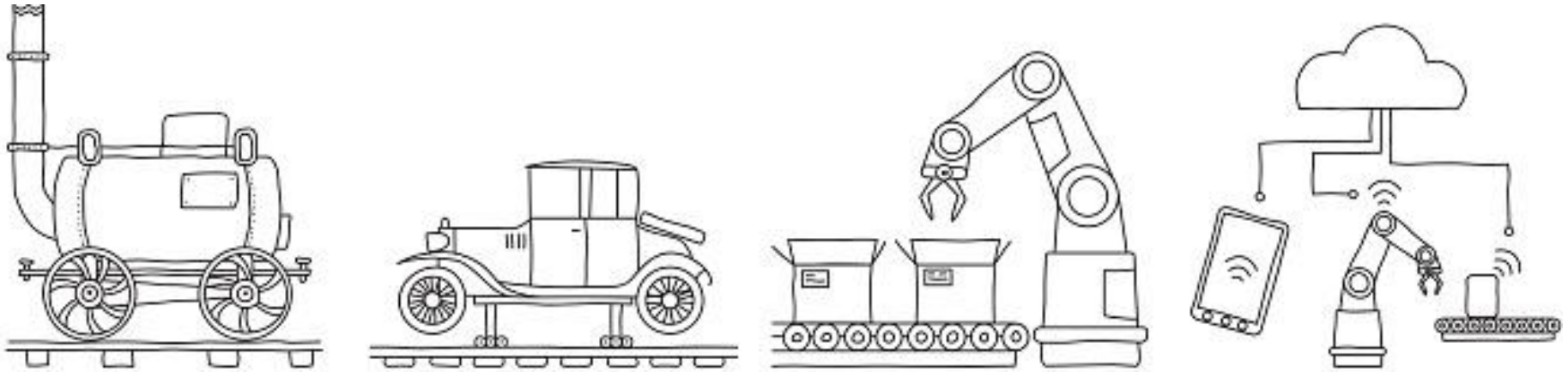
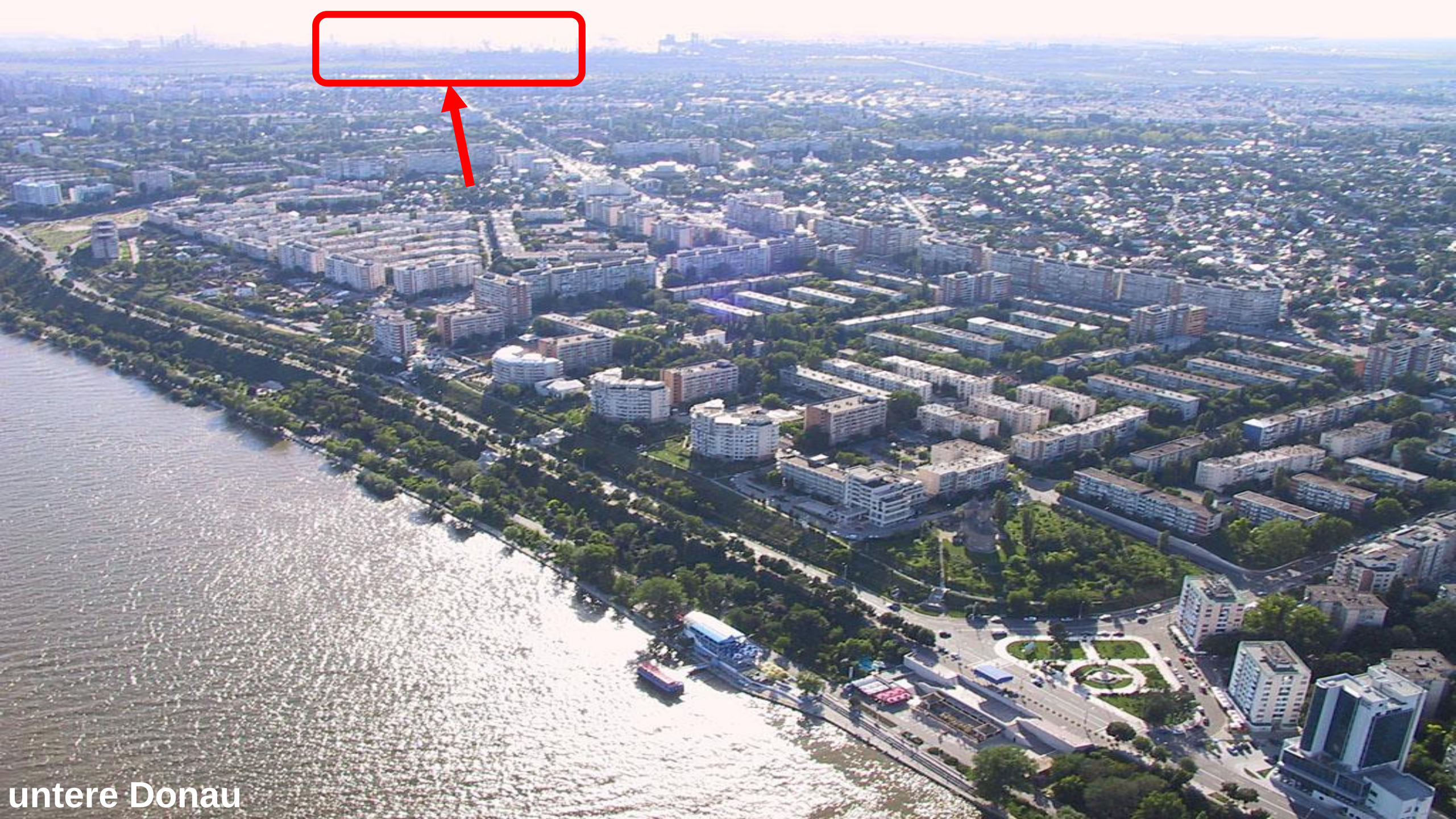


Einführung in die objektorientierte Modellierung und Programmierung an einem Beispiel aus der Produktion

