

Programmierkonzepte in den Naturwissenschaften

12. Objektorientierte Programmierung (OOP)

PD Dr. Till Biskup

Physikalische Chemie und Didaktik der Chemie

Universität des Saarlandes

Sommersemester 2020





- 🔑 OOP ist eine Abstraktion, die die Abbildung realer Probleme auf Code stark vereinfachen kann.
- 🔑 OOP erfordert ein grundlegendes Umdenken verglichen mit anderen Programmierparadigmen.
- 🔑 Richtig eingesetzt erleichtert OOP die Erstellung modularer, robuster und wiederverwendbarer Codes.
- 🔑 Objektorientierte und strukturierte Programmierung schließen sich nicht gegenseitig aus.
- 🔑 Die Kernaspekte von OOP sind Kapselung, Vererbung und Polymorphie.

Motivation: Warum objektorientiert programmieren?

Paradigmenwechsel: strukturiert zu objektorientiert

Grundlagen: Konzepte der objektorientierten Programmierung

Ausblick: objektorientiertes Design

Warum objektorientiert programmieren?

Ein paar Gründe



- ▶ intellektuelle Beherrschbarkeit komplexer Fragestellungen
 - Motivation hinter (fast) allen Programmierkonzepten
 - durch Einschränkung der Freiheiten des Programmierers
 - ▶ Abstraktion
 - OOP erleichtert die Abbildung der Realität auf Code.
 - Konzept des Objektes realitätsnah und sehr mächtig
 - ▶ modularer, robuster, wiederverwendbarer Code
 - Modularität inhärent in OOP
 - Wiederverwendbarkeit durch zentrale Konzepte der OOP
 - ▶ zentrale Konzepte leicht umsetzbar
 - automatisierte Tests (Unittests)
 - Modularität und Wiederverwendbarkeit
- 👉 hier: um wissenschaftlichen Ansprüchen zu genügen

These

Objektorientierte Programmierung erfordert eine *grundlegend andere Art zu denken* als rein strukturierte Programmierung.

- ▶ Ein grundlegendes Verständnis der OOP-Konzepte ist Voraussetzung für einen gewinnbringenden Einsatz.
- ▶ Details der Implementierung können unterschiedlich sein, die grundlegenden Konzepte sind (fast) immer identisch.
- ☛ Nachfolgend sollen die Grundkonzepte vorgestellt werden.
- ☛ Details je nach Programmiersprache unterschiedlich

▶ Objekt

- Eigenschaften (Daten, Variablen) und Verhalten (Funktionen) bilden eine untrennbare Einheit.
- vertrautes Konzept aus dem täglichen Erleben
- ermöglicht eine wirkmächtige Abstraktion: direkt auf reale Probleme und Fragestellungen anwendbar

▶ Kapselung

- Nur das Objekt selbst darf auf seine Daten zugreifen.
- Zugriff von außen nur über Methoden des Objektes
- stellt konsistenten Zustand des Objektes sicher
- sollte unabhängig von der eingesetzten Sprache in jedem Fall konsequent umgesetzt werden

👉 Kapselung schränkt den Zugriff auf Variablen und Funktionen ein und sorgt so für Entkopplung, Modularität und Konsistenz.

Objekt (*object*)

(grundlegender) Baustein eines objektorientierten Programms besteht aus den Daten *und* dem zugehörigen Verhalten

Daten Attribute (*attributes*)
Variablen beliebigen Datentyps

Verhalten Methoden (*methods*)
Funktionen/Routinen, die auf den Daten arbeiten

- ☛ Objekte werden innerhalb einer Anwendung erzeugt und existieren nur über die Laufzeit der Anwendung.

Klasse (*class*)

Blaupause für die Erzeugung eines Objektes
Definition der Daten (Attribute) und des Verhaltens (Methoden)

- ▶ Klassen erweitern die Datentypen einer Sprache.
- ▶ Jedes Objekt gehört zu einer Klasse (Typ).
- ▶ Normalerweise muss ein Objekt erzeugt werden, um auf Attribute und Methoden zugreifen zu können.

