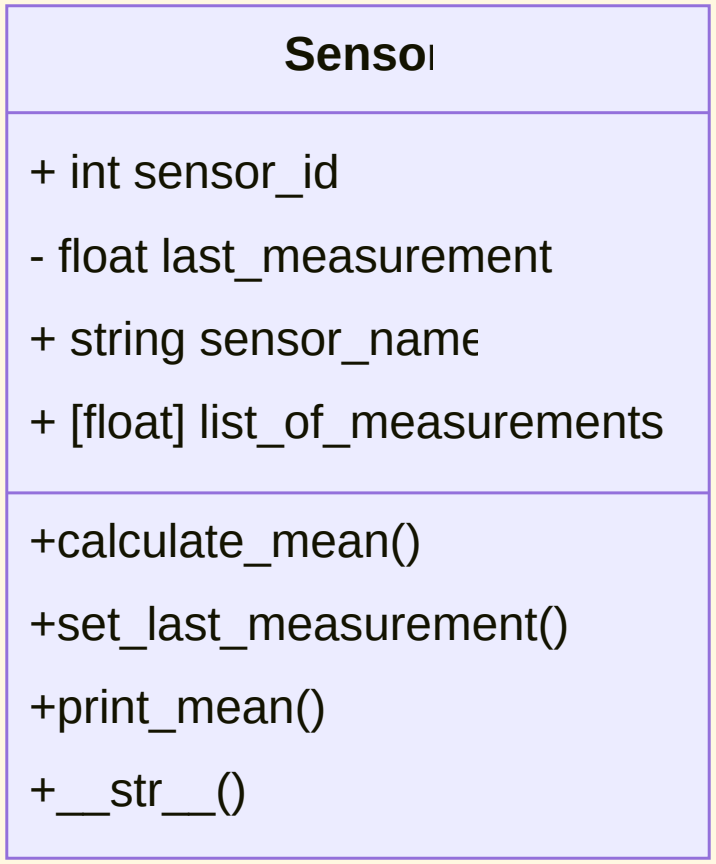
Python Grundlagen

Julian Huber & Matthias Panny

Objektorientierung II

🎯 Lernziele

- Studierenden können Sachverhalte mittels UML-Klassendiagramm abbilden
- Studierende nutzen Vererbung und weitere Konzepte der Objektorientierten Programmierung