Dr. Cristian Axenie

Technische Hochschule Ingolstadt

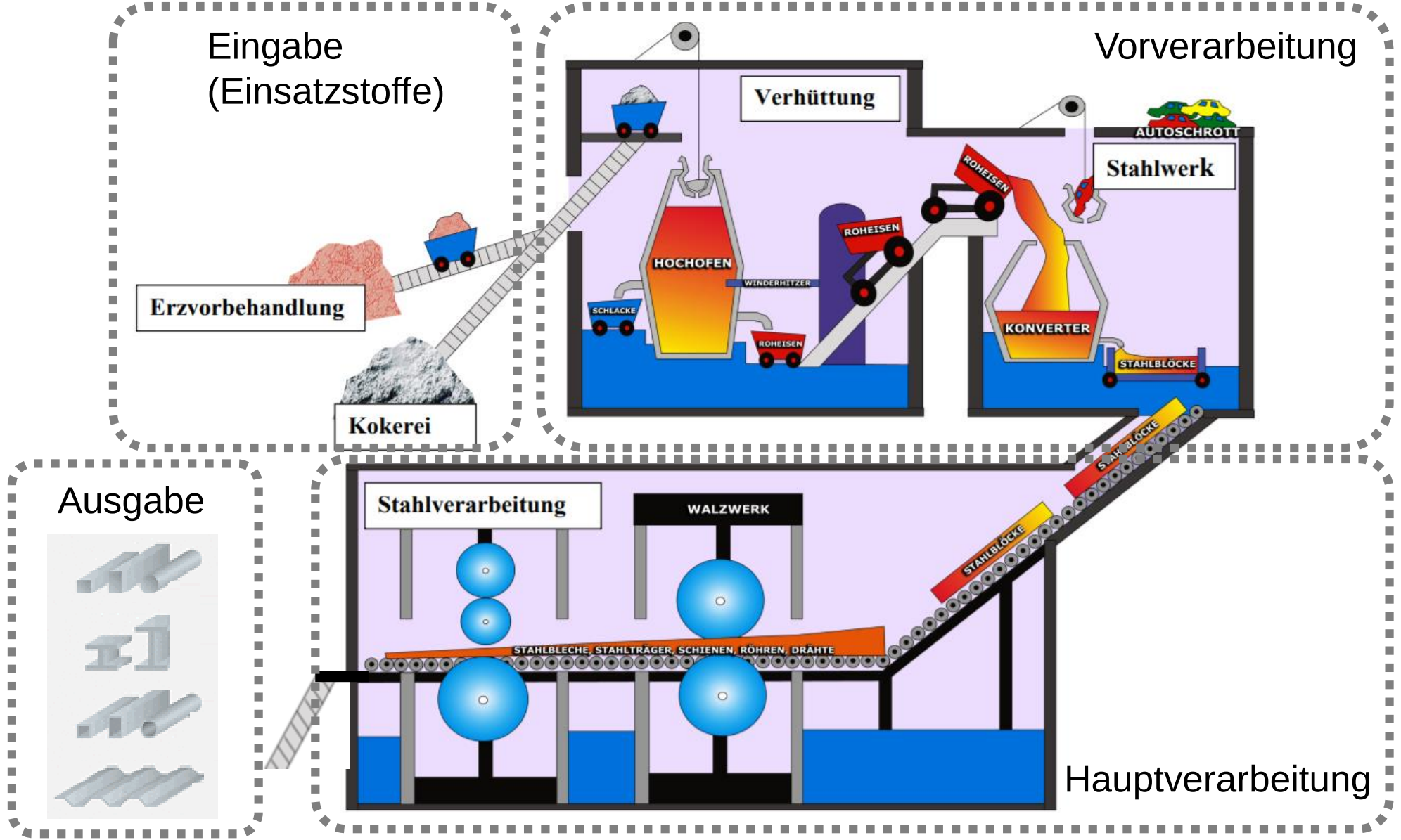

untere Donau

Stahlproduktionslinie

Inhalt

- Problemstellung
- Objektorientierte Modellierung (OOM)
- Objektorientierte Programmierung (OOP)
- Fazit

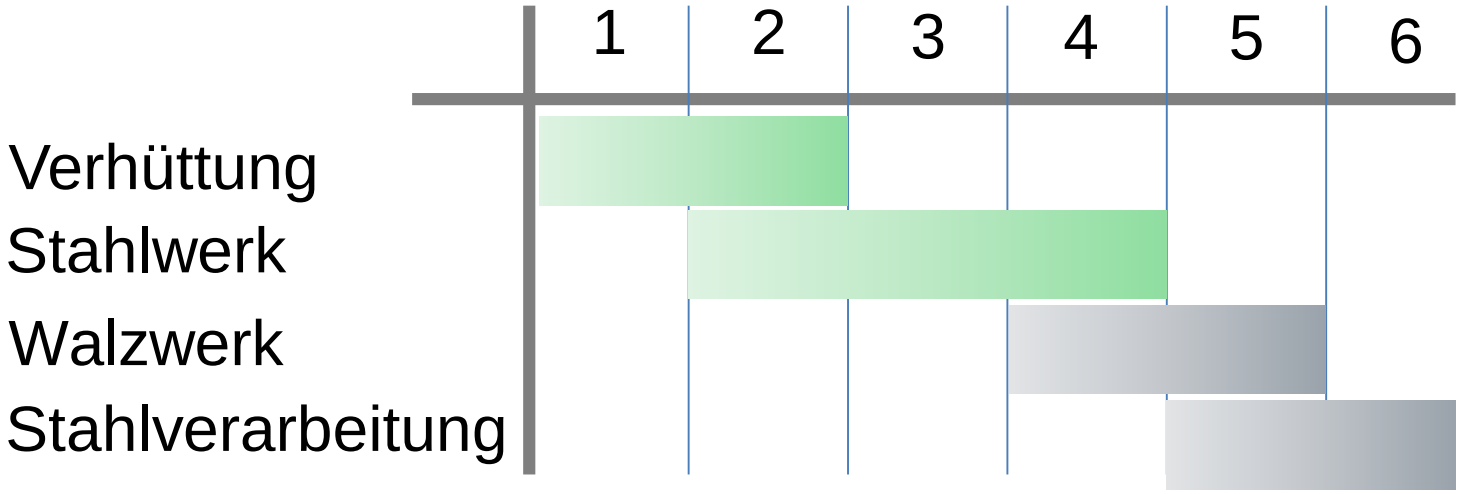
Problemstellung



Problemstellung

Funktionsablauf

Produktionsoptimierungsprobleme: Prozessdaueroptimierung (Std.)

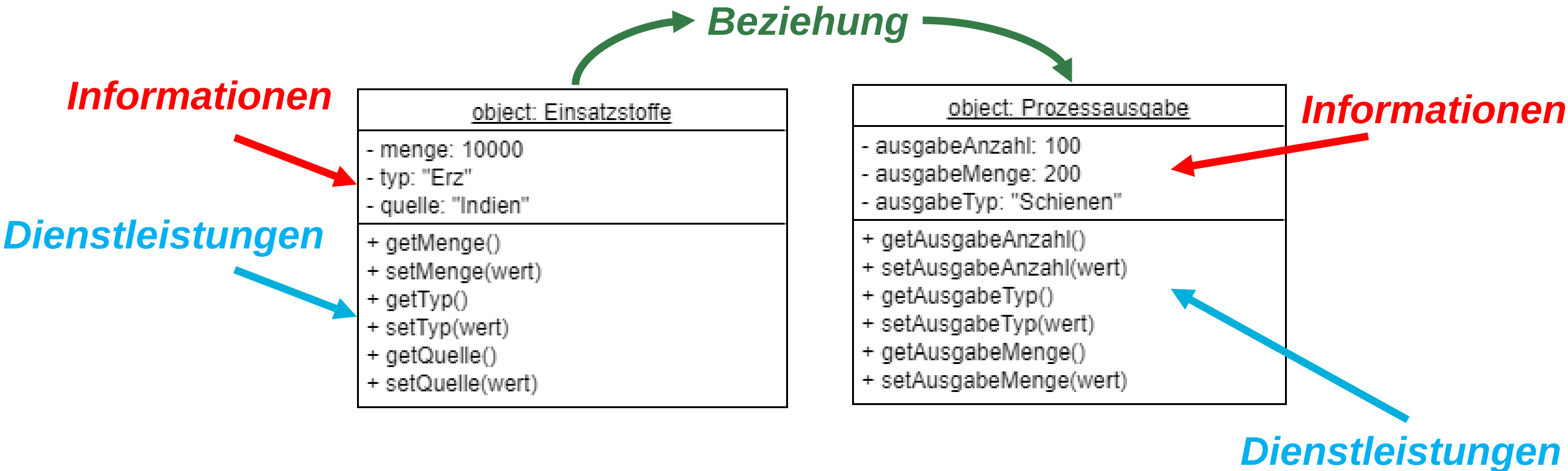


Objektorientierte Modellierung

Objekten

Wir betrachten die Welt als eine Menge von **Objekten**, die zueinander in **Beziehung stehen** und gegebenenfalls miteinander **kommunizieren**.

