

Objektorientierte Programmierung

Kopplung und Kohäsion

Objektorientiertes Design



Roland Gisler



Inhalt

- Kriterien für gutes OO-Design
- Kohäsion
- Kopplung
- Beziehungen zwischen Klassen
- Modellierung / UML
- Zusammenfassung

Lernziele

- Sie kennen die zentralen Ziele des objektorientierten Designs.
- Sie können die Kohäsion einer Klasse beurteilen.
- Sie können die Kopplung zwischen Klassen beurteilen.
- Sie vermögen den Zielkonflikt zwischen hoher Kohäsion und loser Kopplung abzuwägen.

Kriterien für gutes OO-Design

- Es gibt eine Vielzahl von Regeln und Kriterien, welche helfen, ein «gutes» Design zu entwerfen.
 - «Gut» immer im Sinne von: Den jeweils individuellen Anforderungen und dem gegebenen Kontext angemessen.
- Einige Kriterien haben Sie bereits (evtl. unbewusst) kennengelernt.
 - Werden als → Designprinzipien später noch vertieft.
- Zwei sehr fundamentale Kriterien für gutes Design sind:
 - (möglichst) **starke Kohäsion**
 - (möglichst) **lose Kopplung**
- Sehr häufig zitiert und fast mantraartig wiederholt, sind sie zwar relativ leicht verständlich, aber nicht immer leicht einzuhalten bzw. umzusetzen.

Kohäsion

Kohäsion - Begriff

- Lateinisch «cohaesum», «cohaerere»:
Zusammenhang, Zusammenhalt
- Ziel in der Objektorientierung: Eine Programmeinheit (Methode, Klasse, Modul etc.) ist für genau **eine** wohldefinierte Aufgabe verantwortlich.
 - ➔ **S**ingle **R**esponsibility **P**rinzip (SRP)
- Am Beispiel einer Klasse: Die Aufgabe wird durch ein enges Zusammenspiel zwischen Attributen und Methoden erreicht.
 - Attribute und Methoden der Klasse haben einen starken inneren Zusammenhalt, und realisieren damit auch ➔ Datenkapselung.
- Wenn eine Klasse nicht sinnvoll weiter teilbar ist:
 - ➔ Optimale (starke) Kohäsion erreicht.



Kohäsion – **Schlechtes** Beispiel

- Betrachten Sie die folgende Klasse:

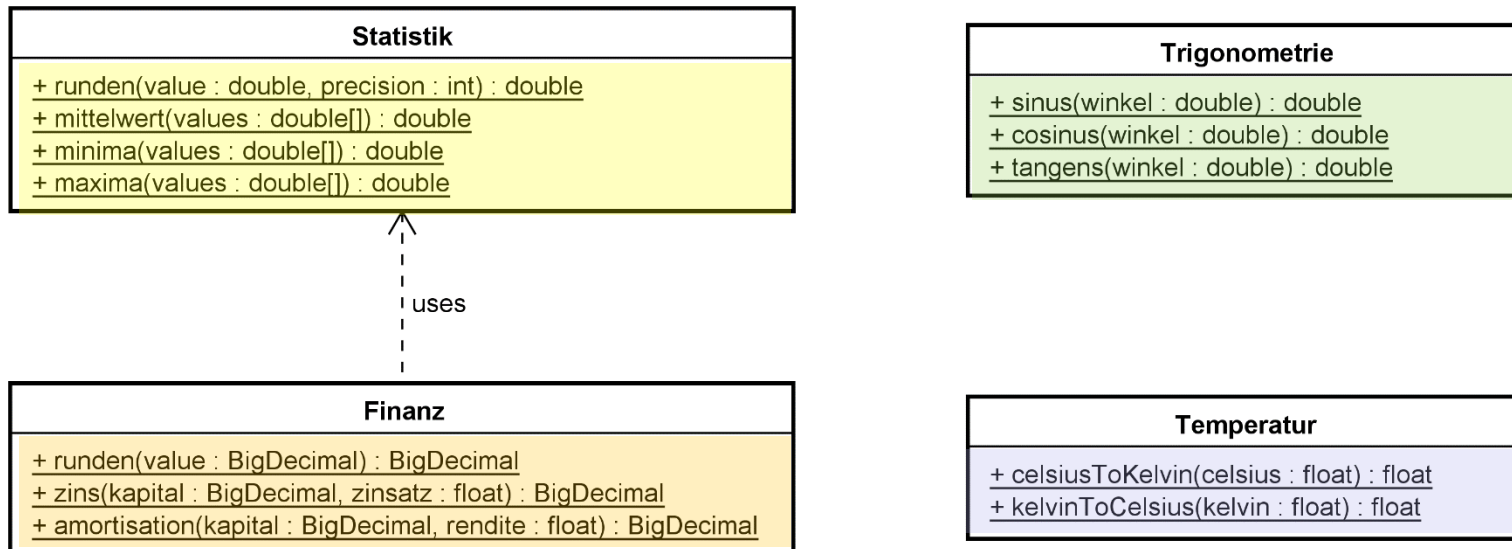
CalcUtil
+ runden(value : double, precision : int) : double
+ rundenGeld(value : double) : double
+ sinus(winkel : double) : double
+ cosinus(winkel : double) : double
+ tangens(winkel : double) : double
+ mittelwert(values : double[]) : double
+ minima(values : double[]) : double
+ maxima(values : double[]) : double
+ celsiusToKelvin(celsius : double) : double
+ kelvinToCelsius(kelvin : double) : double
+ zins(kapital : double, zinsatz : float) : double
+ amortisation(kapital : double, rendite : double) : double



