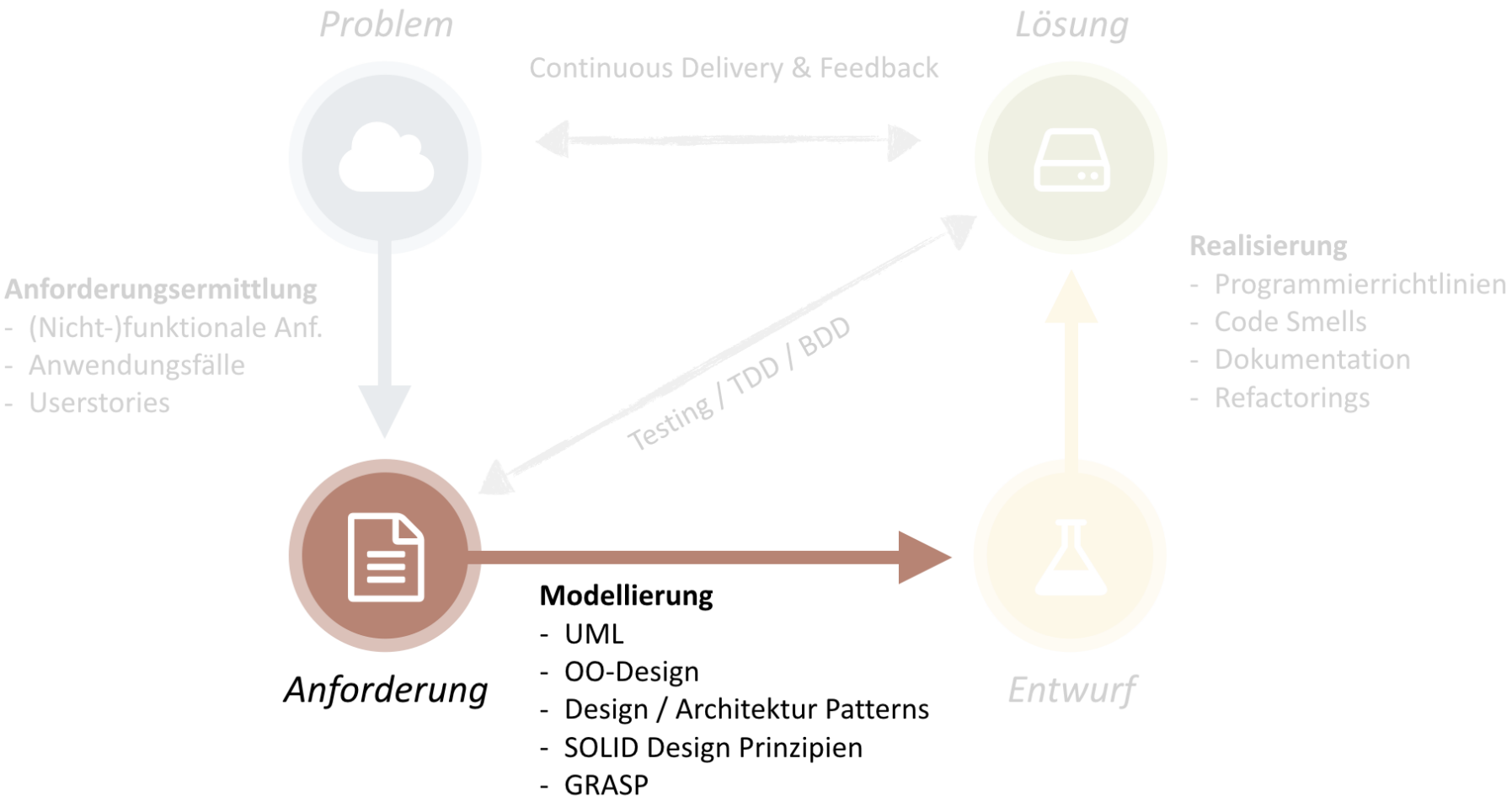
Modellierung mit UML

Jonathan Brachthäuser (mit Folien von Theo Doukas)

10. Mai 2017

Die Folien orientieren sich u.a. am Kurs *Software Engineering* von Hans-Werner Six und Mario Winter.

Einordnung


Problem
Lösung
Continuous Delivery & Feedback
Anforderungsermittlung
- (Nicht-)funktionale Anf.
- Anwendungsfälle
- Userstories
Testing / TDD / BDD
Realisierung
- Programmierrichtlinien
- Code Smells
- Dokumentation
- Refactorings
Modellierung
- UML
- OO-Design
- Design / Architektur Patterns
- SOLID Design Prinzipien
- GRASP
Anforderung
Entwurf

Anforderungen und Designs kommunizieren

Unified Modeling Language (UML)

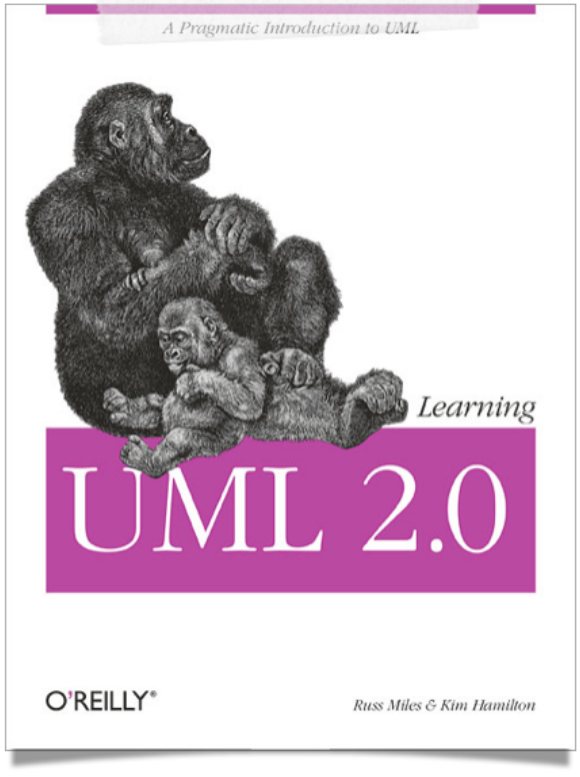
Strukturelle Modellierung: Objekt- und Klassendiagramme

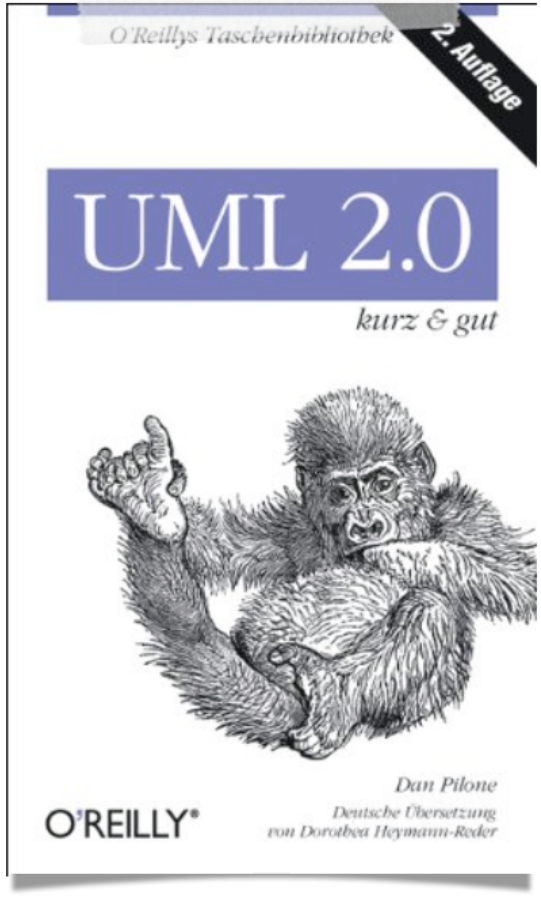
- Objektdiagramme
- Klassen
- Assoziationen
- Aggregationen
- Generalisierung und Interfaces
- Weitere Modellierungselemente

Verhaltensmodellierung

- Interaktionsdiagramm
- Zustandsdiagramm

Literatur


A Pragmatic Introduction to UML
Learning
UML 2.0
O'REILLY
Russ Miles & Kim Hamilton


O'Reillys Taschenbibliothek
2. Auflage
UML 2.0
kurz & gut
Dan Pilone
Deutsche Übersetzung
von Dorothea Heymann-Reder
O'REILLY

Wie kommunizieren wir Anforderungen und Designs?