 **Metapher vom Backen:**
Klassen sind die Ausstecher, Objekte die Plätzchen.

Kontext (*scope*)

Kontrolle des Zugriffs auf Variablen und Routinen

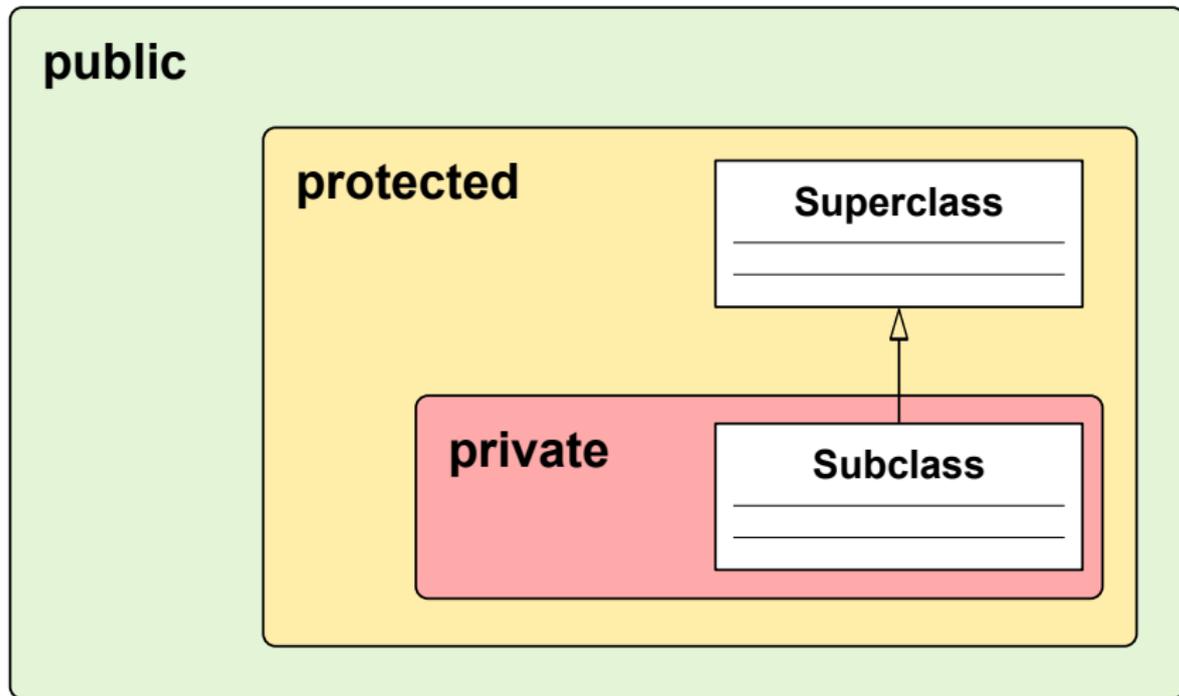
Kontext in der OOP:

`public` Jeder hat lesenden und schreibenden Zugriff.

`protected` Zugriff nur für die aktuelle und abgeleitete Klassen

`private` Nur die jeweilige Klasse/das Objekt hat Zugriff.

- ☛ meist sowohl für Attribute als auch für Methoden angebar
- ☛ nicht von jeder Programmiersprache voll unterstützt



Konstituierend

- ▶ Kapselung (*encapsulation*)
- ▶ Vererbung (*inheritance*)
- ▶ Polymorphie (*polymorphism*)

Zusätzlich

- ▶ Zusammensetzung (*composition*)
- 👉 Eine Programmiersprache gilt nur dann als objektorientiert, wenn sie die ersten drei Konzepte implementiert.

Kapselung (*encapsulation*)

Ein Objekt enthält Daten *und* zugehöriges Verhalten und kann beides nach Belieben vor anderen Objekten verstecken.