- Schon der Name **CalcUtil** lässt es ahnen: Ein Gemischtwarenladen von Methoden, die nicht wirklich zusammengehören!
 - Hat vielleicht mal (gut und nur) mit **runden(...)** angefangen?
- ➔ Schlechte, tiefe, **schwache Kohäsion**; SRP verletzt.

Kohäsion – Gutes (refaktoriertes) Beispiel

- Die Klasse wurde in vier einzelne Klassen aufgebrochen:



- Positiver Effekt: Namensgebung wurde spezifischer und besser!
 - Achtung: Jede weitere, neue Methode, die ergänzt wird, kann ein erneutes Aufbrechen sinnvoll bzw. notwendig machen!
- ➔ Refactoring ist ein **ständiger** Prozess.

Vorteile von Klassen mit hoher Kohäsion

- Das **S**ingle **R**esponsibility **P**inciple (SRP) wird eingehalten.
 - Nur eine einzige Verantwortlichkeit, nur eine Aufgabe.

Klassen die sich an **SRP** halten sind/werden dadurch:

- Kleiner und spezifischer.
- Schneller und leichter überschaubar / verständlicher.
- Viel besser wiederverwendbar.
- Wesentlich einfacher testbar.
- Seltener von Änderungen/Anpassungen betroffen.
- Besser geeignet für parallele Entwicklung.
- Werden nicht zu «hotspots».