Beispiele (für Notationsarten)

- (textuelle) Programmiersprachen
- mathematische Formeln
- informeller Text
- informelle graphische Darstellungen (z.B. grafische ad-hoc Notation an der Tafel)
- formelle graphische Modellierungssprachen (z.B. UML)

Die Wahl der Notation beeinflusst u.a.

- Effektivität und Effizienz der Kommunikation
- Die Art wie über ein Problem nachgedacht wird und den Problemlösungsvorgang selbst

Cognitive Dimensions Framework

Wie wählt man geeignete Notationen aus?

- "Cognitive Dimensions Framework" (Green & Petre, 1996)
- Bietet Vokabular und Konzepte zur Evaluation von Notationen

Beispiele (für Kognitive Dimensionen)

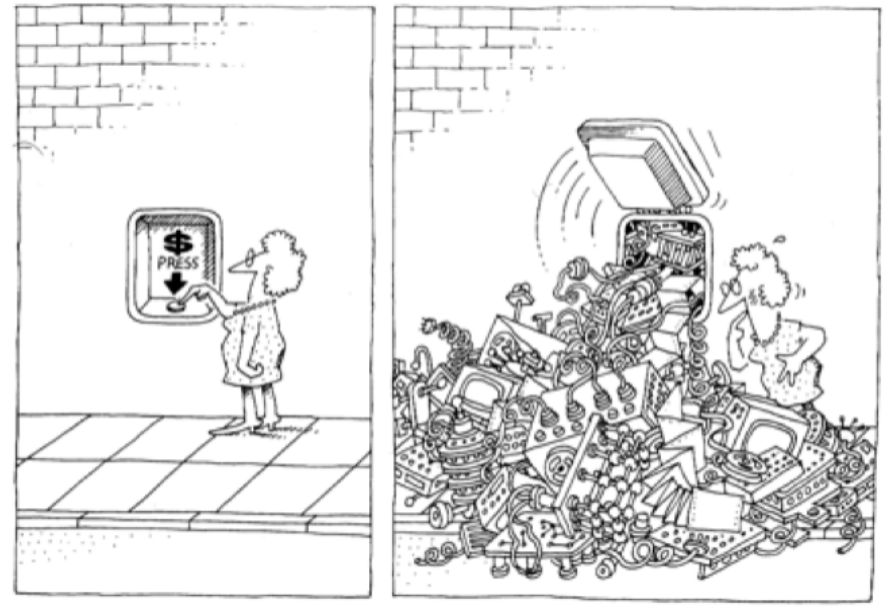
- Abstraction Gradient
- Closeness of Mapping
- Consistency
- Diffuseness
- Error-proneness
- ...

Die Dimensionen können genutzt werden, um einzuschätzen, ob eine Notation für einen gewissen Einsatz geeignet ist.

Abstraction Gradient

Grad, zu dem eine Notation Abstraktion unterstützt

- Gruppierung von Elementen, die als ein Element verstanden / behandelt werden können
- Ermöglicht Details vorrübergehend zu ignorieren und Zusammenhänge zu erkennen

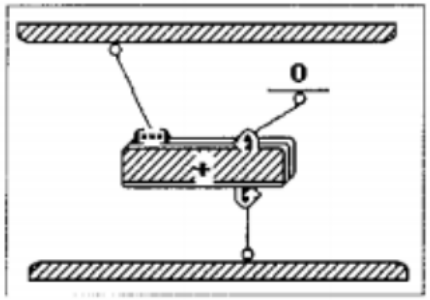


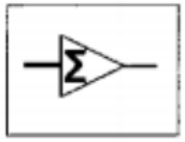
The task of the software development team is to engineer the illusion of simplicity.

Closeness of Mapping

```
sum = 0;
for (i = 0; i < N;  i++)
{
   sum += x[i];
}
```

```
sum = 0
for i = 1 to N
   sum = sum + x(i)
next
```





=SUM(A1..A10)

- Programmieren als “mapping” zwischen Problem Domäne und Programm
- Notation sollte möglichst nah an Problem Domäne sein:
  - minimale Anzahl an lexikalischer Konzepte pro Konzept aus der Problem Domäne
  - Operationen im Programm sollten Operationen aus der Problem Domäne entsprechen

Consistency

- Grad an Widerspruchsfreiheit der einzelnen Notationselemente
- Hier: Wie gut kann ein Nutzer mit partiellem Wissen über die Notation den Rest der Notation erschließen?
- Konsistenz kann innerhalb einer Notation bewertet werden, aber auch zwischen Notationen
- u. A. Motivation für Standards: Ein Standard hilft dem Leser sich auf Bekanntes zu verlassen und den Rest zu erschließen

Diffuseness / Terseness

- Bestimmt durch Anzahl an lexikalischer Notationselemente
- In der Bewertung von Programmiersprachen informell häufig bezeichnet als “Syntactic Noise”
- “Closeness of Mapping” und “Diffuseness” beeinflussen sich gegenseitig

Error-proneness

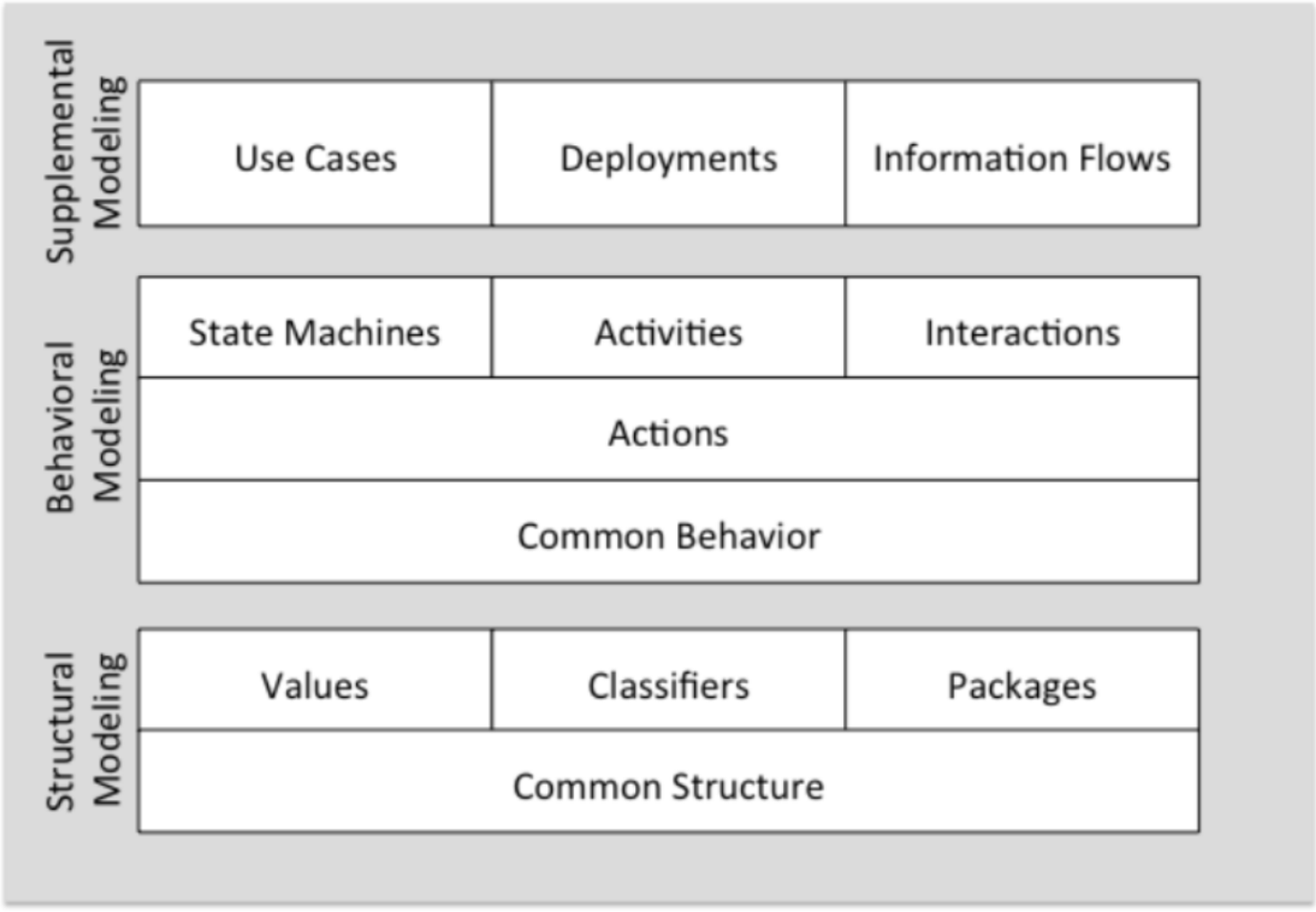
- Beschreibt wie Fehleranfällig / Fehlertolerant die Notation ist
- Fehlerproportionalität: Führen kleine fehlerhafte Änderungen zu kleinen Unterschieden in der Bedeutung?
- Können Fehler einfach identifiziert werden?