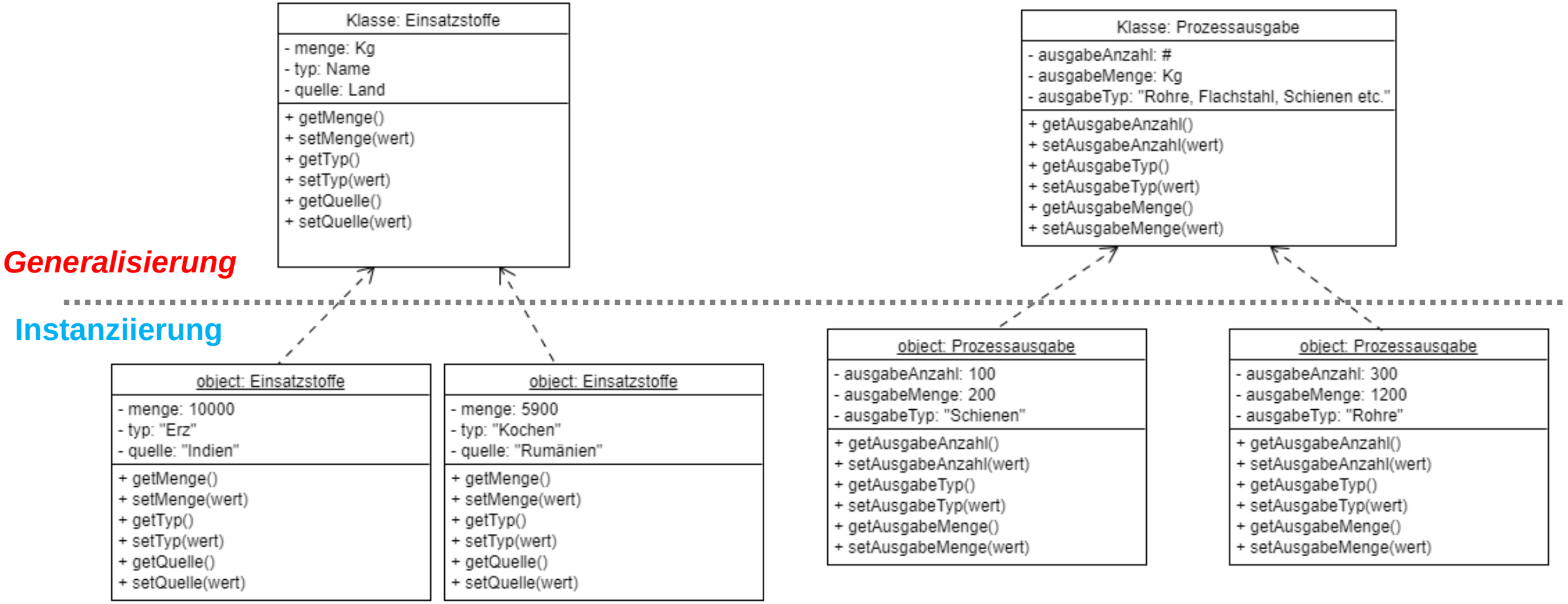
Objekte sind Elemente, die in einem **Anwendungsbereich** von **Bedeutung** sind. Aus der Sicht des **Benutzers** stellen Objekte bestimmte **Dienstleistungen** und **Informationen** zur Verfügung.



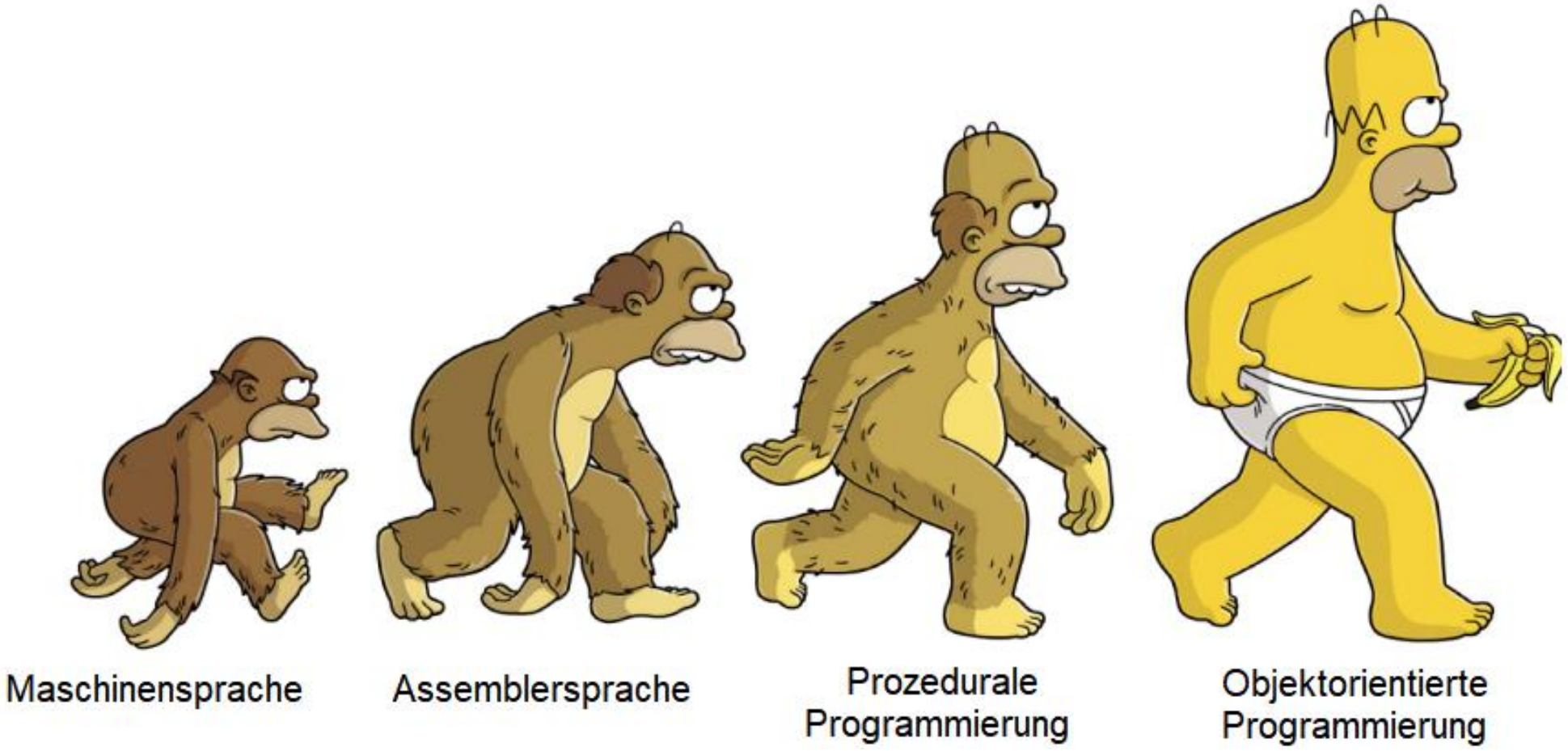
Objektorientierte Modellierung

Klasse

Eine **Klasse** entsteht durch die **Abstraktion** von den Details **gleichartiger Objekte** und beschreibt die **Eigenschaften** und **Struktur** einer Menge **gleichartiger Objekte**.