Kopplung

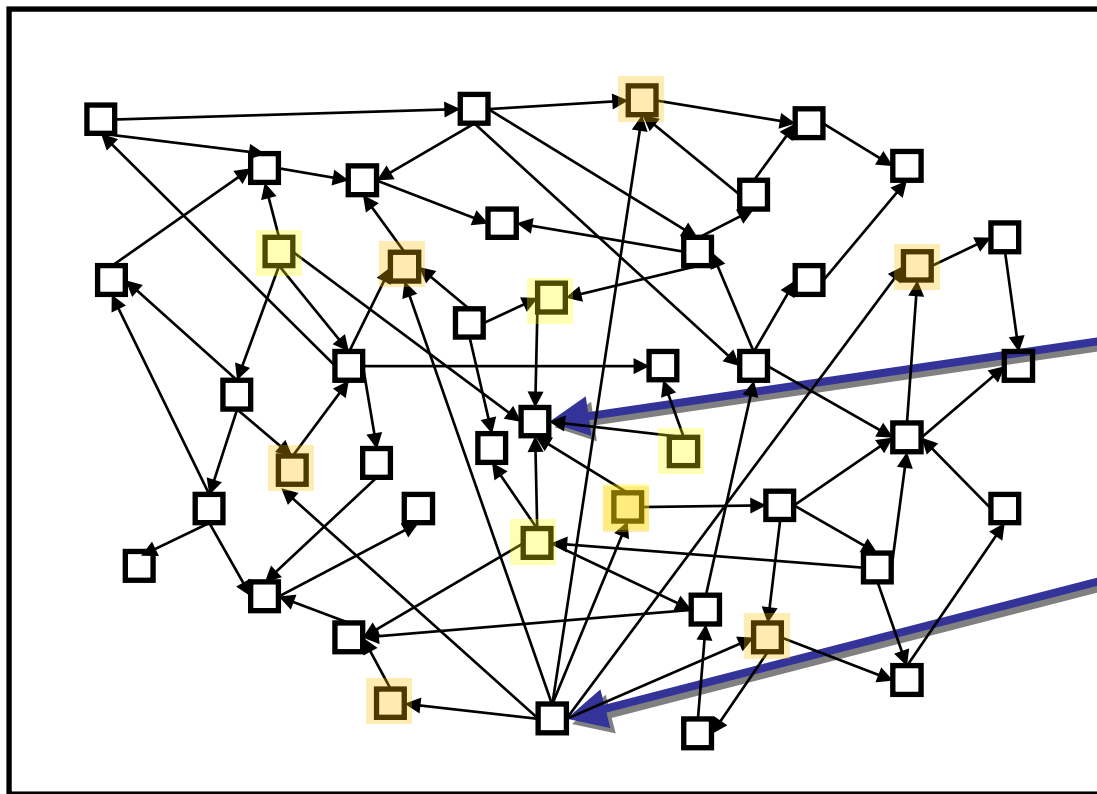
Kopplung - Begriff

- Kopplung beschreibt, wie **stark** einzelne Programmeinheiten (typ. Klasse, Modul, System) untereinander **in Beziehung** stehen.
- Kopplung ist grundsätzlich notwendig, damit überhaupt eine Interaktion (Zusammenarbeit) zwischen Einheiten erfolgen kann.
- Die Kopplung manifestiert sich auf verschiedene Arten:
 - Verwendung (Referenzierung) von Datentypen.
 - Aufruf von Methoden.
 - Explizite Beziehungen zwischen den Einheiten (Modellierung).
- Die Beurteilung der Kopplung ist auch abhängig von der Klasse selber (Art und Herkunft) → z.B. Library vs. Applikationsspezifisch.

**Ziel: So starke Kopplung wie nötig,
so lose Kopplung wie möglich!**

Beispiel - **Schlechtes** OO-Design

- Ist aus vielen, kleinen Einheiten (Klassen) aufgebaut. 😊
 - Kann ein Indiz für hohe Kohäsion der Einzelklassen sein.
- Ist aber extrem stark vernetzt, ohne sichtbare Struktur. ☹️
 - Direkter Hinweis auf eine (viel) zu hohe Kopplung.



Beispiel 1: Klasse wird von vielen anderen Klassen verwendet.

Beispiel 2: Klasse ist von vielen anderen Klassen abhängig.