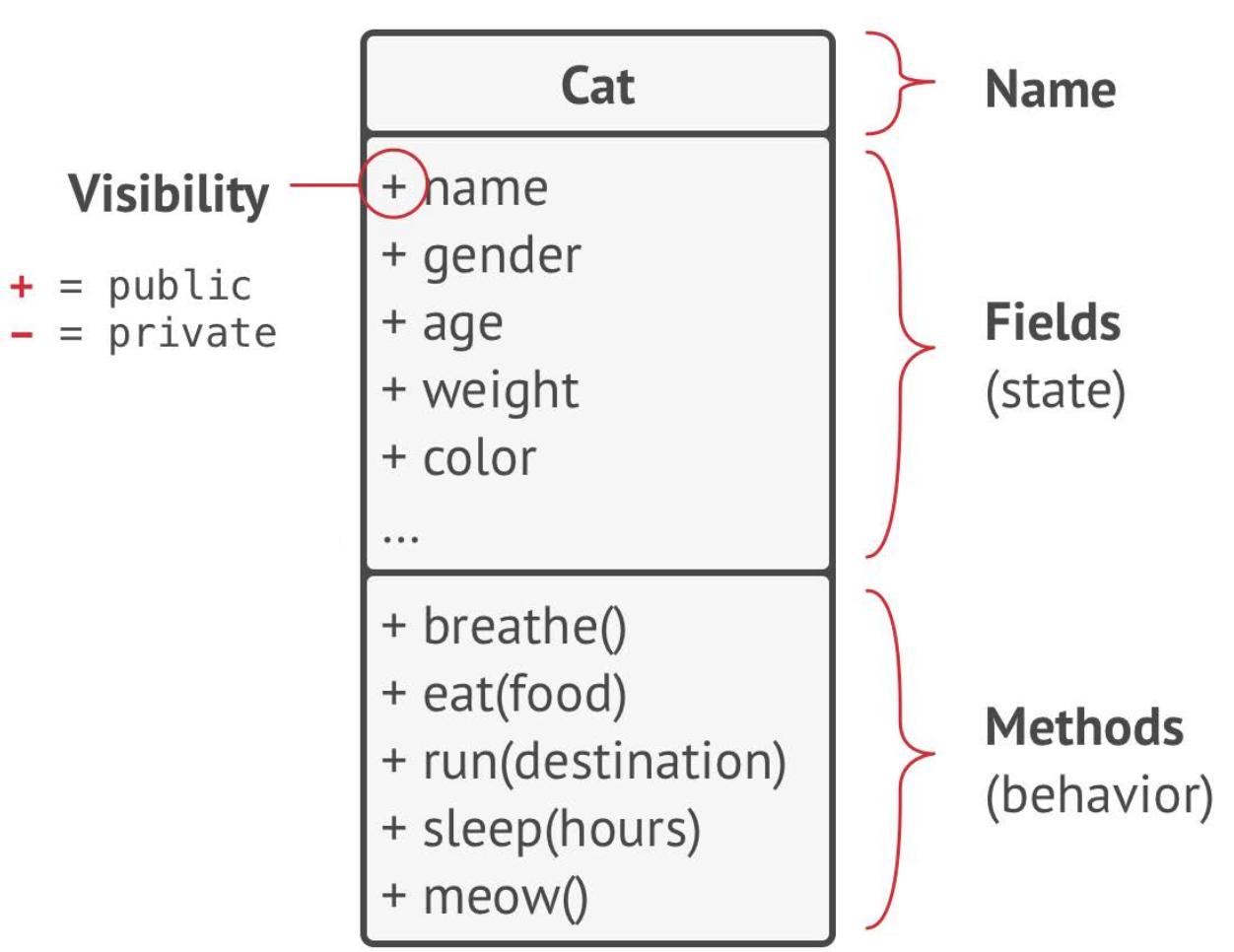


UML-Klassendiagramm

- Werkzeug aus der Unified Modeling Language (UML) um Klassen zu beschreiben

Aufbau

- Es dient dazu Klassen und deren Beziehungen zu beschreiben
  - Oben: Klassenname
  - Mitte: Attribute
  - Unten: Methoden
- Pfeile beschreiben die Beziehungen
- Enthält keine Ausprägungen der Klassen



Alle Bilder entnom. aus [Shvets 2019]