Objektorientierte Programmierung

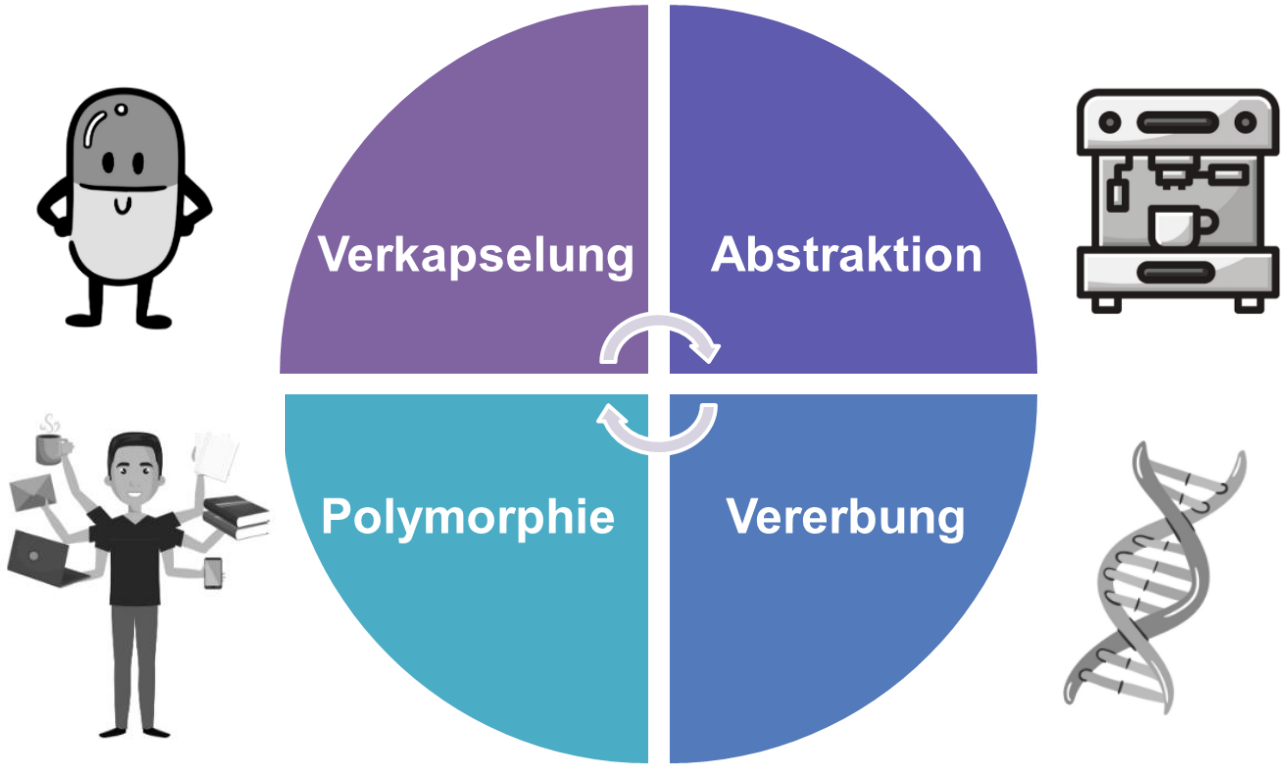


Objektorientierte Programmierung

Objekt = Attributen (Information) + Methoden (Dienstleistungen)

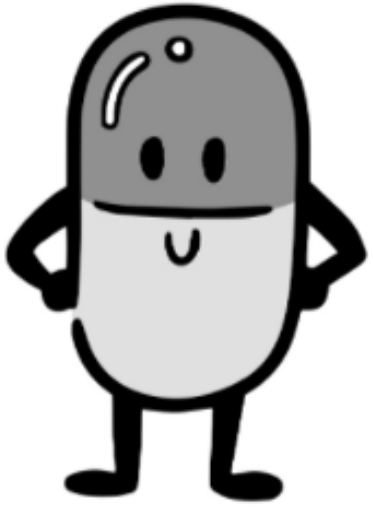
Klasse = Bauplan für die Erstellung von **Objekten**