Was ist UML?

UML ist eine standardisierte, graphische Modellierungssprache, ausgelegt für die Objekt Orientierte Modellierung von Softwaresystemen.

- **Formal**: Jedes Sprachelement hat eine definierte Bedeutung
- **Umfassend**: UML kann verwendet werden, um die meisten Aspekte eines Systems zu modellieren
- **Präzise**: Die Lexeme / atomaren Elemente der Sprache sind einfache Formen und Symbole
- **Praktisch Motiviert**: UML fasst Erfahrungen aus mehreren Jahrzehnten Modellierungspraxis zusammen
- **Standardisiert**: Offener Standard (794 Seiten in Version 2.5), die wichtigsten Konzepte sind den meisten Entwicklern bekannt

Bedeutungsebenen in UML



Diagrammarten in dieser Vorlesung

Strukturelle Modellierung

- Klassendiagramm (*Class Diagram*)
- Objektdiagramm (*Object Diagram*)

Verhaltensmodellierung

- Interaktionsdiagramm (*Interaction Diagram*, früher *Sequence Diagram*)
- Zustandsdiagramm (*State Diagram*)

Was sind Objekte?

- Philosophie: "(Materieller oder gedachter) Gegenstand der Betrachtung, mit bestimmten **Eigenschaften** (Aussehen, Verhalten) ausgestattet".
- OOP: Wir betrachten keine realen Objekte, sondern nur die Modellierung solcher Objekte.
- Die Modellierung fokussiert auf die **wichtigen (beobachtbaren) Eigenschaften** des Objektes.
- Genau die im Kontext benötigten Eigenschaften werden betrachtet, darüber hinausgehende gibt es nicht.

Graphische Darstellung von Objekten

- Darstellung eines Objekts als **Rechteck**. Name des Objektes wird unterstrichen.

- Objekte sind Instanzen von Klassen. Darstellung erfolgt durch Annotation des Klassennamens nach einem Doppelpunkt.

- Objektnamen können weggelassen werden.

Objekte mit Eigenschaften

Attribute

- Objekteigenschaften heißen "**Attribute**".
- Attribute bestimmen den **Zustand** des Objektes.

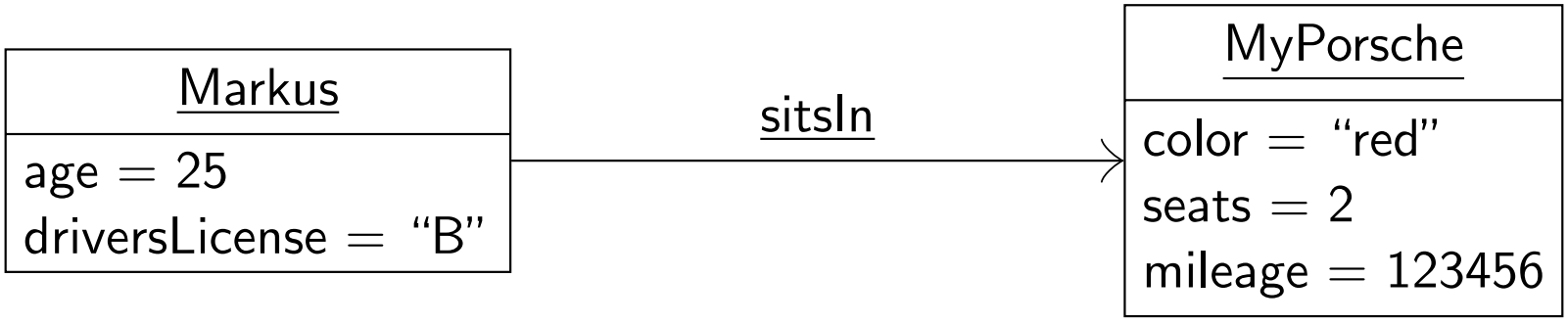
Beispiel

MyPorsche
color = "red"
seats = 2
mileage = 123456
driver = "Markus"

- *driver* ist hier ein (einfaches) Attribut. Wir können den Fahrer selber aber auch als Objekt modellieren.

Assoziationen (Verbindungen) zwischen Objekten

Beispiel



Darstellung

- **Namen** von Assoziationen zwischen Objekten (optional) werden unterstrichen.
- Assoziationen können **gerichtet** sein, wie hier im Beispiel.
- Richtung: A → B bedeutet:
  A "kennt" *B* und kann **Dienste** von *B* aufrufen.

Klassen von Objekten

Definition

- **Klasse** = Menge von Objekten mit denselben **Attributen** und identischem *Verhalten*.
- Das *Verhalten* von Objekten wird durch die bereitgestellten **Operationen** (oder **Dienste**) bestimmt.
- Die von der Klasse beschriebenen Objekte heißen **Instanzen** der Klasse.

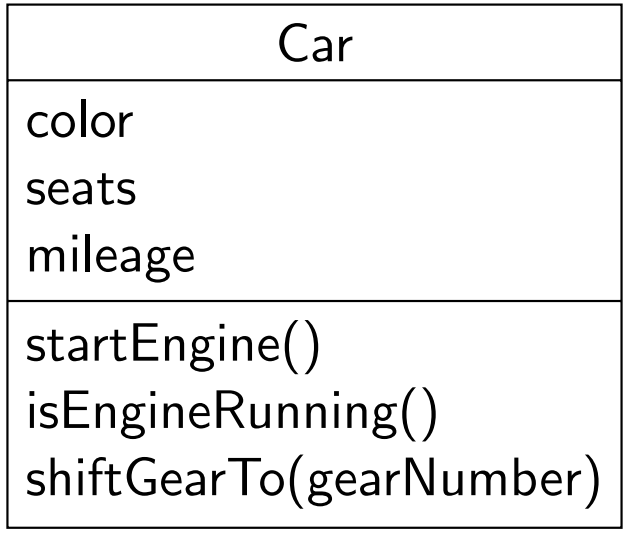
Klassendiagramme

Ein Klassendiagramm

- zeigt die **statische Struktur** eines Systems
- bildet die Grundlage für Objektdiagramme
  - Klassen können zu Objekten instanziiert werden
  - Assoziationen zwischen Klassen können zu konkreten Assoziationen zwischen Objekten instanziiert werden
  - Objektdiagramme als Instanzen von Klassendiagrammen müssen die modellierten Randbedingungen im Klassendiagram (z.B. Multiplizitäten) einhalten