UML-Klassendiagramm

- Objekte sind Instanzen von Klassen → Klasse `Cat` mit Instanzen `Oscar` & `Luna`

Oscar: Cat

```
name    = "Oscar"
sex     = "male"
age     =  3
weight  =  7
color   =  brown
texture =  striped
```

Luna: Cat

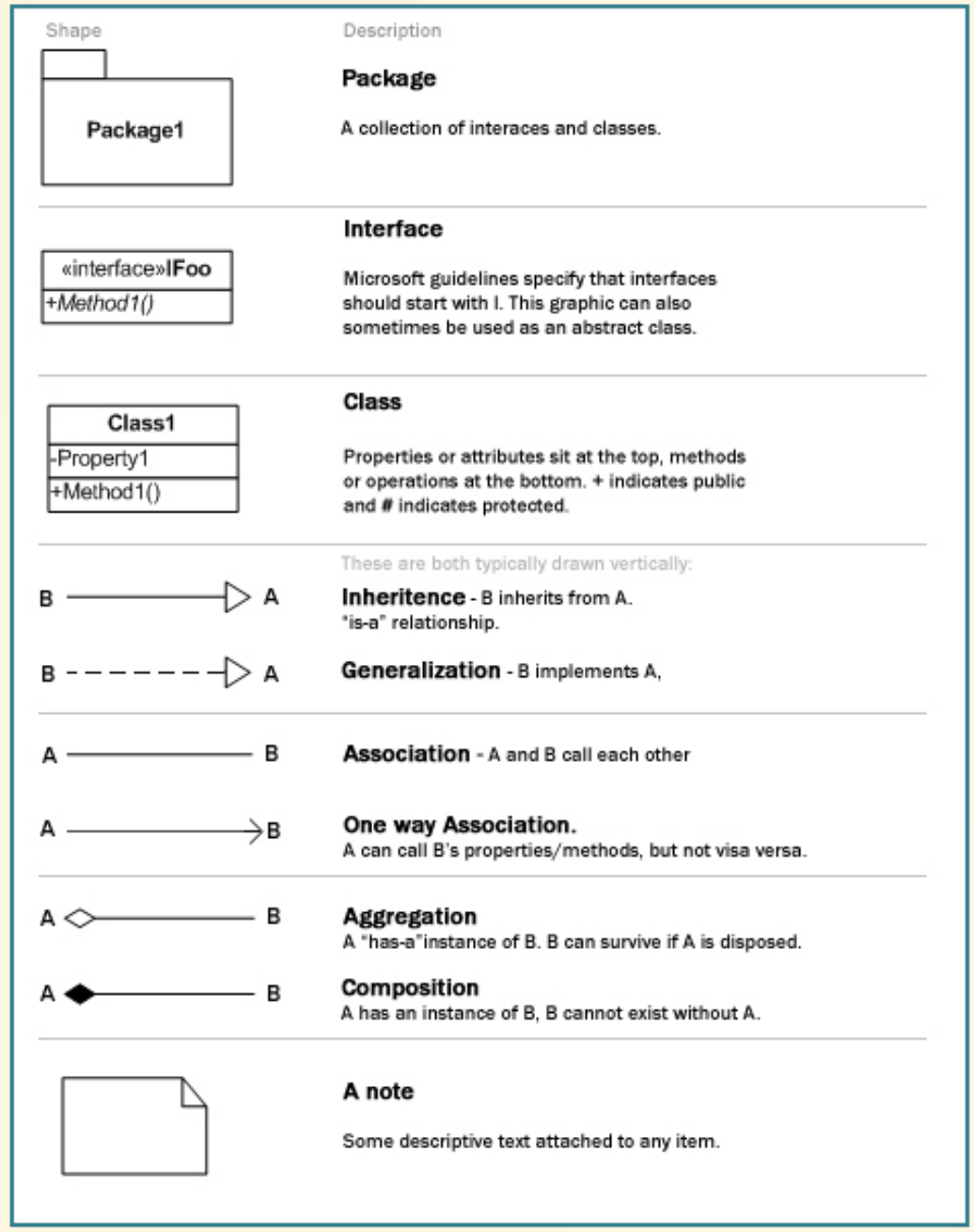
```
name    = "Luna"
sex     = "female"
age     =  2
weight  =  5
color   =  gray
texture =  plain
```