Prinzipien



Objektorientierte Programmierung

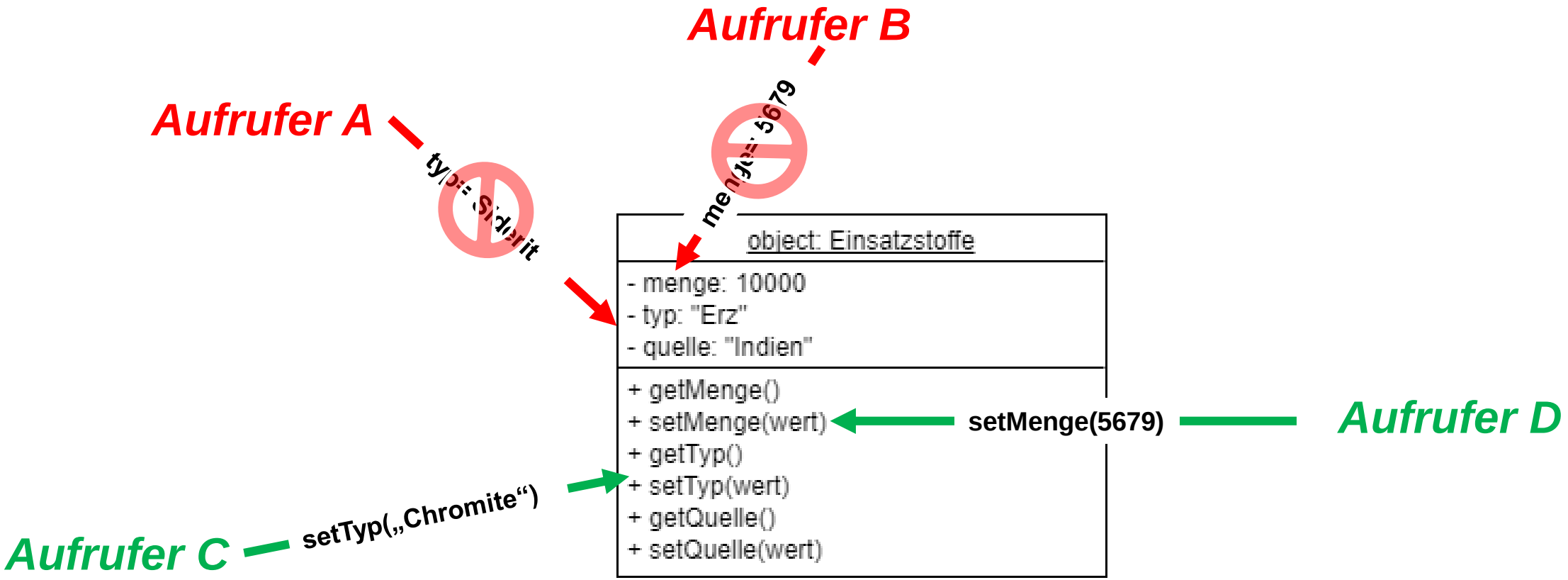
Prinzip 1: Datenerkapselung



Objektorientierte Programmierung

Prinzip 1: Datenerkapselung

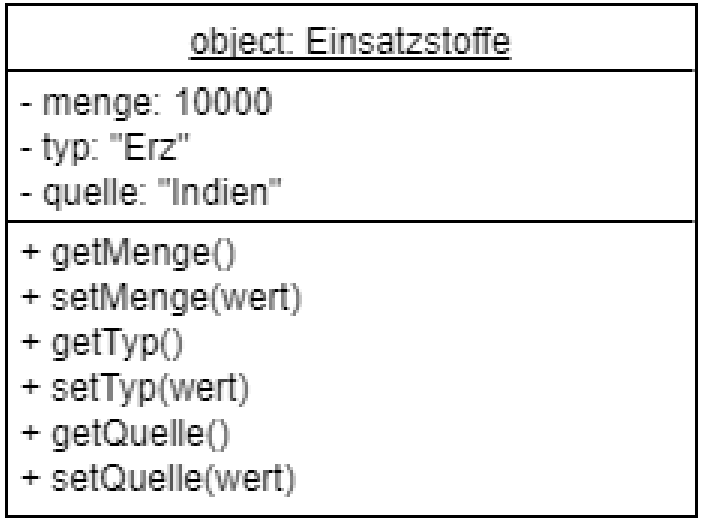
Kapselung wird erreicht, wenn jedes **Objekt** seinen **Zustand private** innerhalb einer Klasse behält. Andere Objekte haben **keinen direkten Zugriff** auf diesen Zustand. Stattdessen können sie nur eine Liste **öffentlicher Funktionen** aufrufen – sogenannte **Methoden**.



Objektorientierte Programmierung

Datenerkapselung implementierung

```
class EinsatzstoffeEingabe(object):
    def __init__(self, m, t, q):
        self.menge = m
        self.typ = t
        self.quelle = q

    def getMenge(self):
        return self.menge

    def setMenge(self, m):
        self.menge = m

    def getTyp(self):
        return self.typ

    def setTyp(self, t):
        self.typ = t

    def getQuelle(self):
        return self.quelle

    def setQuelle(self, q):
        self.quelle = q
```



Objektorientierte Programmierung

Prinzip 2: Abstraktion

Objektorientierte Programmierung

Prinzip 2: Abstraktion

Das Anwenden von **Abstraktion** bedeutet, dass **jedes Objekt nur einen High-Level-Mechanismus** für seine Verwendung **verfügbar** machen sollte. Dieser Mechanismus sollte interne **Implementierungsdetails verbergen**.
Es sollte nur **Vorgänge aufdecken**, die für die anderen Objekte **relevant** sind.

Aufrufer D

Implementierung: optimierung

optimizeProzessDauer()

```
def optimizeProzessDauer(self):
    if self.getGesamtGliederDauer() < MIN_EBENE:
        for i in range(self.prozessGliederAnzahl):
            self.prozessGlieder[i].setDauer(MIN_EBENE)
    elif self.getGesamtGliederDauer() > MAX_EBENE:
        for i in range(self.prozessGliederAnzahl):
            self.prozessGlieder[i].setDauer(self.prozessGlieder[i].getDauer() + self.getGesamtGliederDauer() / 5.0)
    else:
        for i in range(self.prozessGliederAnzahl):
            self.prozessGlieder[i].setDauer(
                self.prozessGlieder[i].getDauer() + self.getGesamtGliederDauer() / 5.0 + random.random())
```

Klasse: StahlProzessPlannung
- prozessGliederAnzahl: # - prozessGliederName: "Vorverarbeitung, Hauptverarbeitung etc." - prozessGlieder: VorVerarbeitung, HauptVerarbeitung Objekte
+ getProzessGliederAnzahl() + setProzessGliederAnzahl(wert) + getProzessGliederName() + setProzessGliederName(wert) + getProzessGlieder() + setProzessGlieder(wert) + getGliederDauer(gliederName) + getGesamtGliederDauer() + optimizeProzessDauer()

Objektorientierte Programmierung

Abstraktion implementierung

```
class StahlProzessPlannung(object):
    def __init__(self, a, n, g):...

    def getProzessGliederAnzahl(self):...

    def setProzessGliederAnzahl(self, a):...

    def getProzessGliederName(self):...

    def setProzessGliederName(self, n):...

    def getProzessGlieder(self):...

    def setProzessGlieder(self, g):...

    def getGliederDauer(self, name):...

    def getGesamtGliederDauer(self):...

    def optimizeProzessDauer(self):
        if self.getGesamtGliederDauer() < MIN_EBENE:
            for i in range(self.prozessGliederAnzahl):
                self.prozessGlieder[i].setDauer(MIN_EBENE)
        elif self.getGesamtGliederDauer() > MAX_EBENE:
            for i in range(self.prozessGliederAnzahl):
                self.prozessGlieder[i].setDauer(self.prozessGlieder[i].getDauer() + self.getGesamtGliederDauer() / 5.0)
        else:
            for i in range(self.prozessGliederAnzahl):
                self.prozessGlieder[i].setDauer(
                    self.prozessGlieder[i].getDauer() + self.getGesamtGliederDauer() / 5.0 + random.random())
```

Klasse: StahlProzessPlannung
- prozessGliederAnzahl: # - prozessGliederName: "Vorverarbeitung, Hauptverarbeitung etc." - prozessGlieder: VorVerarbeitung, HauptVerarbeitung Objekte
+ getProzessGliederAnzahl() + setProzessGliederAnzahl(wert) + getProzessGliederName() + setProzessGliederName(wert) + getProzessGlieder() + setProzessGlieder(wert) + getGliederDauer(gliederName) + getGesamtGliederDauer() + optimizeProzessDauer()