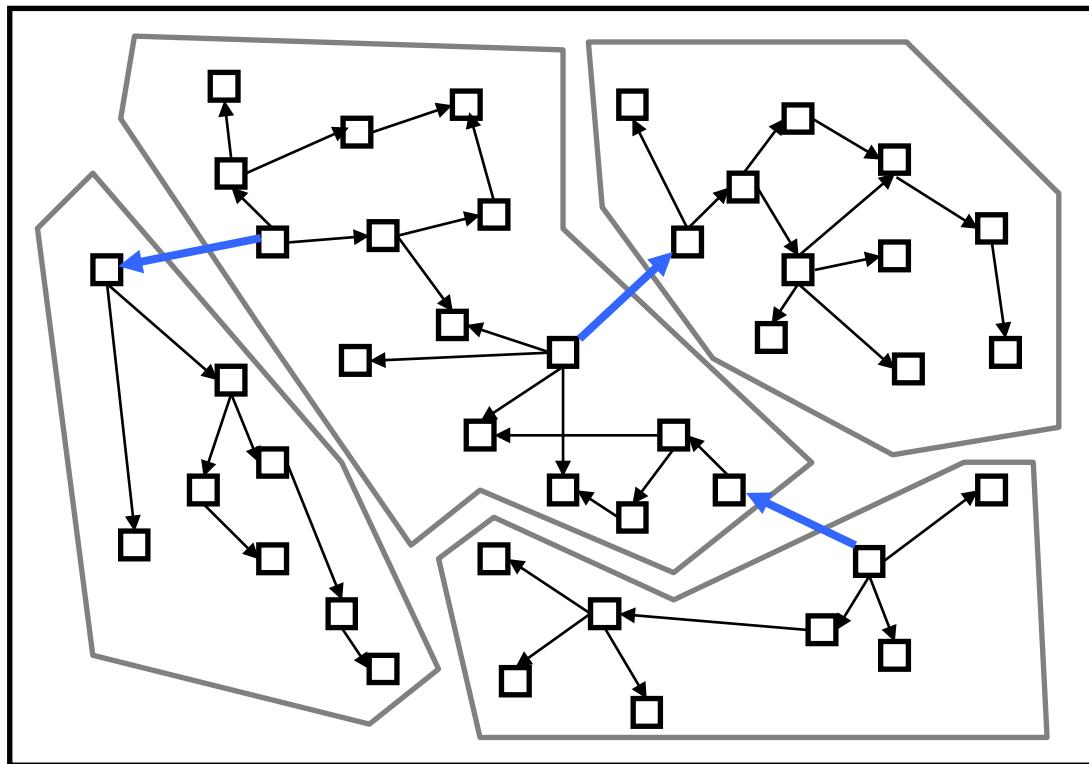
abhängig von →

Schlechtes OO-Design - Konsequenzen

- Funktionsweise ist schwer durchschaubar und schwer verständlich.
Das führt zu:
 - Grundsätzlich langsamere und aufwändigere Entwicklung.
 - Die Fehlersuche ist wesentlich schwieriger.
 - Schlechte Testbarkeit (speziell Unit-Testing).
 - ➔ Konsequenz: Führt meist zu deutlich mehr Fehlern!
- Weiterentwicklung und Wartung wird dadurch erschwert.
 - Spätere Änderungen und/oder Erweiterungen sind schwierig.
 - Schlimmster Fall: Entwicklung wird abgebrochen!

Besseres OO-Design - mit loser Kopplung

- Klassen gruppiert, hohe Kohäsion innerhalb der **grauen** Einheiten*.
 - Beziehungen sind zwischen Klassen grundsätzlich reduziert!
- **Schmale Schnittstellen** zwischen den **grauen** Einheiten*.
- **Graue** Einheiten führen implizit zu einer ➔ **Modularisierung**!



Man beachte die Struktur innerhalb der **grauen** Einheiten! Gibt es evtl. noch mehr Potential?

abhängig von
→

Kopplung und Kohäsion - Erkenntnis

Gutes OO-Design - Auswirkungen

- **Tiefe** Kopplung und **hohe** Kohäsion verbessern das Design:
Es führt zu **mehr** und eher **kleineren** Klassen.
- Daraus resultiert:
 - Bessere und schnellere Verständlichkeit / Lesbarkeit.
 - Einfachere und bessere Testbarkeit.
 - Höhere Wiederverwendbarkeit (auch in anderem Kontext).
 - Höhere Parallelität in der Entwicklung möglich.
- Änderungen sind einfacher möglich:
 - Weil lokaler, betreffen weniger andere Elemente.
 - Auswirkungen von Fehlern schneller eingrenzbar.
 - Somit tendenziell auch weniger Fehler!

Gutes OO-Design wirkt sich unmittelbar und umfassend positiv auf die Software-Qualität aus.