UML-Klassendiagramm

- Klassendiagramme enthalten noch weitere Komponenten → Beziehungen zw. Klassen

Beziehungen zwischen Klassen

- Klasse (Class)
- Vererbung (Inheritance)
- Assoziation: Beziehung zw. zwei oder mehr Klassen
- Aggregation: Spezielle Assoziation, die eine Zuordnung ausdrückt
- Komposition: Beziehung zw. einem Ganzen und seinen Teilen

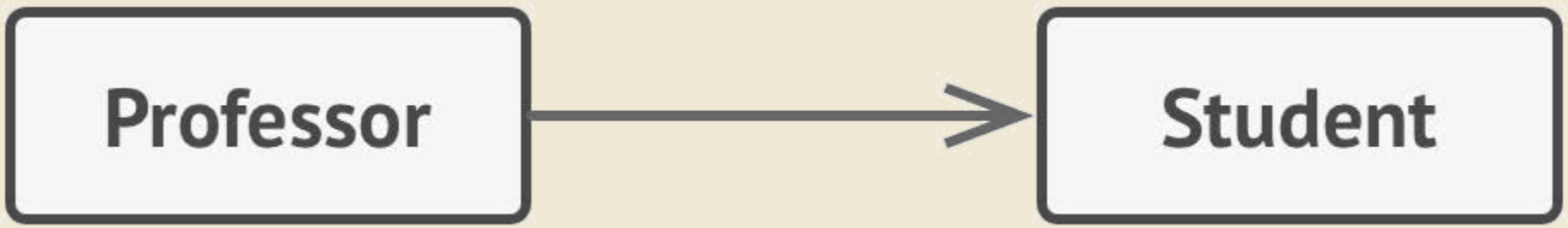


Assoziation (engl. association)

- Ein Objekt benutzt oder interagiert mit einem anderen
- dauerhafte, gerichtete Verbindung
- auch in beide Richtungen denkbar

Beispiel

- Klassen `Professor` und `Student`
- Ein Professor unterrichtet einen (bzw. mehrere) Studenten



Assoziation (engl. association)