Darstellung von Klassen

Beispiel

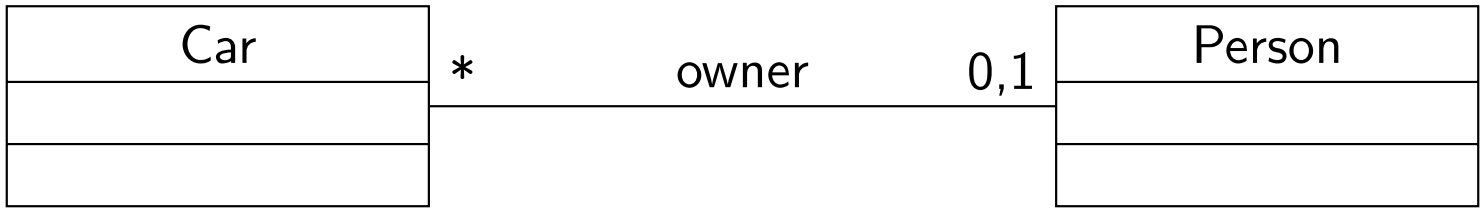


Darstellung von Klassen

- Dreigeteilt: Name, Attribute, Operationen.
- Optional: weitere Angaben für Attribute und Operationen, z.B. Sichtbarkeit (public/private) und Typen. Dazu später mehr.

Assoziationen

Beispiel



- Jedes Auto hat höchstens eine Person als Besitzer.
- Jede Person kann beliebig viele Autos besitzen.

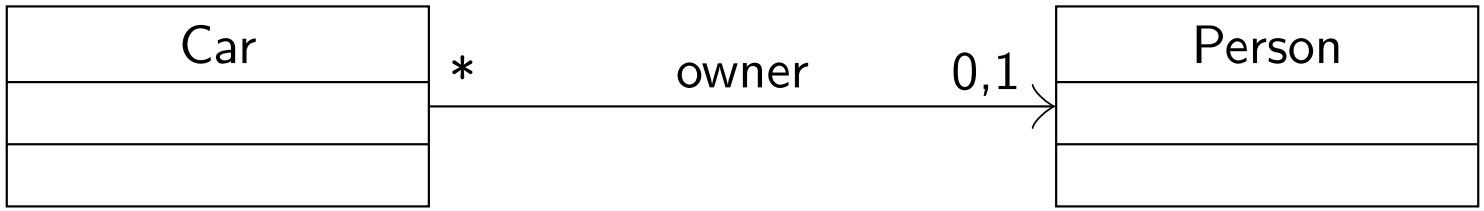
Multiplizitäten

- 1: genau eine verbundene Instanz,
- *: beliebig viele verbundene Instanzen (evtl. *keine*),
- 0,1: höchstens eine verbundene Instanz,
- 1..*: mindestens eine verbundene Instanz.

Gerichtete Assoziationen

Auch Assoziationen können gerichtet sein, sie heißen dann "**navigierbar**".

Beispiel

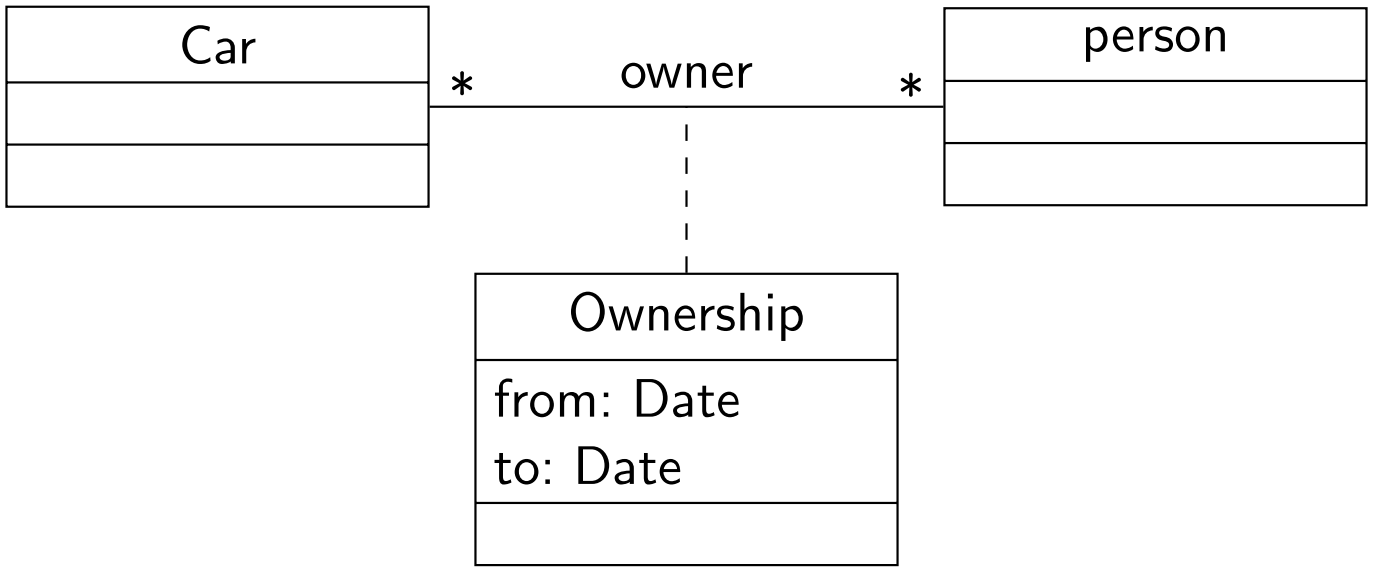


- Bedeutet: zu jedem Auto lässt sich der Besitzer feststellen.
- Impliziert: Die Klasse *Car* enthält eine Operation, die ein Objekt der Klasse *Person* zurückgibt, z.B. `getOwner()`.

Assoziationen mit Attributen

Assoziationen können selbst als Klassen modelliert werden und mit Attributen ausgestattet werden.

Beispiel



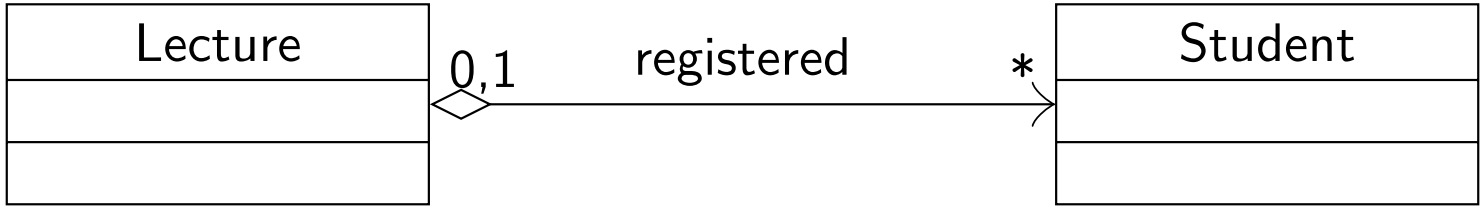
- Einem *Auto* werden nun beliebig viele Besitzer zugeordnet.
- Jeder *Besitz* ist durch ein Zeitintervall (from . . . to) zusätzlich gekennzeichnet.

Aggregation

Hierarchie auf verbundenen Instanzen

- *Teil-Ganzes*-Verhältnis
- *Ganzes*-Instanz hat "Verantwortung" für die *Teil*-Instanzen.
- Zerstören der *Ganzes*-Instanz erfordert Strategie für Umgang mit *Teil*-Instanzen.

Beispiel



- Nicht-ausgefüllte Raute am *Ganzes*-Ende.