Beziehungen zwischen Klassen


Beziehungsarten zwischen Klassen

- Es gibt verschiedene Arten von Beziehungen, die sich im Quellcode auch unterschiedlich (und nicht immer eindeutig) manifestieren:
 - **Lokale Variable** von einem bestimmten Typ.
 - **Formaler Parameter** von einem bestimmten Typ.
 - **Rückgabewert** von einem bestimmten Typ.
 - **Attribut** von einem bestimmten Typ.
 - eine Klasse **implementiert** eine **Schnittstelle**.
 - eine Klasse **erbt** von einer anderen **Klasse** (Spezialisierung).
- Zusätzlich unterscheidet man auch nach der Herkunft der Typen:
 - aus den Java Libraries (z.B. `java.lang.*`, `java.util.*`).
 - aus dem **eigenen** (selben) Projekt (aber anderes Package).
 - Klassen aus einem anderen (aber eigenen) Projekt.
 - Klassen aus einem externem Projekt (Thirdparty).

Kopplung nimmt zu

Kopplung nimmt zu

Beziehungstyp: use / depends

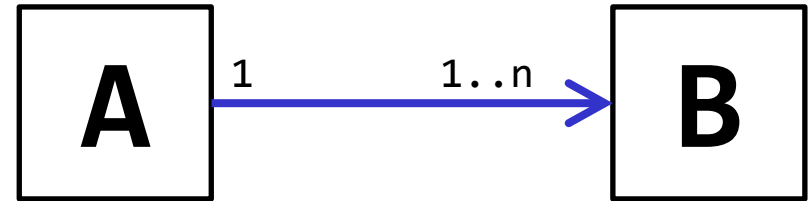
- Die **schwächste** Form einer Beziehung.
- Eine Klasse «nutzt» einen anderen Typ.
 - Meist nur temporal z.B. in einer oder wenigen Methoden.
- Manifestation im Code (Beispiele):
Lokale Variable, formaler Parameter, Returntyp, statischer Aufruf.
- Visualisierung in UML:
A hängt von B ab,
weil B von A verwendet wird.
- Beispiel:
Verwendung von `Math.PI` zur Berechnung des Kreisumfanges.

Beziehungstyp: Assoziation

- Die **häufigste** Form einer Beziehung.
- Eine Klasse hat eine **explizite** Beziehung zu einem anderen Typ.
 - Beziehung ist essenziell für die Funktion / das Modell.

- Manifestation im Code:
Häufig ein Attribut/Referenz von einem anderen Typ (B).

- Visualisierung in UML:
A kennt (und nutzt) B.
(auch bidirektional möglich)



- Beispiel: Ein Student A besucht (verwendet) ein Modul B.
- Empfehlung: Bei Modellen immer sinnvolle Kardinalitäten angeben.
 - Beispiel: Ein A verwendet **ein oder mehrere** (1..n) B.

Beziehungstyp: Aggregation

- Eine Spezialisierung der Assoziation: **Hierarchisch**, «ist Teil von».
- Eine Klasse hat eine explizite Beziehung zu einem anderen Typ.
 - Dieser Typ ist ein logischer Teil (part of) der nutzenden Klasse.

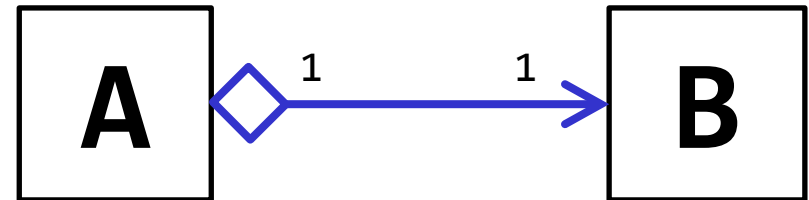
- Manifestation im Code:

Wie Assoziation: Meistens ein Attribut vom Typ des Teiles (B).

- Visualisierung in UML:

A hat (und nutzt) B.

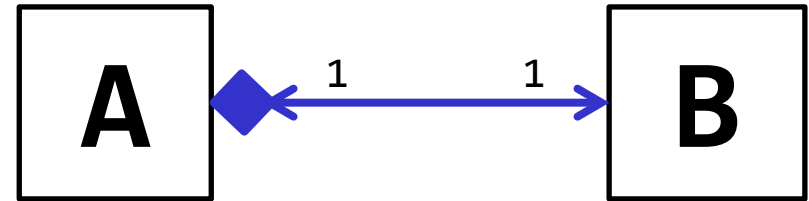
(auch bidirektional möglich).



