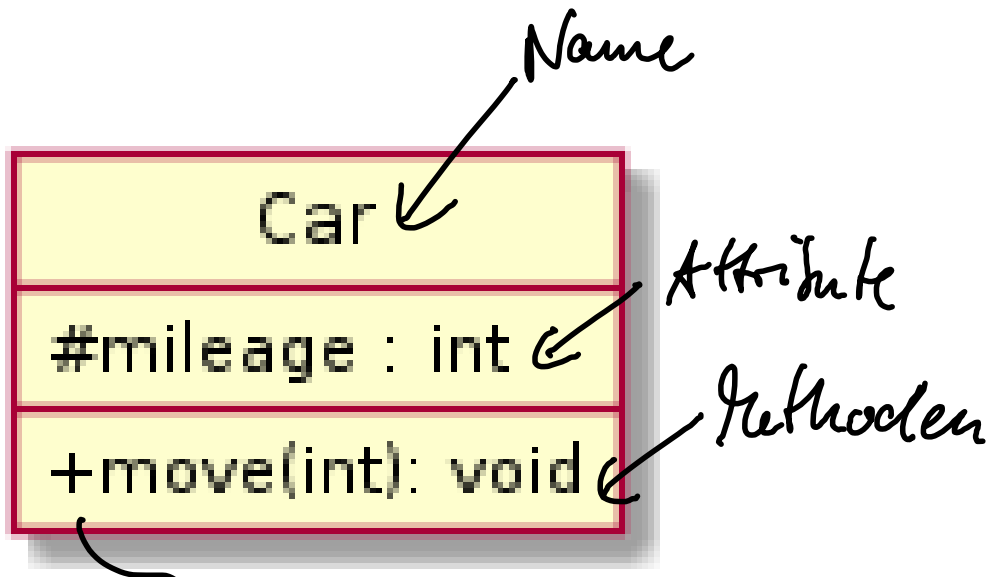
- Klassen teilw. "degradiert" zu Attributen
- Hier keine Attribute aus Adjektiven, sondern aus Substantiven begründete Objekte, die aber selbst Teile anderer Objekte sind

UML

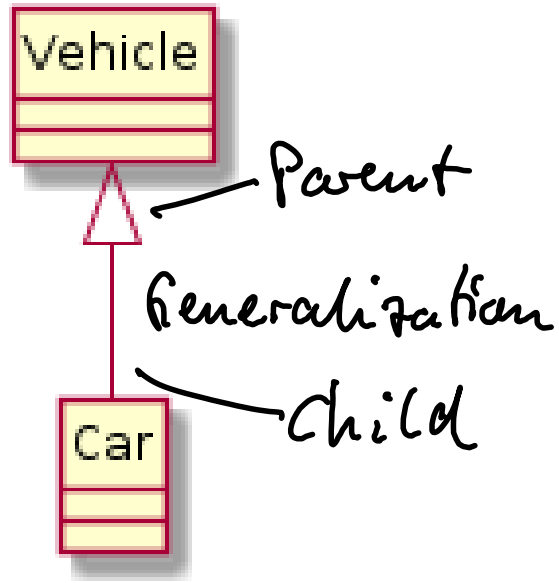
- Seite 1997 OMG-Standard für Objektmodellierung
  OMG = Object Management Group (`http://omg.org`)
- Modellelemente sind selbst als Modell beschrieben: "Metamodell"
- Hat die meisten konkurrierenden Ansätze verdrängt bzw. assimiliert
- Meilenstein 2005: UML2 $\Rightarrow$ u.a. neue Diagrammtypen, bessere Abbildung auf XML-Dokumente
- Sprachneutral
- Hier Einführung der ersten Elemente, mehr im Verlauf des Semesters und in SWT

UML: Klassen und Klassenhierarchien

Klasse:



Vererbung:

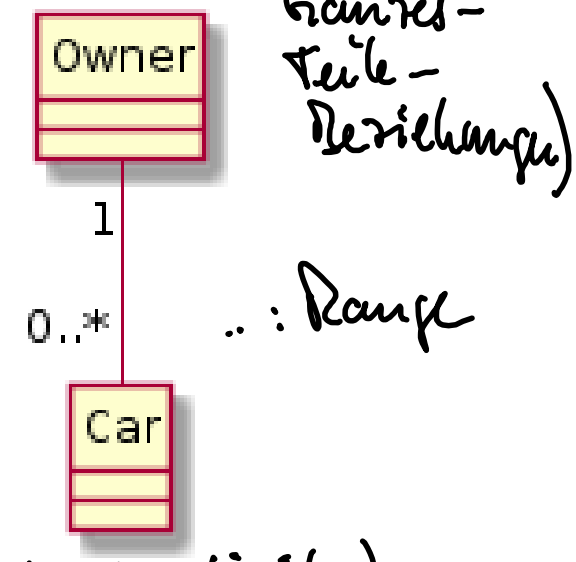


Sichtbarkeit:

- \+ public
- \# protected
- – private
- ~ package (f. Java)