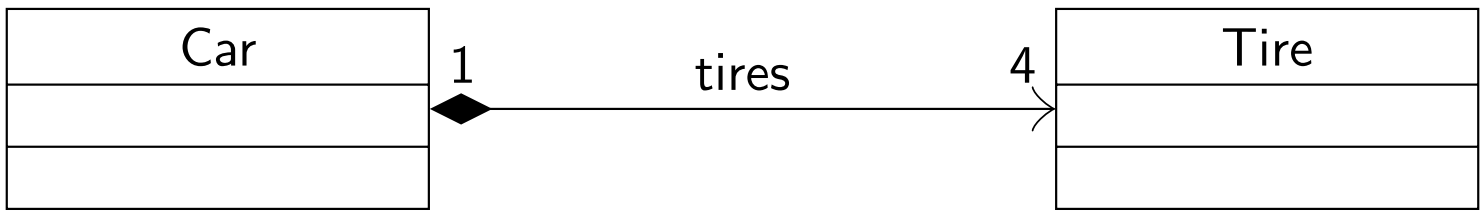
Komposition

Besondere Art der Aggregation

Teil-Instanzen . . .

- dürfen nur von Operationen der *Ganzes*-Klasse entfernt werden
- dürfen nicht Teil anderer Kompositionen sein
- werden beim Zerstören der *Ganzes*-Instanz automatisch (kaskadierend) mit zerstört.

Beispiel



- Ausgefüllte Raute am *Ganzes*-Ende.

Generalisierung

Beziehung zwischen allgemeiner und spezieller Klasse

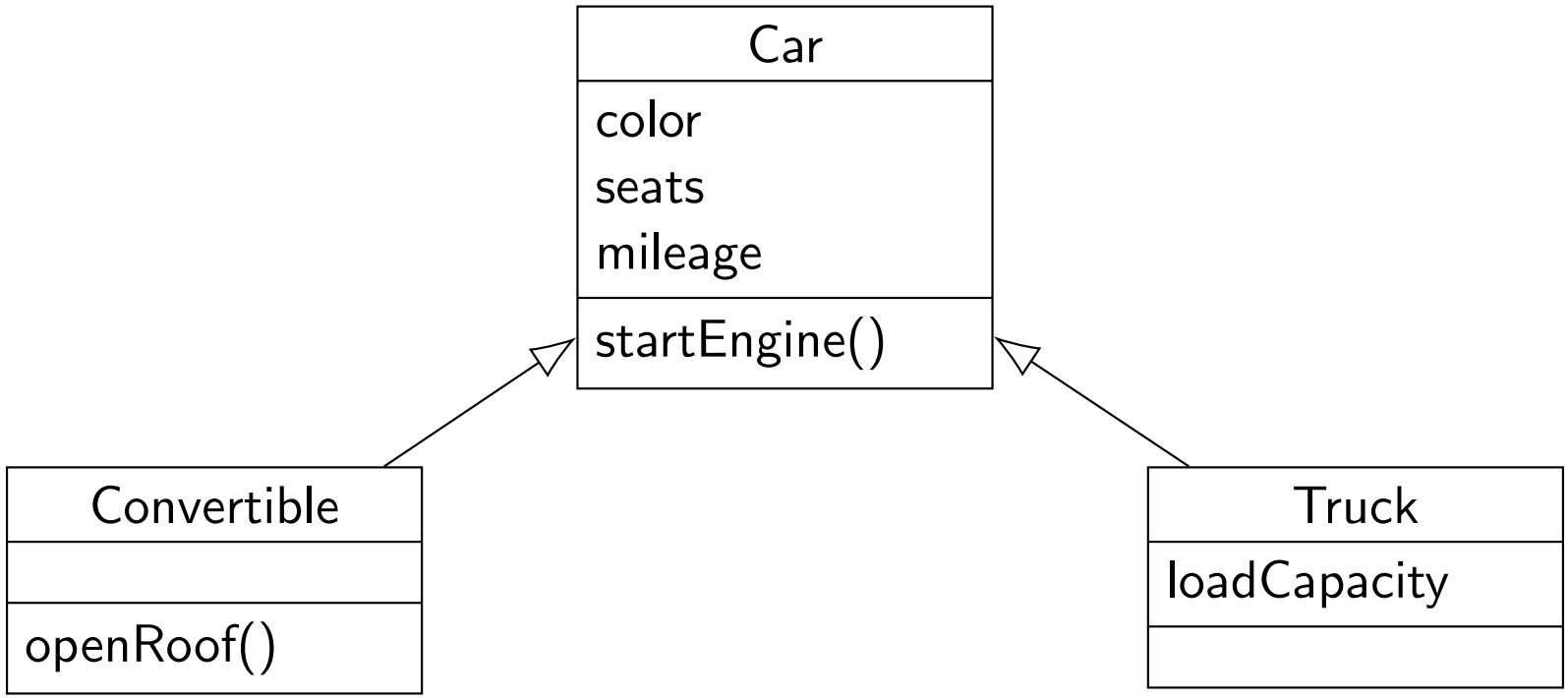
- Alles, was für eine Instanz der allgemeinen Klasse (Oberklasse) zutrifft, gilt auch für die spezielle Klasse (Unterklasse).
- *Alles* = Attribute, Operationen, Beziehungen.

Darstellung

- Nicht-ausgefülltes Dreieck und durchgezogene Linie
- Impliziert häufig in Code:
  `class Unterklasse extends Oberklasse {}`

Generalisierung

Beispiel



Substituierbarkeitsprinzip

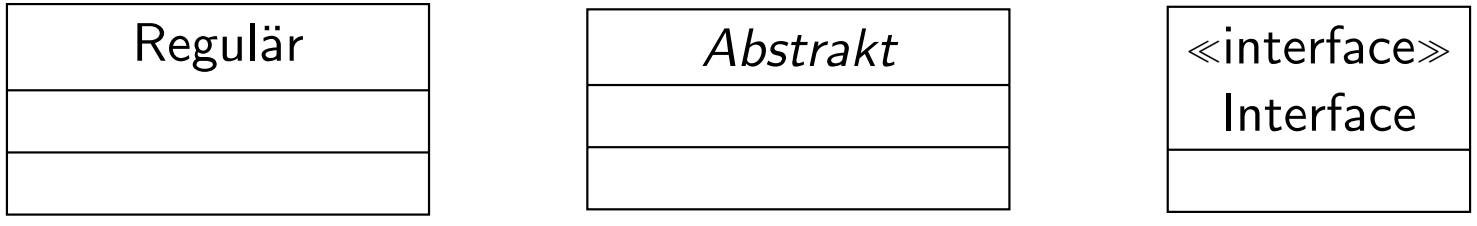
- Jedes Vorkommen einer Instanz der Oberklasse (in Spezifikation, Quelltext oder Programm) kann durch eine Instanz der Unterklasse ersetzt werden.