- Beispiel: Ein Auto **A** **hat** (genau) **einen** Motor **B**.
- Pfeilrichtung drückt die Navigierbarkeit aus:
A kennt «sein» B, aber B weiss nicht wer sein A ist.

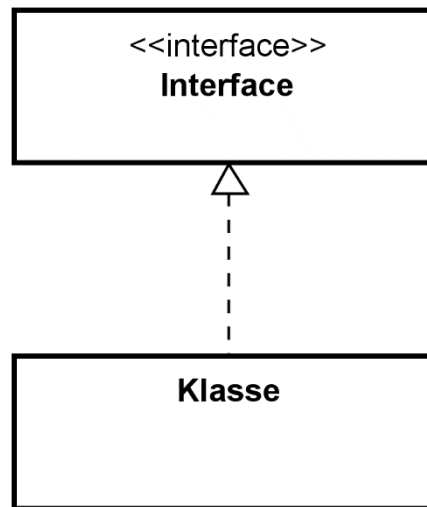
Beziehungstyp: Komposition

- Verstärkung der Aggregation: Teil ist **existenziell** gekoppelt.
- Eine Klasse nutzt einen anderen Typ als Teil von sich selber.
 - Dieser Teil ist **existenziell** (an die Lebensdauer) gekoppelt.
- Manifestation im Code:
Wie Aggregation, aber zusätzlich erstellt (**new**) die Klasse das Teil.
- Visualisierung in UML:
A erzeugt und hat B.
(auch mehrere möglich).
- Beispiel: Ein Mensch A hat einen Kopf B.
 - Wobei hier sowohl A sein B, als auch B sein A kennt.
 - Ein Mensch ist ohne Kopf nicht lebensfähig (und umgekehrt).



Beziehungstyp: Implementation (**implements**)

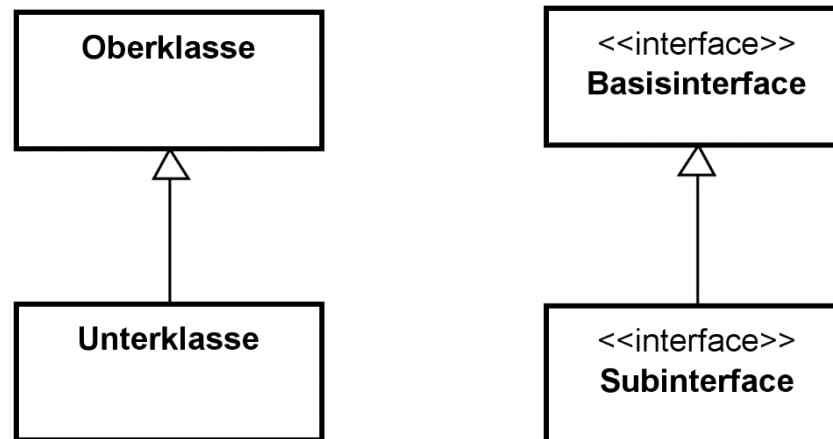
- Eine Klasse Implementiert ein (oder auch mehrere) Interfaces.
- Manifestation im Code: Schlüsselwort **implements**.
- Visualisierung in UML:



➔ Details dazu siehe Input O06_IP_Schnittstellen.

Beziehungstyp: Vererbung (**extends**)

- Eine Klasse erbt von einer anderen Klasse.
 - Analog auch (mehrfach) zwischen Interfaces möglich.
 - Das ist die **stärkste** Form der Kopplung!
- Manifestation im Code: Schlüsselwort **extends**
- Visualisierung in UML:



➔ Details dazu siehe Input O07_IP_Vererbung.