Implementierung: optimierung

Objektorientierte Programmierung

Prinzip 3: Vererbung

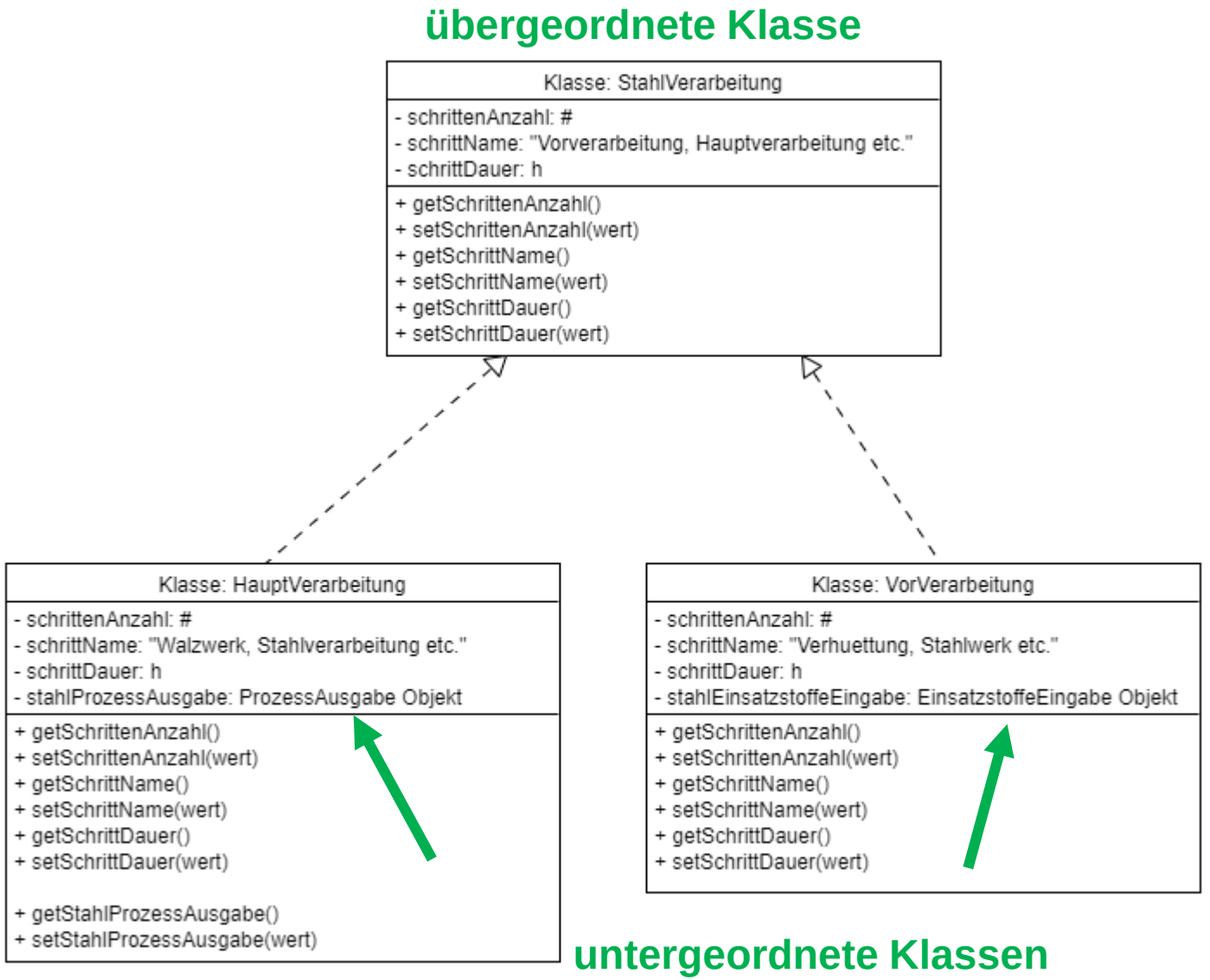
Objektorientierte Programmierung

Prinzip 3: Vererbung

Die **Vererbung** ermöglicht es uns, eine **(untergeordnete) Klasse** zu erstellen, indem wir von einer anderen **(übergeordneten) Klasse ableiten**. Auf diese Weise bilden wir eine **Hierarchie**.

Die **Kindklasse** verwendet **alle Felder** und **Methoden** der **Elternklasse** (gemeinsamer Teil) und kann **eigene** (eindeutiger Teil) **implementieren**.

Auf diese Weise **fügt jede Klasse** nur das hinzu, **was** für sie **erforderlich ist**, während die gemeinsame **Logik** mit den Elternklassen **wiederverwendet** wird.