Abstrakte Klassen und Interfaces

Drei Arten von Klassen

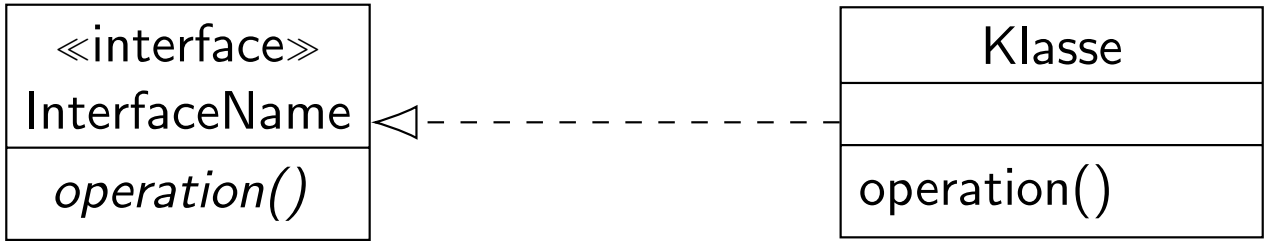
- **Reguläre Klassen:** sämtliche Operationen sind implementiert
- **Abstrakte Klassen:**
  - für mindestens eine Operation wird keine Impl. angegeben
  - haben keine Instanzen
- **Interfaces:**
  - besitzen keine Attribute
  - implementieren keine Operationen
  - sind höchstens Ziel einseitig navigierbarer Assoziationen

Darstellung



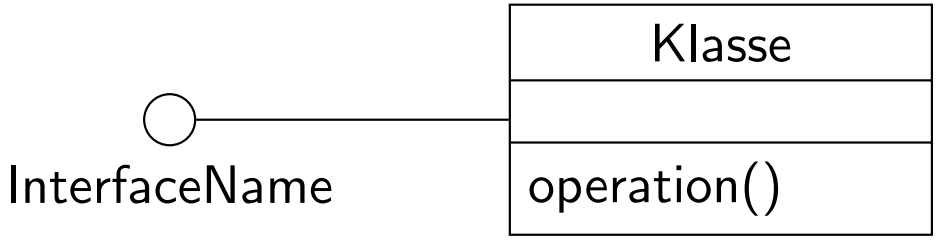
Interfaces

Implementierungsbeziehung



- vgl. Generalisierung: "◁——" (zwischen Klassen).

Alternative (einfache) Darstellung

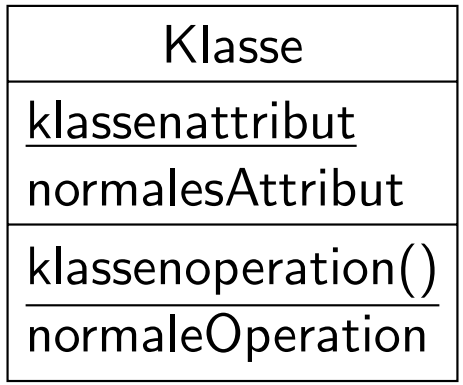


Klassenattribute und -operationen

Semantik

- **Klassenattribut:** Statisches Attribut, zugehörig zur Klasse, nicht zu den Instanzen der Klasse.
- **Klassenoperation:** Statische Operation, welche nicht im Kontext einer bestimmten Instanz, sondern im Kontext der Klasse ausgeführt wird.

Darstellung



- Klassenattribute und -operationen werden unterstrichen.

Sichtbarkeit und Typen

Beispiel

Car
- mileage: int
+ getMileage() : int + shiftGearTo(in gearNumber : int) : void

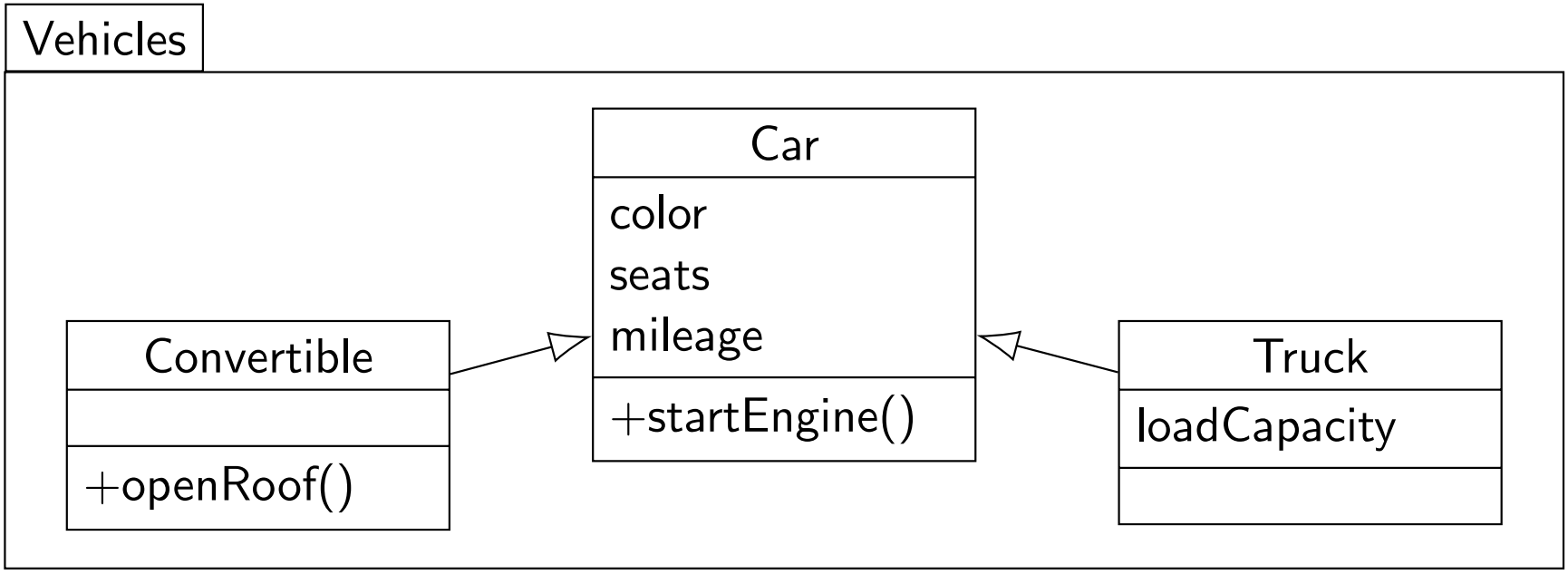
Notation

- **Sichtbarkeit:** public (+), private (-), protected (#), package
- **Übergabeart:** in, out, inout
- **Typen:** primitive Typen und Namen von Klassen und Interfaces

Diese Elemente sind optional und dürfen weggelassen werden.