Modellierung und UML

Modellierungssichten und Abstraktionsebenen

		Modellierungssichten		
		<i>System</i>	<i>Prozess</i>	<i>Information</i>
Abstraktionsebene	<i>Fachliches Konzept (WAS)</i>	Requirements/Features Kontext Diagramm Use Case Diagram	Prozesslandkarte & strategisches Prozessmodell	Fachliches Informations-Modell
	<i>Technisches Konzept (WIE)</i>	Architektur- und Komponenten-Diagramm, Service-Integration Technische Spezifikation	Operatives Prozessmodell, Case Management Modell	Datenbank-Modell (Schema für dedizierte DB, Tabellen etc.)
	<i>Konkrete Realisierung</i>	Verteilungsdiagramm Klassendiagramm Sequenzdiagramm auf Programmirebene	Technisches Prozessmodell, Betriebsorganisation	Vollständiges physikalisches DB- Schema (inkl. Optimierungen & Keys)

Wichtiger Hinweis: Die Zelleninhalte sind «nur» **Beispiele** für typische Artefakte der betreffenden Ebene/Sicht.

UML – Unified Modeling Language

- «Vereinheitlichte Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Softwareteilen und anderen Systemen.» (Definition nach **O**bject **M**anagement **G**roup, OMG)
- Einfach formuliert: Eine Art «Esperanto»
Weltsprache der objektorientierten SW-Modellierung!
- Enthält eine Vielzahl verschiedener Diagramm- und Visualisierungsarten:
 - Klassendiagramme, Sequenzdiagramme, Aktivitätsdiagramme, Use Case Diagramme u.v.m.
- Aktueller Stand: Version 2.5.1
- Mehr dazu im Modul «Modellieren Basics».