- ▶ Normalfall: alle Attribute eines Objektes sind privat
 - Nur die Methoden des Objektes haben Zugriff.
- ▶ Nutzer kennt nur die Signatur der Methoden/Objekte
 - Signatur: Namen und Parameter der Methoden
 - konkrete Implementierung irrelevant (*separation of concerns*)
- ▶ nur so viele öffentliche Methoden wie notwendig
 - Sparsamkeitsprinzip: Implementation verstecken
 - erleichtert die Wiederverwendbarkeit

Grundparadigma der Kapselung

Jedes Objekt ist für sich selbst verantwortlich.
Der Zugriff erfolgt nur über die öffentlichen Methoden des Objektes.

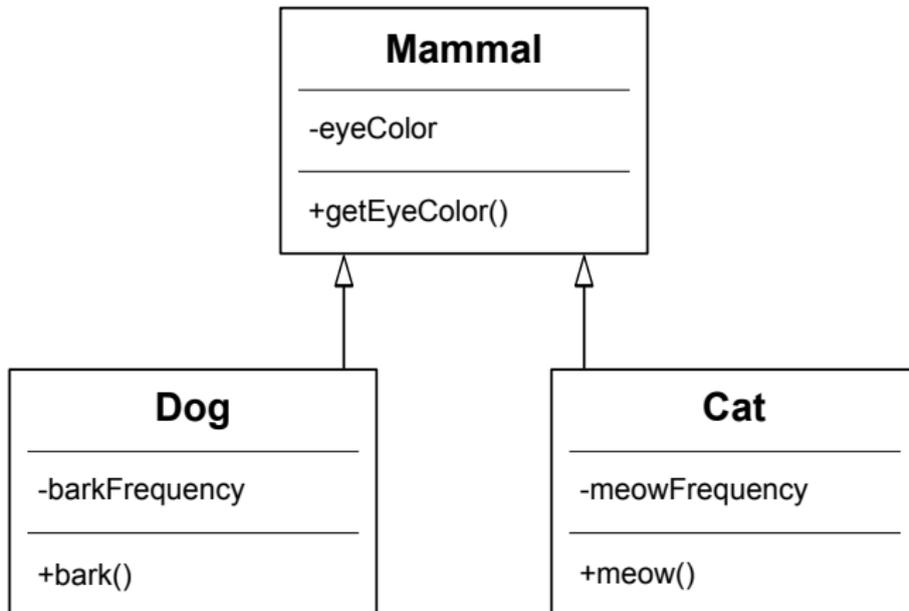
Warum ist Kapselung so wichtig?

- ▶ Grundlage der Modularität und Austauschbarkeit
- ▶ Voraussetzung für die anderen OOP-Konzepte:
Vererbung, Polymorphie, Zusammensetzung
- ☛ sorgt für strikte Trennung zwischen
Schnittstelle (*interface*) und Implementierung

Vererbung (*inheritance*)

Eine Klasse kann von einer anderen Klasse erben und aus den Attributen und Methoden der Superklasse Nutzen ziehen.

- ▶ zentrales Konzept für die Wiederverwendung von Code
- ▶ Superklassen implementieren nur das Notwendigste (den kleinsten gemeinsamen Nenner).
- ▶ Erben von mehr als einer Klasse (*multiple inheritance*)
 - nicht von allen Programmiersprachen unterstützt
 - Alternative: Schnittstellen-Klassen (*interfaces*)
- ▶ Wichtig: Zu lange Stammbäume werden unübersichtlich.