Pakete

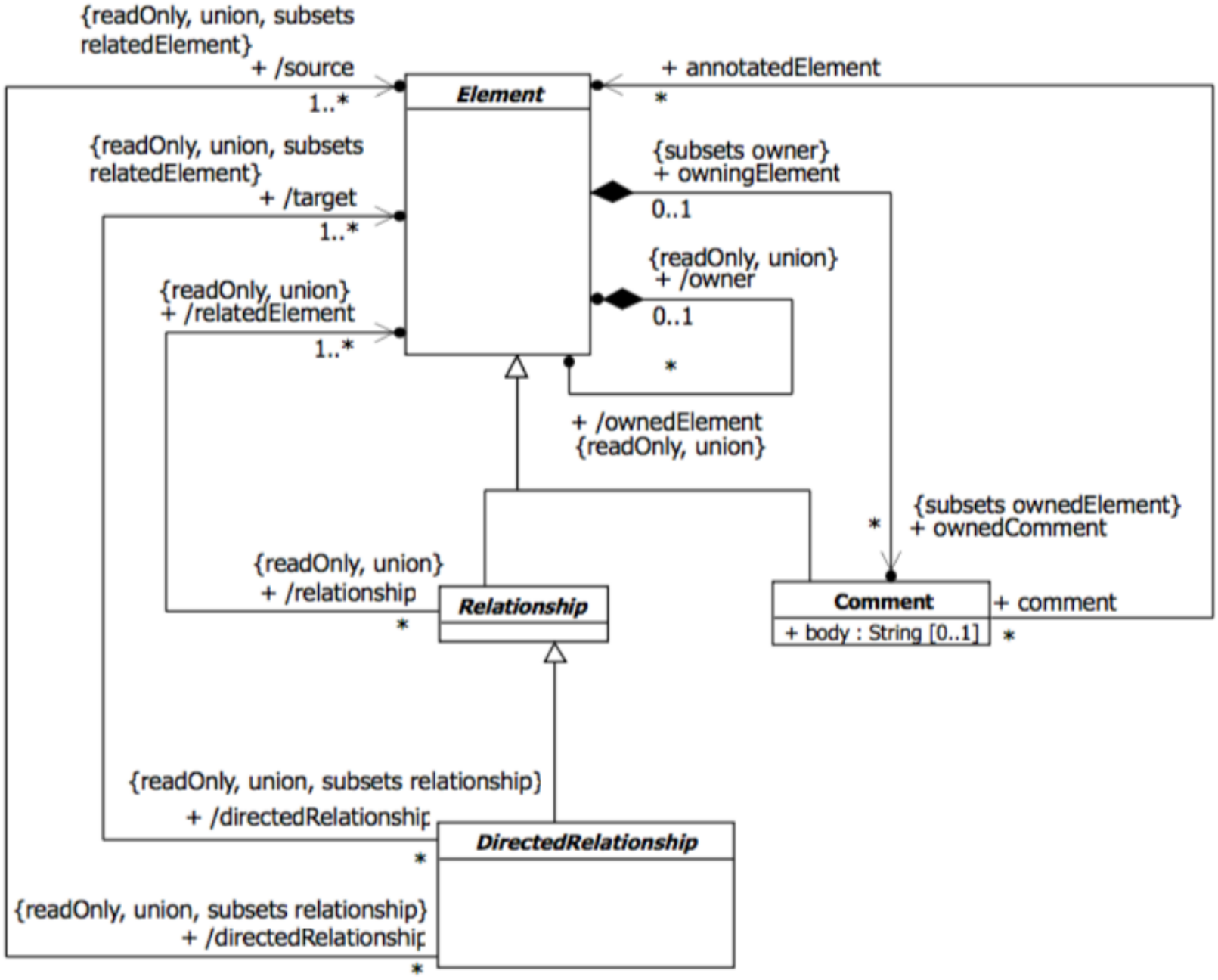
Beispiel



Nutzung

- Gruppierung von Modellierungselementen
- Sichtbarkeitsgrenzen (public $>$ package $>$ private)

Beispiel Klassendiagramm: Abstrakte Syntax von UML



Verhaltensmodellierung

Wir betrachten hier zwei Arten von Diagrammen:

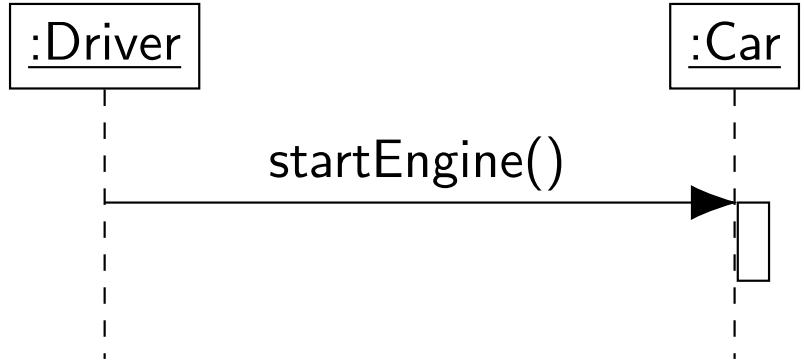
- **Interaktionsdiagramm:** Ablauf von Anwendungsfällen; Austausch von Nachrichten zwischen Objekten.
- **Zustandsdiagramm:** Zustandsorientiertes Verhalten; Zusammenspiel von Zuständen und Operationen.

Bemerkung

- UML kennt noch andere Formen der Verhaltensmodellierung:
  - Anwendungsfalldiagramm
  - Kollaborationsdiagramm

Interaktionsdiagramm

Beispiel

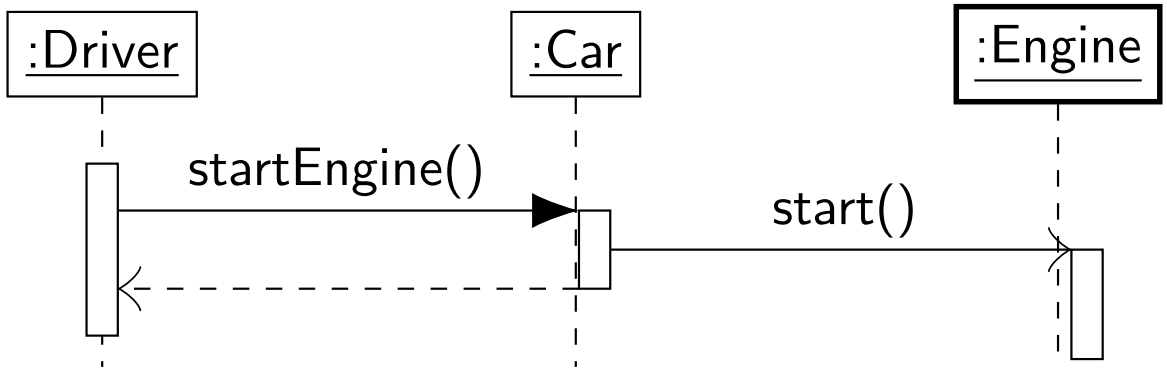


Austausch von Nachrichten zwischen Objekten

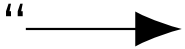
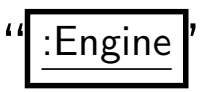
- Objekte und **Lebenslininen** (gestrichelt)
- Zeitachse: von oben nach unten
- :Driver bedeutet: Objekt vom Typ *Driver*
- Pfeil in Richtung der Nachricht
  (*synchrone Nachricht*: ausgefülltes Dreieck)
- **Aktivierungsbalken**: Objekt führt Operation aus

Ablaufkontrolle

Beispiel

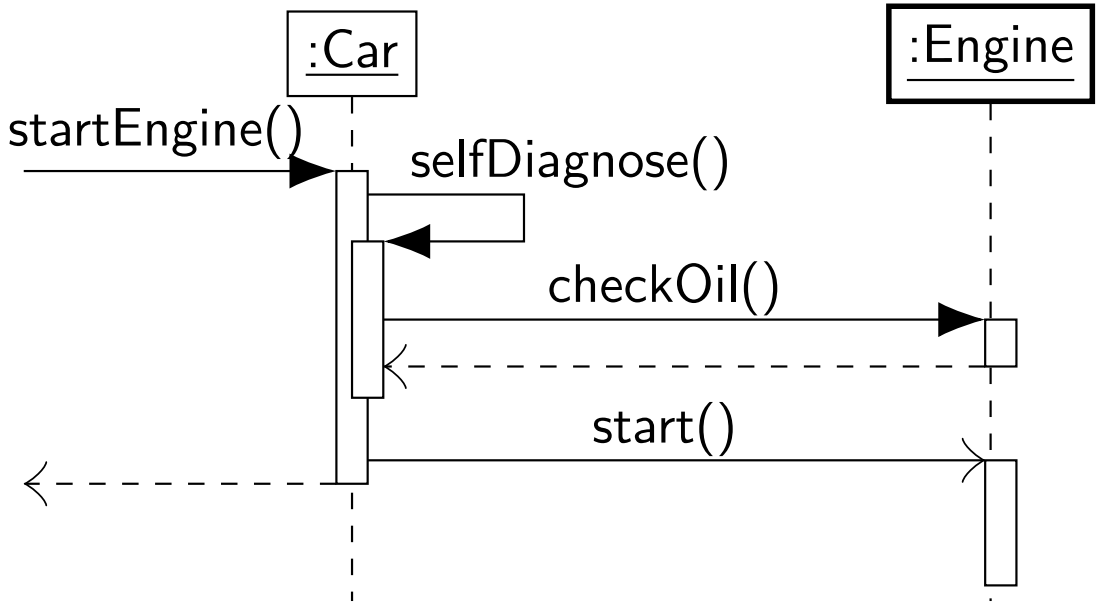


Darstellung

- "  ": **synchrone** Nachricht, Aufrufer wartet auf Rückkehr
- "  ": Rückkehr nach Operationsabschluss
- "": **asynchrone** Nachricht, nur an aktives Objekt
- "  ": **aktives Objekt** (dicker Rahmen), besitzt eigene Ablaufkontrolle *(Thread)*

Selbstdelegation

Beispiel



- Klasse *Auto*: *startEngine()* ruft *selfDiagnose()* auf sich selbst auf.
- Die Operation *selfDiagnose()* kann selbst weitere Operationen aufrufen (hier: *Engine.checkOil()*).