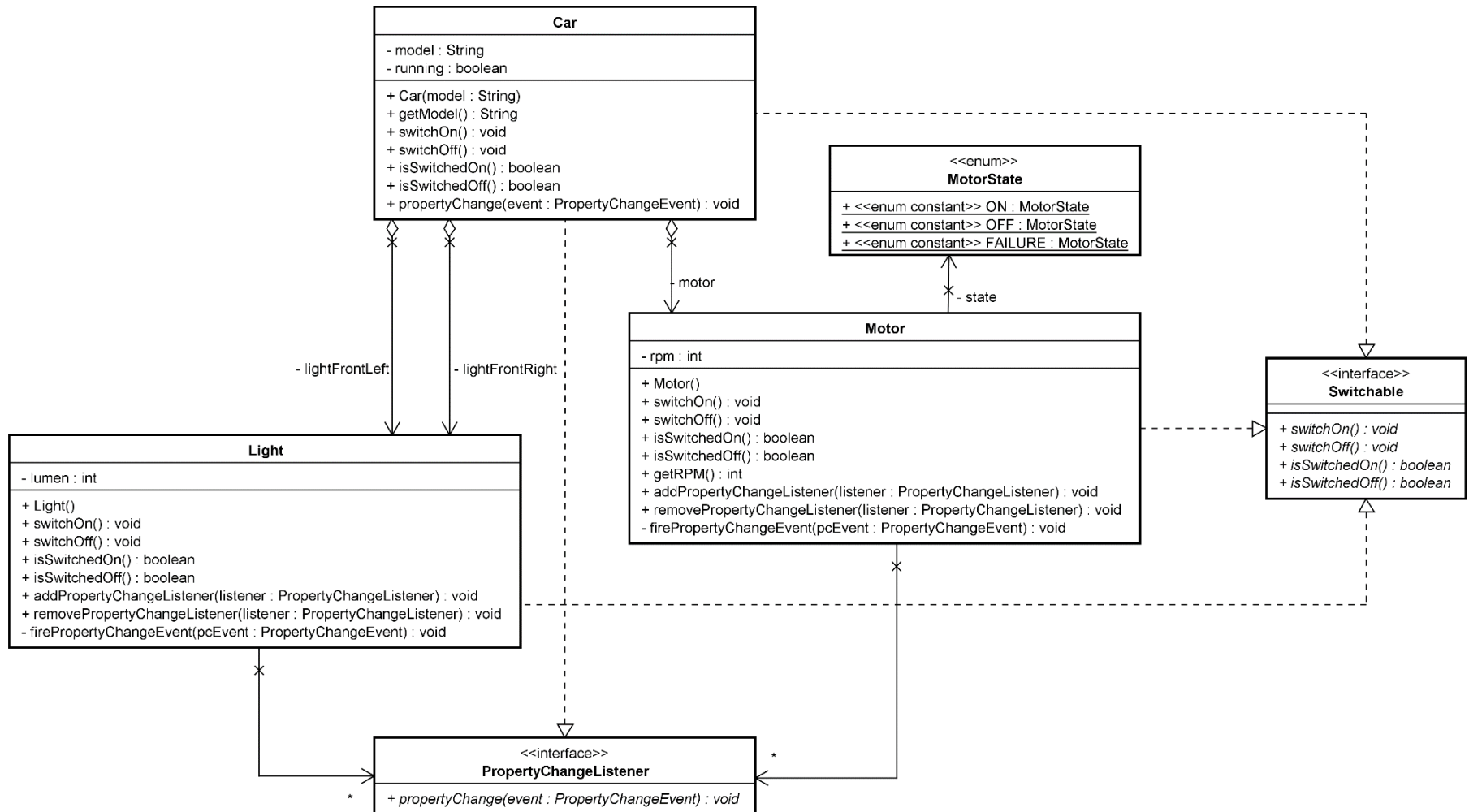
UML - Werkzeuge

- UML lässt sich sehr gut mit Bleistift und Papier zeichnen!
- Eine spontane (Hand-)Skizze kann das Verständnis für die Zusammenhänge sehr schnell vergrössern.
- Es gibt eine Reihe von (komplexen und teuren) Tools.
- Meine persönliche Empfehlung:
astah Professional - <http://astah.net/>
- Seit Herbst 2017 hat die HSLU I eine «faculty site license».
- Steht auf SWITCHdrive (./50_astha_UML) zur Verfügung:
<http://bit.ly/2OH3Uhh>

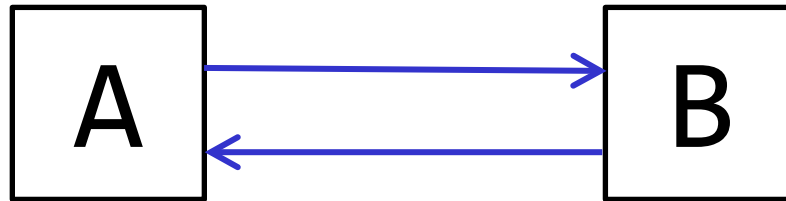


Klassendiagramm nach UML – Beispiel 1

- Studieren Sie die Beziehungen und beurteilen Sie die Kopplung!



Beurteilen Sie die Kopplung – Beispiel 2



Empfehlungen



- Immer das **Single Responsibility Principle** (SRP) einhalten.
 - Eine Klasse soll nur eine Aufgabe haben → **hohe Kohäsion.**
- Tendenziell kleinere (und dafür mehr) Klassen entwerfen.
 - Vorteile: Verständlichkeit, Testbarkeit, Wiederverwendung...
- Im Sinne einer möglichst **losen Kopplung**:
 - Dem Nutzer einer Klasse/Schnittstelle/Methode sollten möglichst wenig weitere Typen aufgedrängt werden!
 - Beziehungen möglichst «schwach» ausprägen.
 - Beziehungsketten: Sind so stark wie das schwächste Glied.
- Grundsatz:
«**Eine so starke Kohäsion wie möglich,
eine so lose Kopplung wie möglich!**»

Zusammenfassung

- Begriff der Kohäsion:
Innerer Zusammenhalt von Softwareeinheiten.
- Begriff der Kopplung:
Menge, Art und Stärke der Beziehungen zwischen verschiedenen Softwareeinheiten.
- Fundamentales Ziel im objektorientierten Design:
Möglichst **hohe** Kohäsion und **lose** Kopplung.
- Beurteilung der Kopplung bedarf der Analyse, welche Art von Beziehung zwischen welcher Art von Softwareeinheiten herrscht!
 - Einfaches «Zählen» führt nicht zum Ziel!
 - Ohne Kopplung geht es nicht.
 - Visualisierung z.B. mit Klassendiagrammen nach UML.

Fragen?



Fragen bitte im
[ILIAS-Forum](#)