- In Programmcode z.B. durch Zuweisung als Attribut realisiert

Beispiel - association.py

```
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

    def get_student_info(self):
        return f"Student: {self.name}, ID: {self.student_id}"

class Professor:
    def __init__(self, name):
        self.name = name
        self.students_taught = [] #List to hold student objects associated with the professor

    # Method to associate a student with a professor
    def add_student(self, student):
        self.students_taught.append(student)

    def get_teaching_info(self):
        students_info = [stu.get_student_info() for stu in self.students_taught]
        return f"Professor {self.name} teaches: {', '.join(students_info)}"

# Creating instances
prof = Professor("Dr. Johnson")
# Associating students with the professor
prof.add_student(Student("Alice", 101))
prof.add_student(Student("Bob", 102))

print(prof.get_teaching_info())
#> Professor Dr. Johnson teaches:Student: Alice, ID: 101, Student: Bob, ID: 102
```

Abhängigkeitsbeziehung (engl. dependency)

- schwächere Version → ohne permanente Verbindung zwischen den Objekten
- Ein Objekt akzeptiert z.B. ein anderes als Parameter einer Methode oder instanziiert dieses
- Liegt vor, wenn eine Änderung eines Objektes auch das andere Objekt verändert

Beispiel

- Klassen `Professor` und `Salary`
- `Professor` hängt von `Salary` ab → wenn sich das Gehalt ändert, ändert sich auch die Information des Professors

Abhängigkeitsbeziehung (engl. dependency)

- Sehr ähnlich zur Assoziation, aber schwächer

Beispiel - dependency.py

```python
class Salary:
    def __init__(self, amount: float) -> None:
        self.amount = amount
    def get_monthly_amount(self) -> float:
        return self.amount / 12

# Professor class depending on Salary class
class Professor:
    def __init__(self, name: str, account_balance: float) -> None:
        self.name = name
        self.account_balance = account_balance

    def receive_salary(self, salary: Salary) -> None:
        self.account_balance += salary.get_monthly_amount()

    def get_professor_info(self) -> str:
        return f"Professor {self.name} has a bank account balance of {self.account_balance}"

# Creating instances
salary_of_professor = Salary(60000) # Creating a Salary object

prof = Professor("Dr. Smith", 10000.0)   # Creating a Professor object
print(prof.get_professor_info()) #> Professor Dr. Smith has a bank account balance of 10000.0

prof.receive_salary(salary_of_professor) # Receiving salary

# Accessing professor information
print(prof.get_professor_info())#> Professor Dr. Smith has a bank account balance of 15000.0
```

Aggregation und Komposition

Komposition

- "Whole-Part"-Beziehung → Objekt besteht aus einem (oder mehreren) anderen Objekten
- `University` kann nicht ohne die dazugehörigen `Departments` existieren → wenn `University`-Objekt gelöscht wird, werden auch die `Departments` gelöscht

Aggregation

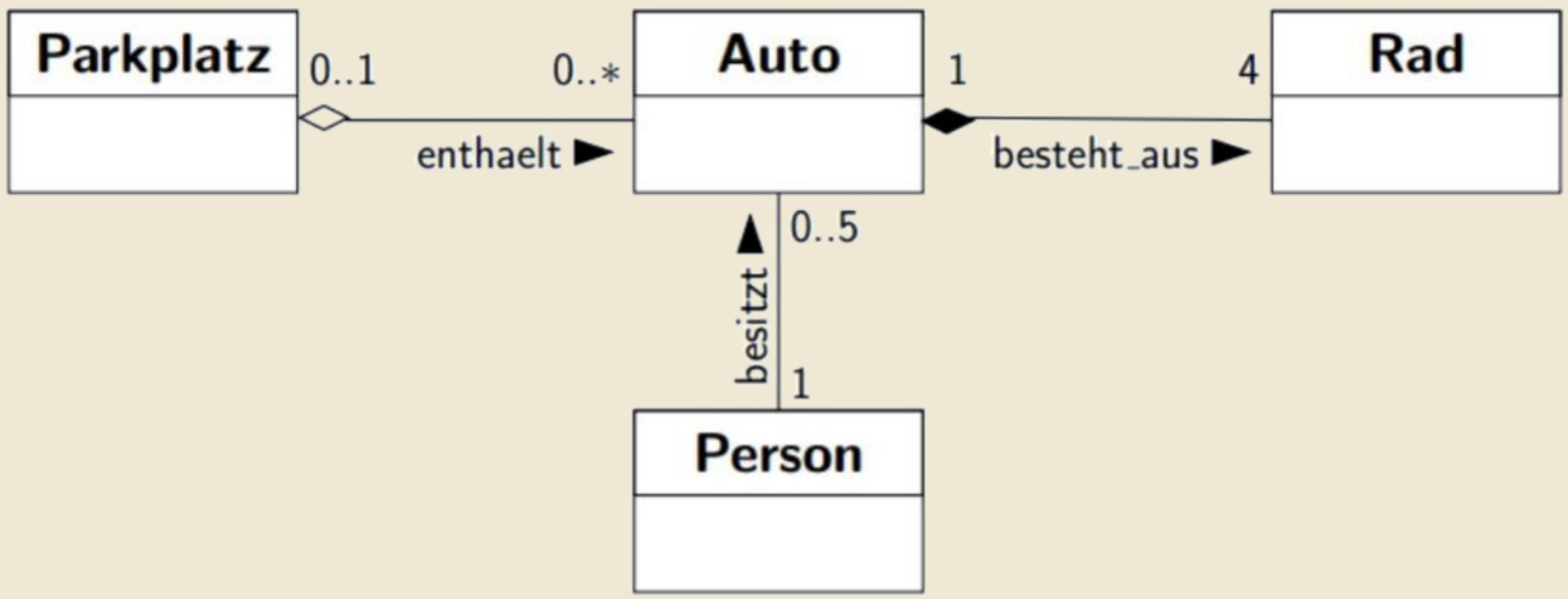
- Weniger starke Beziehung als Komposition
- `Professor` existiert auch ohne `Department` → kann auch an anderer `University` mit anderen `Departments` lehren