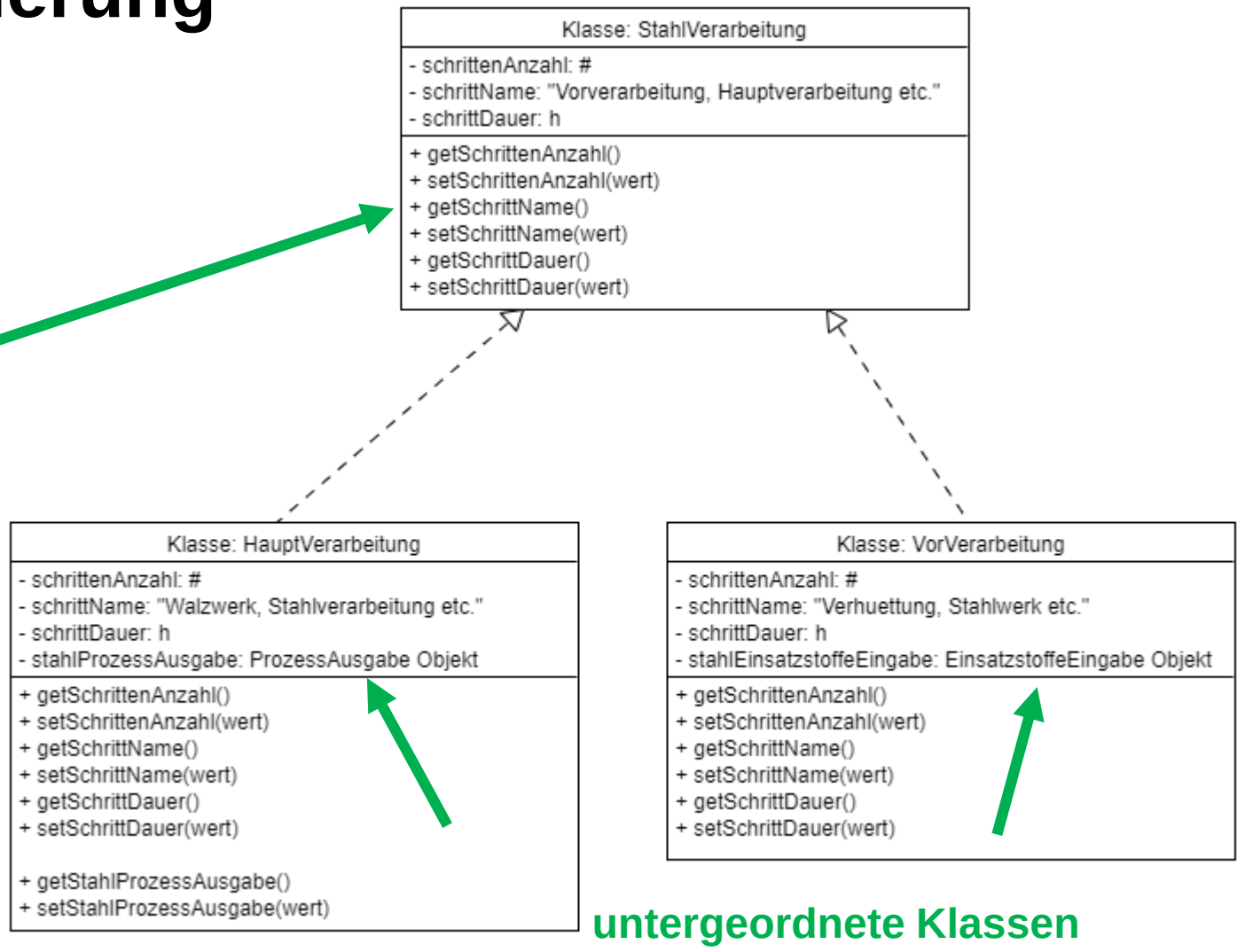


Objektorientierte Programmierung

Vererbung implementierung

```
class HauptVerarbeitung(StahlVerarbeitung):
    def __init__(self, a, n, d, pa):
        self.stahlProzessAusgabe = pa
        super().__init__(a, n, d)

    def getAnzahl(self):
        return self.schrittenAnzahl

    def setAnzahl(self, a):
        self.schrittenAnzahl = a

    def getSchrittName(self):
        return self.schrittName

    def setSchrittName(self, n):
        self.schrittName = n

    def getSchrittDauer(self):
        return self.schrittDauer

    def setSchrittDauer(self, d):
        self.schrittDauer = d

    def getStahlProzessAusgabe(self):
        return self.stahlProzessAusgabe

    def setStahlProzessAusgabe(self, pa):
        self.stahlProzessAusgabe = pa
```

übergeordnete Klasse



untergeordnete Klassen

Objektorientierte Programmierung

Prinzip 4: Polymorphismus

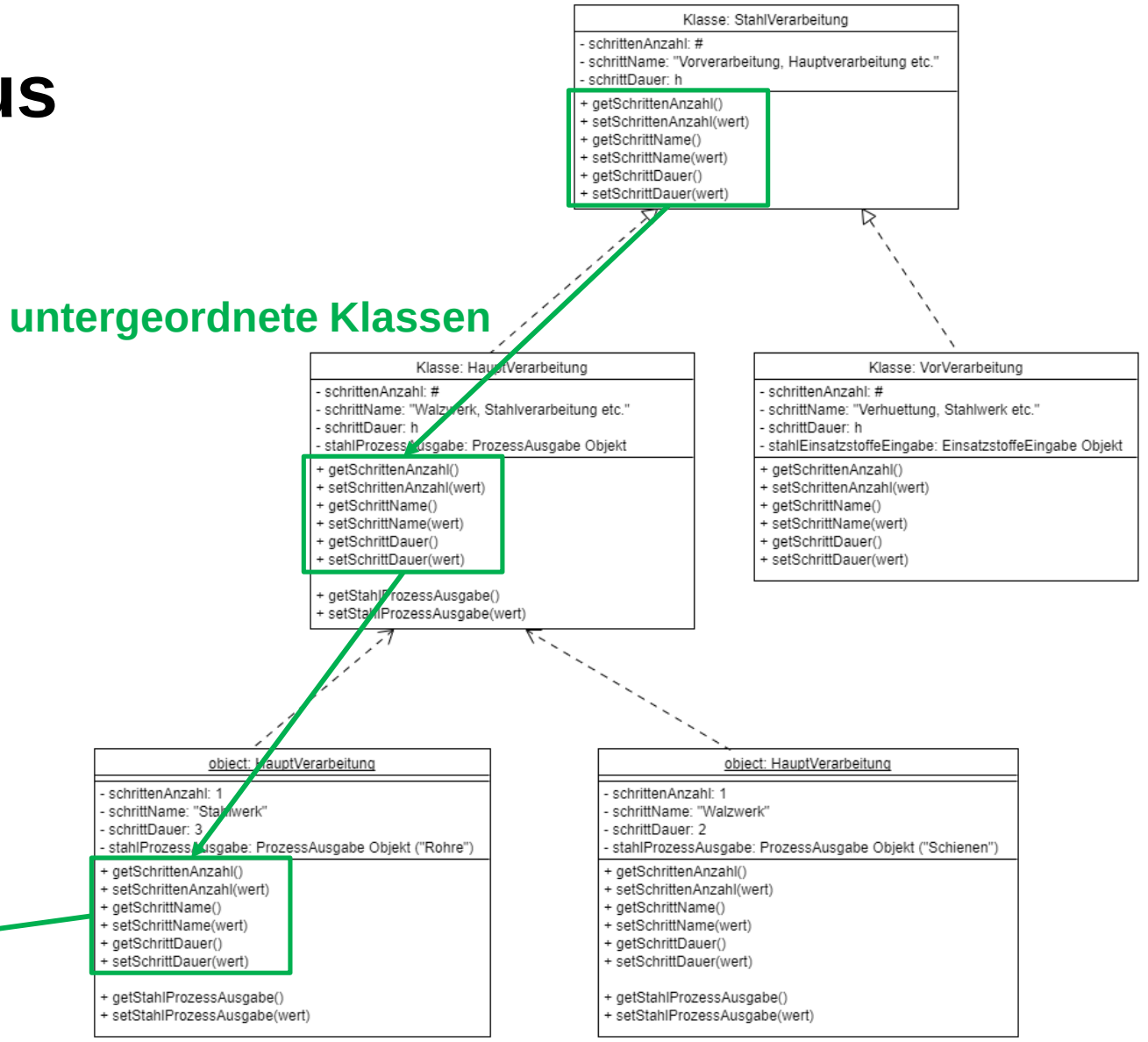
Objektorientierte Programmierung

Prinzip 4: Polymorphismus

Polymorphismus bietet eine Möglichkeit, **eine Klasse genau wie ihre Eltern zu verwenden**, damit es **keine Verwechslungen** mit dem Mischen von Typen gibt.

Aber jede **untergeordnete Klasse behält** ihre eigenen **Methoden bei**, wie sie sind.

Jedes Mal, wenn eine eine **Methode** eine **Instanz** des **übergeordneten Elements erwartet** kümmert sich die Sprache darum, die **richtige Implementierung** der **gemeinsamen Methode auszuwerten** – **unabhängig** davon, welches **untergeordnete Element** übergeben wird.