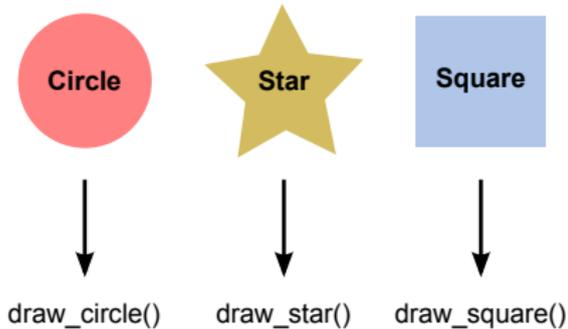


Polymorphie (*polymorphism*)

Ähnliche Objekte können auf die gleiche Botschaft in unterschiedlicher Weise reagieren („Vielgestaltigkeit“).

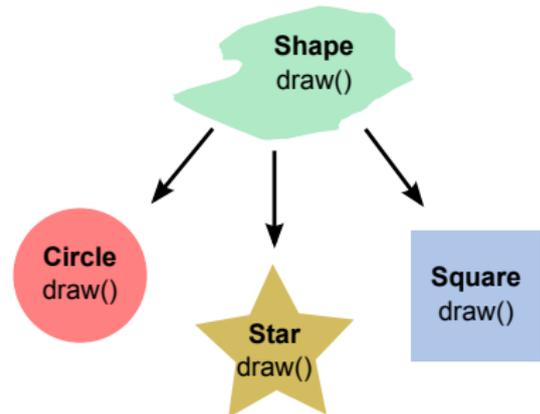
- ▶ eng mit Vererbung verknüpft, von zentraler Bedeutung
- ▶ Eine Klasse erbt eine Methode von einer Superklasse und implementiert die Funktionalität entsprechend.
- ▶ hochgradig modular und erweiterbar:
 - Jede Klasse ist für sich selbst verantwortlich.
 - bei neuer Subklasse keine Modifikation der Superklasse
 - Aufruf der Methode bleibt identisch

non-OO example



Choose a shape and print

OO example



A shape knows how to print itself

Listing 1: Zeichnen unterschiedlicher Formen: strukturiert

```
shape = "star"

if shape == "circle":
    draw_circle()
elif shape == "star":
    draw_star()
elif shape == "square":
    draw_square()
```

☞ Code muss für jede neue Form angepasst werden.

Listing 2: Zeichnen unterschiedlicher Formen: objektorientiert

```
shape = Star()
shape.draw()
```

☞ Jede Klasse mit einer Methode `draw` kann genutzt werden.

Listing 3: Zeichnen unterschiedlicher Formen: objektorientiert

```
class Shape():
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        # actual drawing of circle

class Star(Shape):
    def draw(self):
        # actual drawing of star

class Square(Shape):
    def draw(self):
        # actual drawing of square

# Instantiate object and call draw method
shape = Star()
shape.draw()
```

☛ Jede Subklasse überschreibt die Methode `draw` der Superklasse.

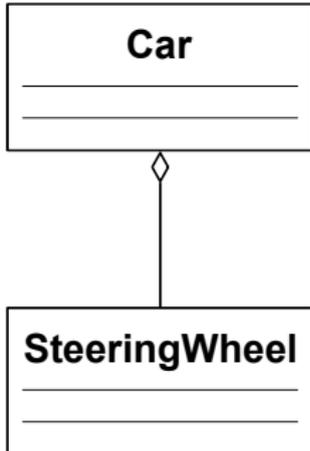
Zusammensetzung (*composition*)

Ein Objekt ist aus anderen Objekten zusammengesetzt.
Die Objekte können anderweitig vollkommen unabhängig sein.