Multiplizitäten

Angabe von Multiplizitäten

Multiplizität	Bedeutung
1	genau ein
+	viele, kein oder mehr, optional
1..*	ein oder mehr
0..1	kein oder ein, optional
m..n	m bis n
m..*	m bis unendlich
m	genau m

Beispiel

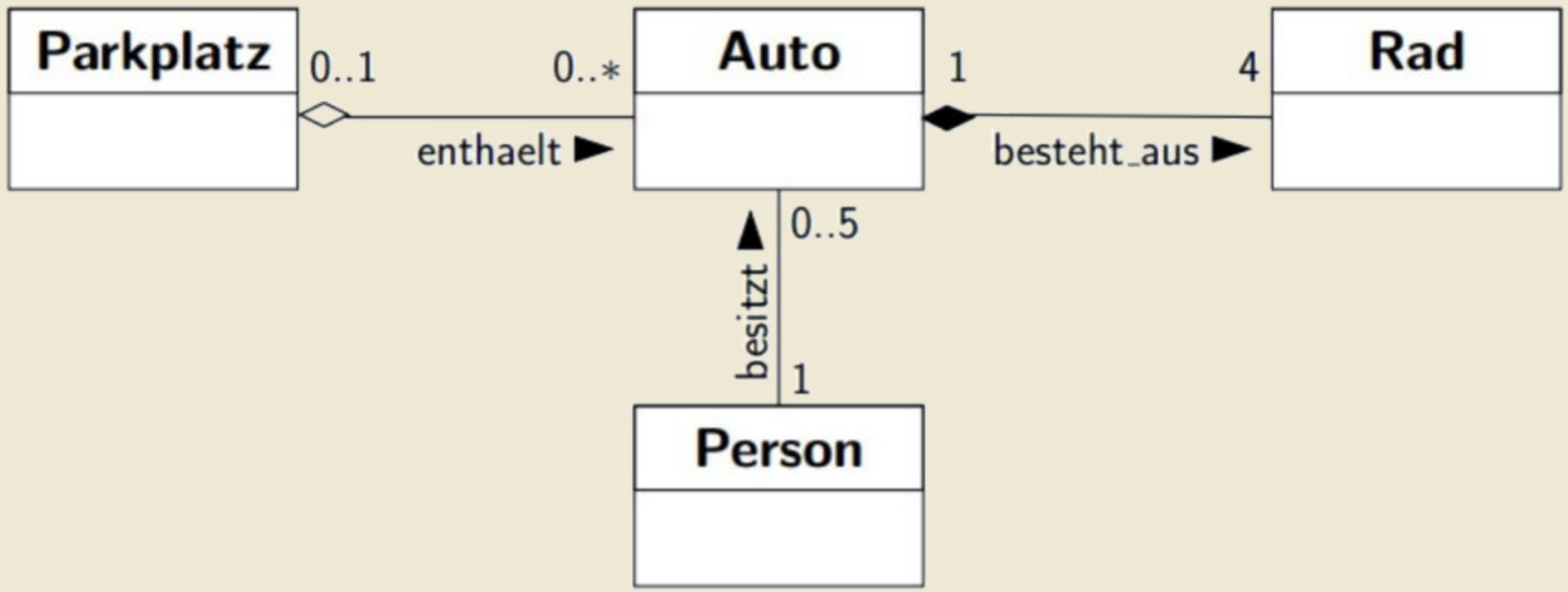


Multiplizitäten

"Lesen" von Multiplizitäten

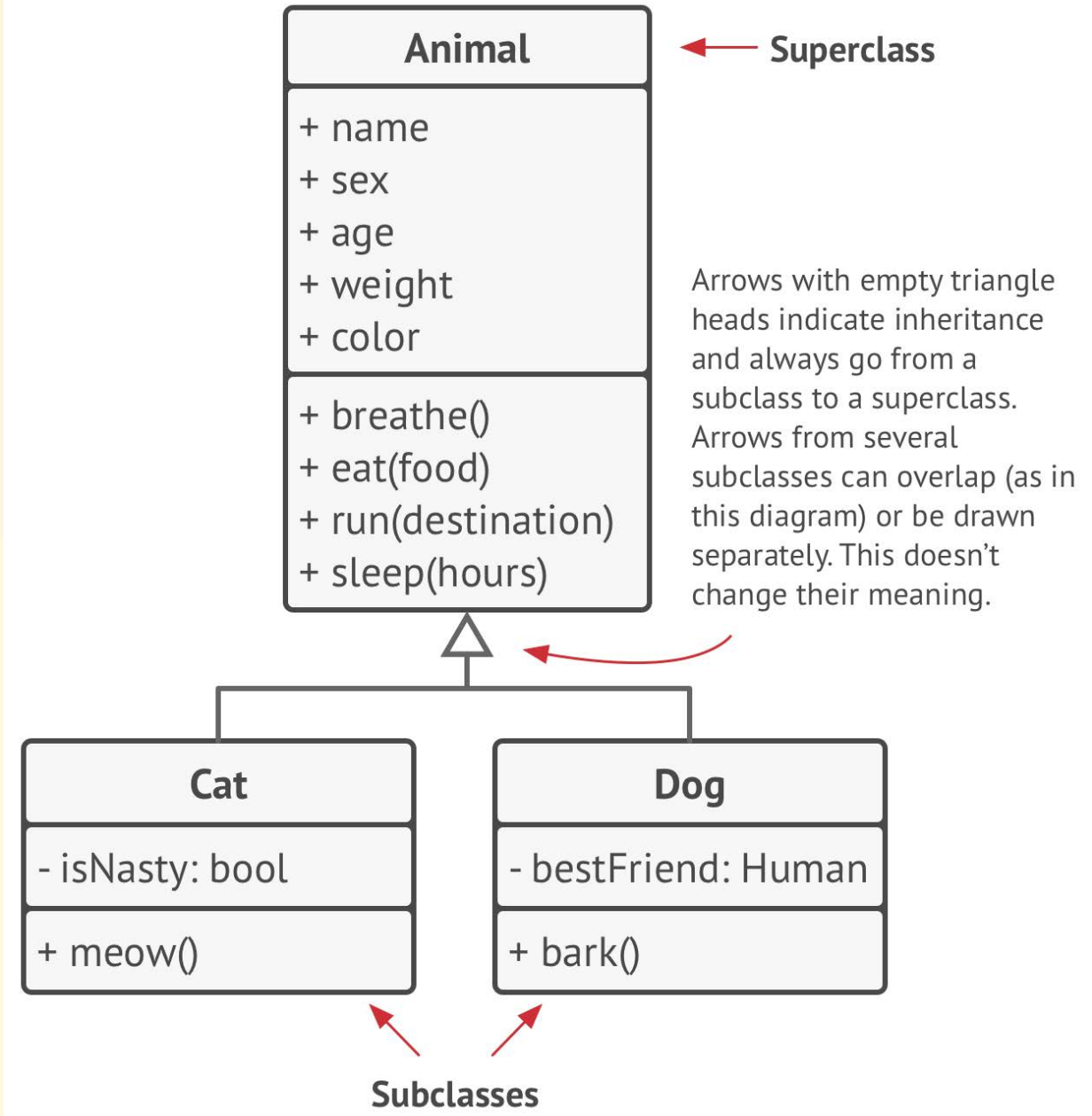
- Die Leserichtung *von der Klasse weg*:
  - *"Genau vier Räder gehören zu genau einem Auto!"*
  - *"Ein Parkplatz enthält null bis unendlich viele Autos!"*
  - *"Ein Auto steht auf einem oder keinem Parkplatz!"*