- schrittenAnzahl: 1
- schrittName: "Stahlwerk"
- schrittDauer: 3
- stahlProzessAusgabe: ProzessAusgabe Objekt ("Rohre")

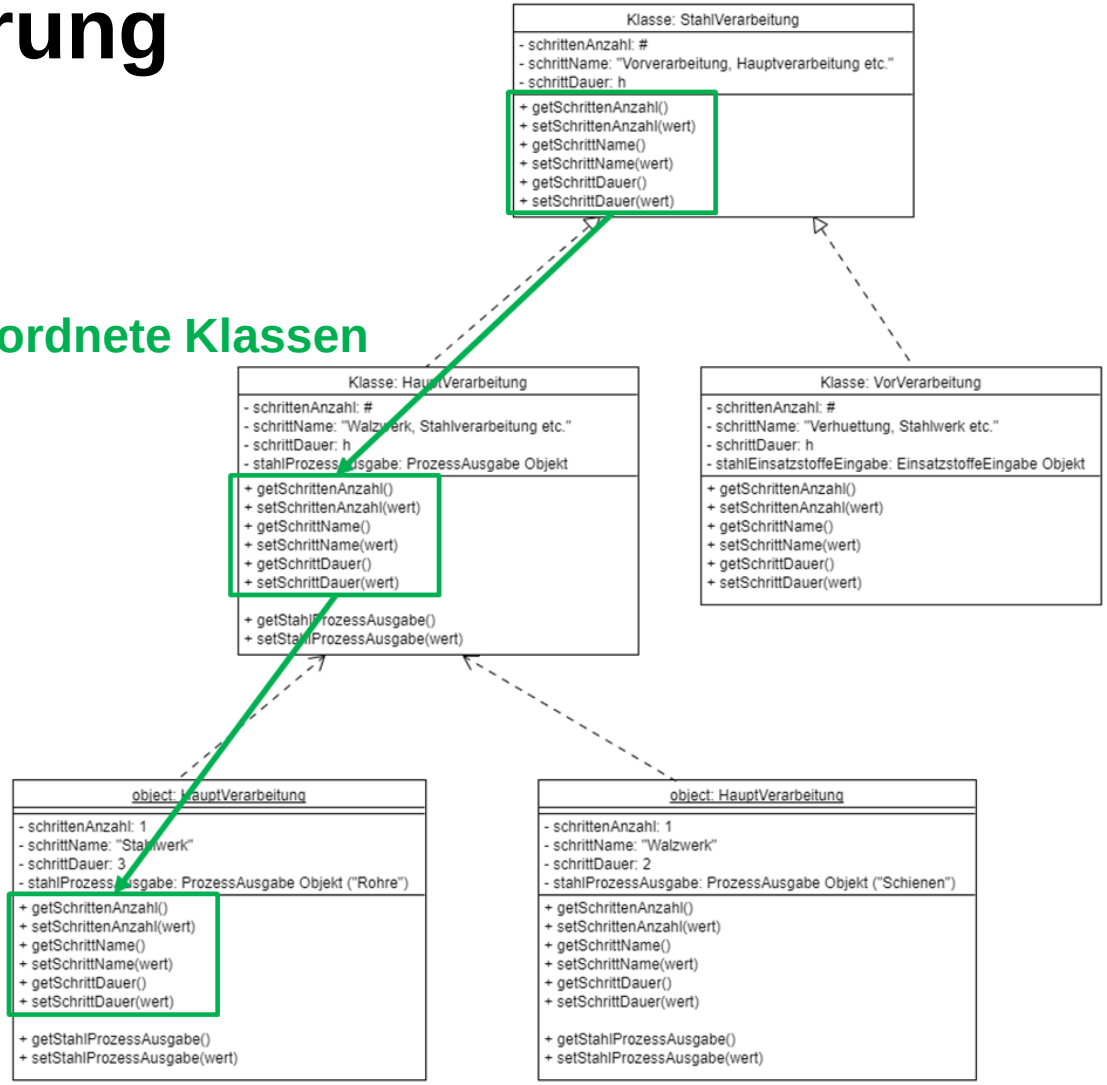
Objektorientierte Programmierung

Polymorphismus implementierung

```
from eingabe import EinsatzstoffeEingabe
from ausgabe import *
from verarbeitung import *
from plannung import *

if __name__ == '__main__':
    eingabe1 = EinsatzstoffeEingabe(100000, "Erz", "Indien")
    eingabe2 = EinsatzstoffeEingabe(59000, "Kochen", "Rumänien")

    ausgabe1 = ProzessAusgabe(100, 200, "Schienen")
    ausgabe2 = ProzessAusgabe(300, 1200, "Rohre")

    verarbeitung0 = StahlVerarbeitung(1, "Walzwerk", 4)
    verarbeitung1 = VorVerarbeitung(1, "Walzwerk", 2, ausgabe1)
    verarbeitung2 = HauptVerarbeitung(1, "Stahlwerk", 3, ausgabe2)

    verarbeitung0.getSchrittDauer()
    verarbeitung1.getSchrittDauer()
    verarbeitung2.getSchrittDauer()

    plannung1 = StahlProzessPlannung(2, "Vorverarbeitung", verarbeitung1)
    plannung2 = StahlProzessPlannung(2, "Hauptverarbeitung", verarbeitung2)
```

übergeordnete Klasse

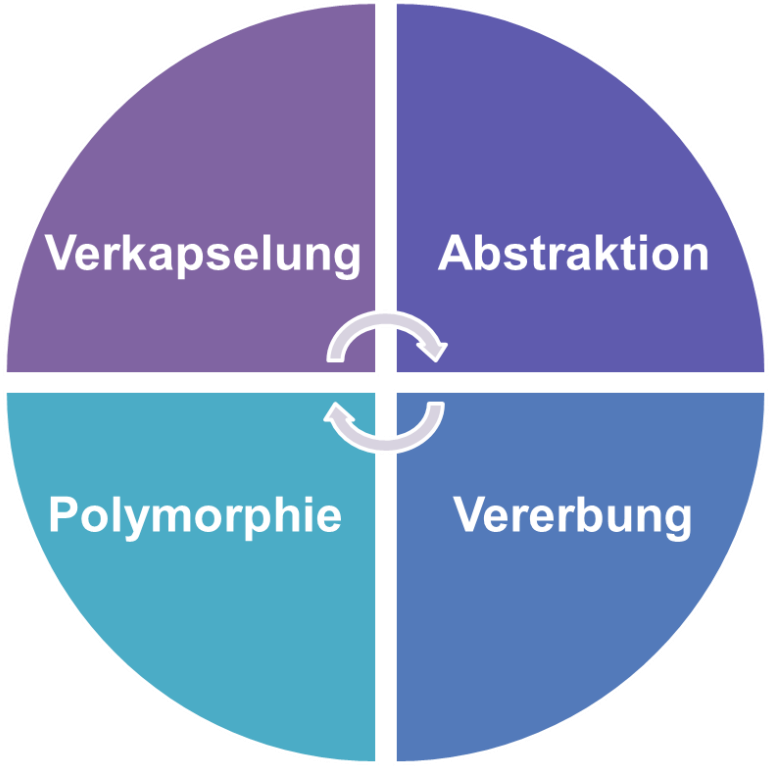
untergeordnete Klassen



Fazit

OOM bietet ein neues **Paradigma**, um **reale Probleme zu modellieren**, und **OOP** bietet die **Methoden** und **Praktiken**, um Modelle in **Softwarekomponenten** umzuwandeln.

1. Effektive **Problemlösung** durch **Abstraktion**
2. **Modularität** für einfachere Fehlersuche durch **Verkapselung**
3. **Wiederverwendung** von Code durch **Vererbung**
4. **Flexibilität** durch **Polymorphismus**



Literaturverzeichnis

1. Bernhard Lares, Gregor Raýman, und Stefan Strich. Objektorientierte Programmierung: Das umfassende Handbuch. Lernen Sie die Prinzipien guter Objektorientierung. Rheinwerk, 2018.

2. Eric Freeman, Elisabeth Robson, Bert Bates, und Kathy Sierra. Head First Design Patterns. O'Reilly, 2014.

Online-Version

Python Programmcodes
auf **GitHub** verfügbar