- ▶ zentrales Konzept für die Wiederverwendung von Code
- ▶ Unterschied zur Vererbung (*inheritance*):
Wechselwirkung *unabhängiger* Objekte/Klassen
- ▶ Zwei Formen:
 - Aggregation (*aggregation*)
 - Assoziation (*association*)

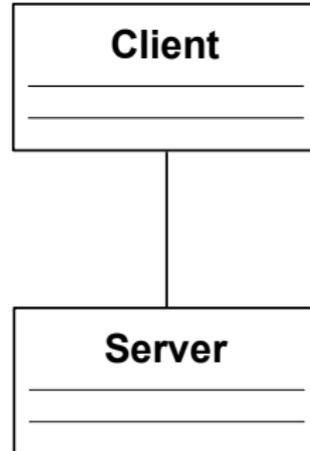
Aggregation

parts of a whole



Association

services provided





Zusammensetzung (*composition*)

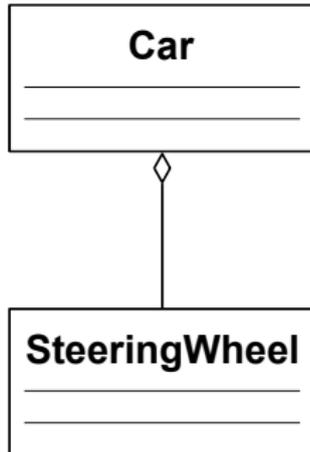
- ▶ Wechselwirkung zwischen *unabhängigen* Objekten
- ▶ Kapselung (*encapsulation*) bleibt voll erhalten.

Vererbung (*inheritance*)

- ▶ Die Subklasse erbt alle Eigenschaften (Attribute, Methoden) von der Superklasse.
- ▶ Die Subklasse ist vom gleichen Typ wie die Superklasse (Grundlage der Polymorphie).
- ▶ Änderungen der Superklasse beeinflussen die Subklasse (Kapselung wird entsprechend geschwächt).

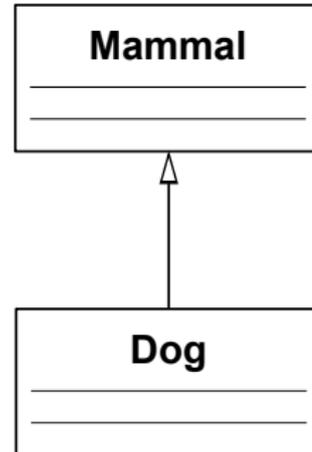
Composition

has-a



Inheritance

is-a



These

Das Verständnis der Grundlagen objektorientierter Programmierung führt nicht automatisch zu gutem, lesbarem und genauso wenig zu modularem, wiederverwendbarem Code.

- ▶ Hammer-Nagel-Problem
 - Vererbungshierarchien anfangs oft hoffnungslos komplex
- ▶ Überblick über die Problemstellung verschaffen
 - Eigenschaften und Verhalten von Einheiten
 - liefert Hinweise auf Design von Klassen
- ☞ Ausprobieren! Der erste Entwurf ist nie perfekt.

- ▶ Programmierung gegen Schnittstellen (*interfaces*), nicht gegen Implementierungen
 - zentral für die Kapselung und Modularität
 - Voraussetzung: frühe Definition von Schnittstellen
- ▶ möglichst flache Hierarchien bei der Vererbung
 - Vererbung bricht in gewisser Weise die Kapselung.
 - mögliche Alternativen: Zusammensetzung bzw. Implementierung von Schnittstellen (*interfaces*)
- ▶ minimale öffentliche Schnittstelle
 - Kapselung: Verstecken von Daten und Implementierung
 - wesentlich für die Wiederverwendbarkeit
- ☛ (Entwurfs-)Muster (*design patterns*) setzen genau hier an.
- ☛ Wird im Abschnitt „Softwarearchitektur“ noch thematisiert.



- ❏ OOP ist eine Abstraktion, die die Abbildung realer Probleme auf Code stark vereinfachen kann.
- ❏ OOP erfordert ein grundlegendes Umdenken verglichen mit anderen Programmierparadigmen.
- ❏ Richtig eingesetzt erleichtert OOP die Erstellung modularer, robuster und wiederverwendbarer Codes.
- ❏ Objektorientierte und strukturierte Programmierung schließen sich nicht gegenseitig aus.
- ❏ Die Kernaspekte von OOP sind Kapselung, Vererbung und Polymorphie.