Beispiel



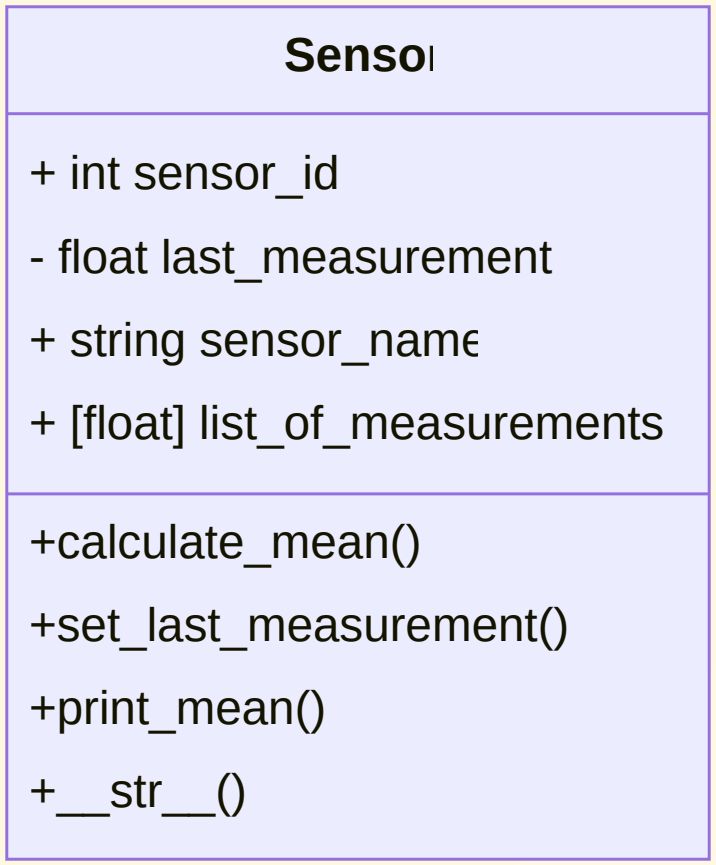
Vererbung

- Attribute und Methoden einer (Parent-/Super-)Klasse an (Child-/Sub-)Klassen weitergegeben
- Nur neu Attribute/Methoden müssen implementiert werden
- Im UML-Klassendiagramm wird die Vererbung mit meinem im leeren Dreieck endenden Pfeil dargestellt



Vererbung

- Wir wollen Sensoren modellieren bzw. beschreiben → Klasse `Sensor`
- Abhängig von der zu messenden Größe gibt es verschiedene Sensoren → nicht zu verwechseln mit verschienden Instanzen eines Sensortyps
- Jeder Typ kann unterschiedliche relevante Attribute und Methoden haben



Vererbung

- Klasse `Sensor` fungiert als Basisklasse → `HeartRateSensor` und `PowerMeter` erben von dieser
- Ein Objekt der Klasse `HeartRateSensor` bzw. `PowerMeter` verfügt über alle Methoden und Attribute der Klasse `Sensor`
- In Python keine Unterscheidung zwischen private, protected und public Vererbung

Beispiel - `inheritance.py`

```python
class Sensor():                                  # Klasse Device wird beschrieben
    def __init__(self, id, current_value = 0.0): # Bauplan der Klasse mit allen Attributen
        self.id = id
        self.current_value = current_value

    def get_id(self):                            # Methode der Klasse
        print(f"Hi, Ich bin {self.id}")

class HeartRateSensor(Sensor):                   # Definition einer Child-Klasse
    def __init__(self, id, current_value = 0.0):
        super().__init__(id, current_value)

    def calc_heart_rate_variability():
        #[...]
class PowerMeter(Sensor):                        # Definition einer Child-Klasse
    def __init__(self, id, current_value = 0.0):
        super().__init__(id, current_value)

    def calc_normalized_power():
        #[...]
```

✍️ Aufgabe

- Zeichnen Sie ein UML-Klassendiagramm zum vorangegangen Python Code der Sensoren