Zustandsdiagramm

Zustand eines Objekts

- Zustand: zu jedem Zeitpunkt gegeben durch Attributwerte und Verbindungen mit anderen Objekten.
- Im Zustandsdiagramm werden aber gewöhnlich viel weniger Zustände unterschieden als es Kombinationen von Attributwerten gibt.

Beispiel

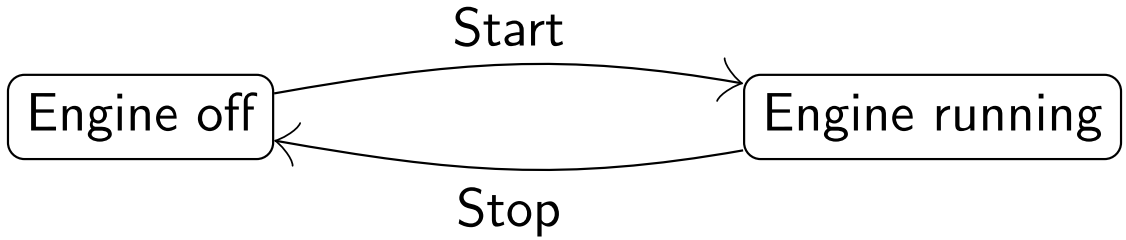
- (Zustand: Wasser ist flüssig) $\stackrel{\wedge}{=}$
  Alle Kombinationen von Attributwerten mit $0 < T < 100$

Modellierung

- Eigenschaften außer acht lassen, die das Verhalten nicht beeinflussen.

Zustände und Übergänge

Beispiel



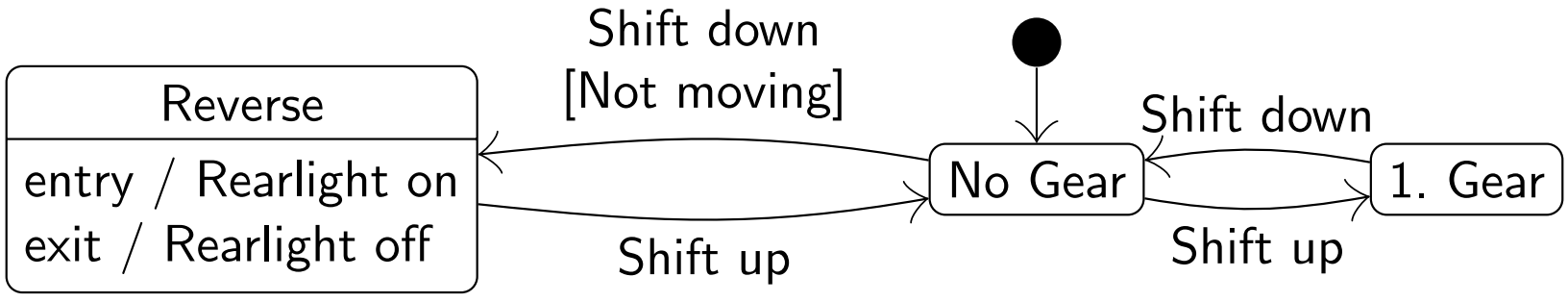
- *Start* und *Stop* bezeichnet **Ereignisse**, die Zustands**übergänge** auslösen.

Arten von Ereignissen

- **Aufrufereignis:** Empfang einer Nachricht ($\rightarrow$ Ausführung einer Operation)
- **Änderungsereignis:** Änderung von *Umgebungsbedingungen*, z.B. Zuständen anderer Objekte
- **Zeitereignis:** z.B. Ablauf eines Timers

Wächter, Entry/Exit-Aktionen, Initial-/Terminalzustand

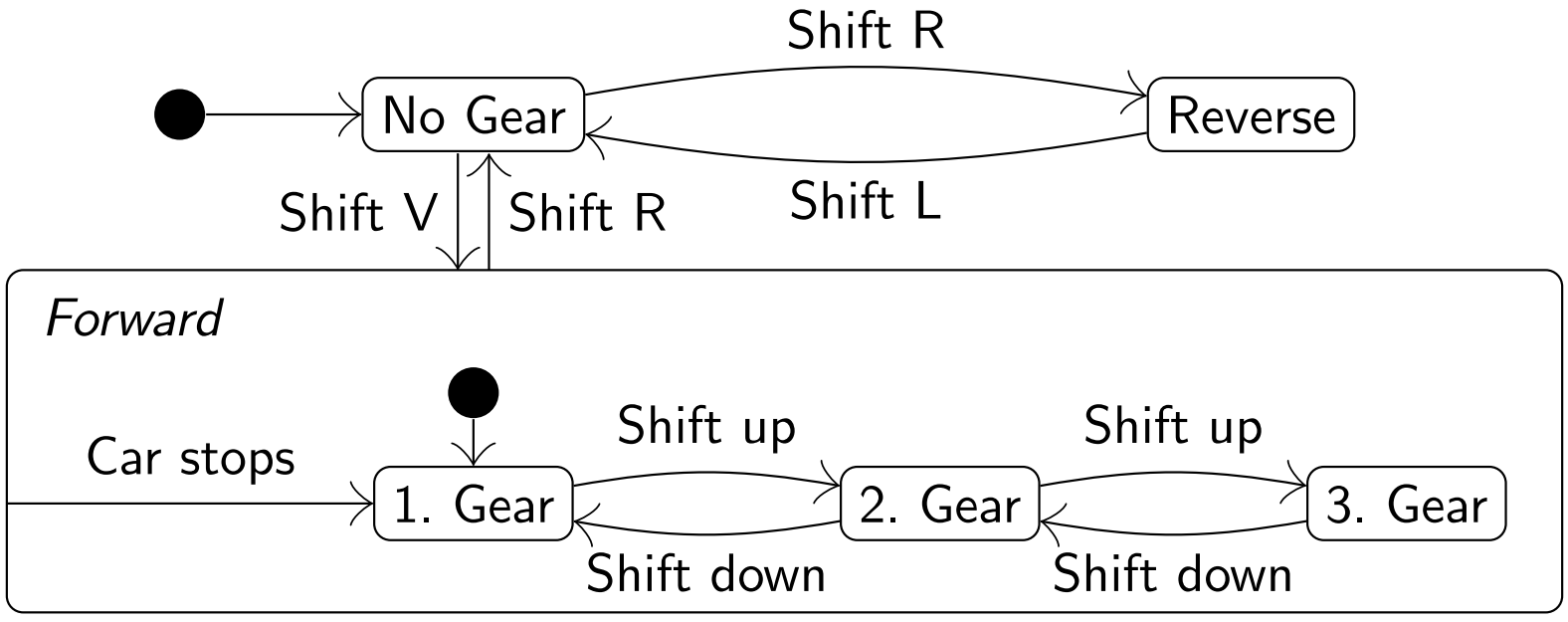
Beispiel (einfaches Getriebe: Nur Hoch- und Runterschalten)



- Schalten in Rückwärtsgang nur bei stehendem Wagen (**Wächterbedingung**)
- **Eingangsaktion** beim Erreichen (entry) und **Ausgangsaktion** beim Verlassen (exit) des Zustands
- ●: **Initialzustand** (keine eingehenden Transitionen), ◉:**Terminalzustand** (keine ausgehenden Transitionen)

Zusammengesetzte Zustände, Unterzustände

Beispiel (automatisches Getriebe)



Forward ist **zusammengesetzter Zustand**. Übergang...

- endet auf Kontur = endet auf Unter-Initialzustand
- geht von Kontur aus = verbindet alle Unterzust. (außer ●)
- von Kontur zu Unterzust.: → bei Eintritt des Ereignisses