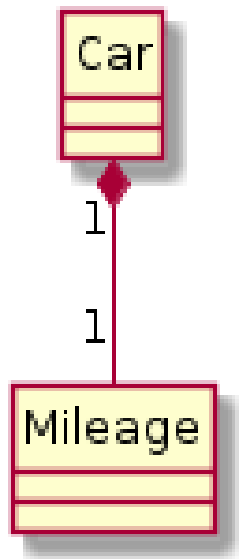
UML: Assoziationen

Association: (generisch)



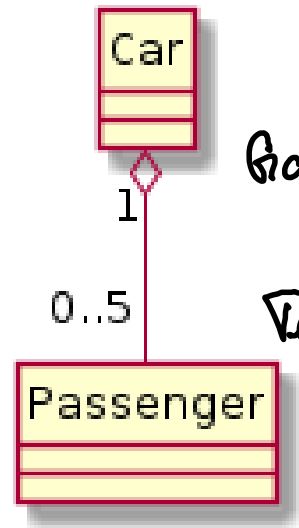
Kardinalität:
"Car is owned by *exactly* 1 owner", "An owner can have 0 up to any number of cars"

Composition:



Teil (Part) kann nicht ohne Whole (Ganzes) existieren

Aggregation:



Teil ist unabhängig von Ganzem, aber assoziiert

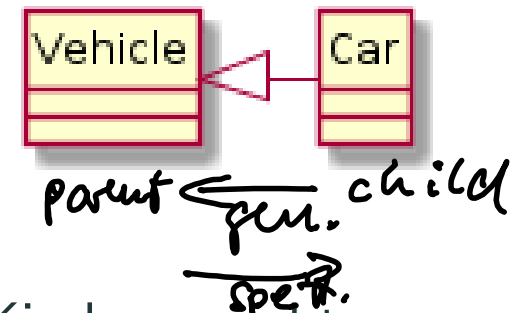
Kapselung

- Grundstrategie: Data Hiding, Verstecken von Details der inneren Datenstruktur eines Objekts
- Konzept des "Abstrakten Datantypen"
  (Abstract Data Type, ADT)
  - Daten versteckt (Inneres ist *opak*) opaque
  - Typ weist sich durch seine Schnittstelle (Interface) aus
- Ziele:
  - Integrität der Daten: können nicht von außen manipuliert werden
  - Portabilität: Implementierung kann ausgetauscht werden
    Voraussetzung: Schnittstelle bleibt stabil
  - Wiederverwendbarkeit: Interface-Fokussierung bedeutet Design für Verwendung, nicht Implementierung
    $\Rightarrow$Chance für andere Verwendungsmöglichkeiten erhöht sich

Vererbung

- Idee aus Taxonomien, z.B. in der Biologie
  $\Rightarrow$ Stammbaum der Arten
- Objekte werden aufgrund gemeinsamer Eigenschaften *klassifiziert*
  $\Rightarrow$ Zugehörigkeit zu einer *Klasse*
- Zwischen Klassen besthehen Verwandschaftsbeziehungen:

  Spezialisierung $\Leftrightarrow$ Generalisierung

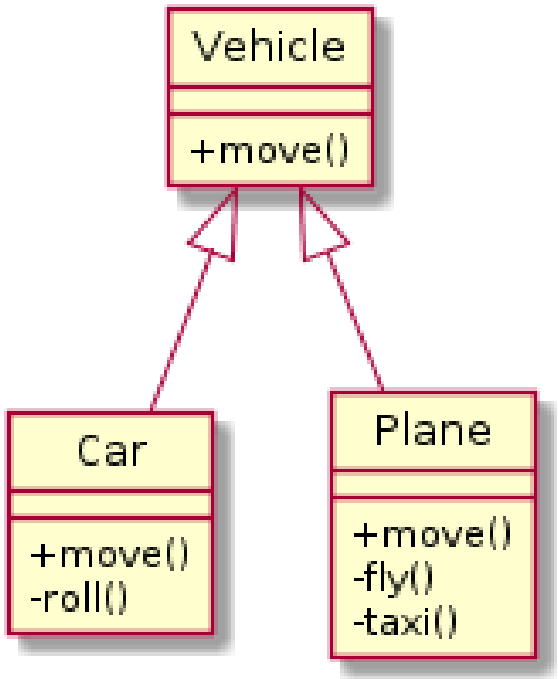


- Eigenschaften werden von Eltern an Kinder vererbt
- Kind-Objekte können deshalb unter bestimmten Umständen auch als Objekte vom Typ der Eltern auftreten

Vorsicht: "Eltern" und "Kind" werden hier *taxonomisch* betrachtet, nicht mit der natürlichen Eltern-Kind-Beziehung verwechseln!

Method Overriding und Polymorphie

Vererbte Methoden müssen z.T. in Kindklassen spezialisiert werden:



`move()` ist in `Car` und `Plane` *unterschiedlich* implementiert:

- Bei `Car` basierend auf `roll()`,
- bei `Plane` and Mix aus `taxi()` und `fly()`

- `move()` ist in `Vehicle` generalisiert, `Car` und `Plane` *überschreiben* (override) die Methode von `Vehicle`.
- Ziel: `move()` an einem `Vehicle`-Objekt aufrufen, das ein `Car` oder `Plane` sein kann.

Die drei Säulen der Objektorientierung

Sind

- Kapselung
- Vererbung
- Polymorphie

...sehen wir uns in Kapitel 3 und 4 konkret für C++ an