Tipp: `Mermaid Diagrams`

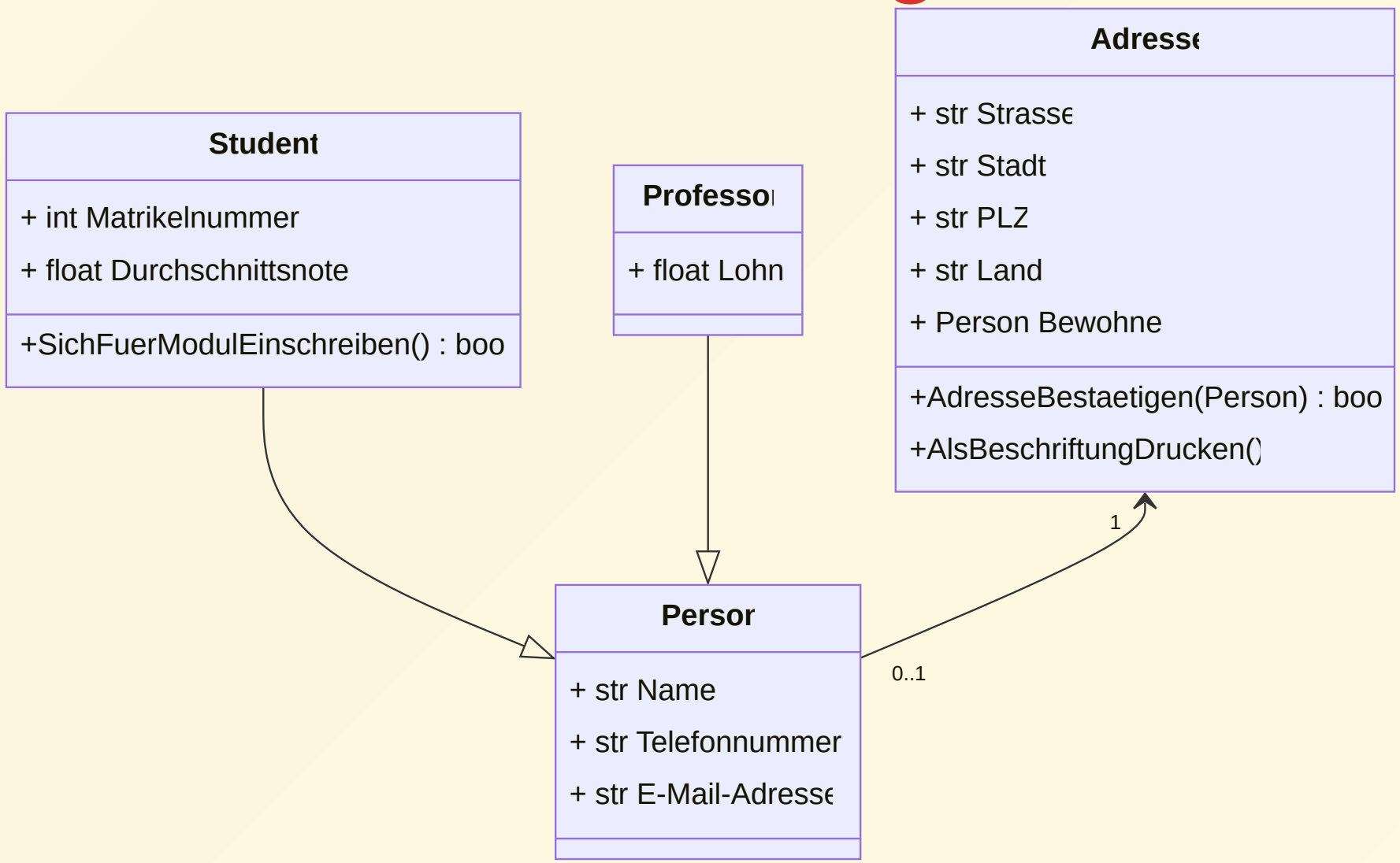
- Wir können UML-Diagramme mit gewissen Werkzeugen einfach erstellen
- Markup-Language zur Erstellung vieler Diagramme (UML-Klassendiagramm, Entity Relationship, Gantt-Chart)
- Grafiken können in verschiedenen Formaten erzeugt werden
- Änderungen für jeden online möglich

✍️ Aufgabe

Gegeben ist der folgende Sachverhalt:

- Jede Person hat einen Namen, eine Telefonnummer und E-Mail.
- Jede Adresse wird von nur einer Person bewohnt. Es kann aber sein, dass einige Adressen nicht bewohnt sind.
- Den Adressen sind je eine Strasse, eine Stadt, eine PLZ und ein Land zugeteilt.
- Adressen können als Wohnsitz einer Person bestätigt werden und als Beschriftung für Postversand gedruckt werden.
- Es gibt zwei Sorten von Personen: Student, welcher sich für ein Modul einschreiben kann und Professor, welcher einen Lohn hat.
- Der Student besitzt eine Matrikelnummer und eine Durchschnittsnote.
- Modellieren Sie diesen Sachverhalt mit einem UML Klassendiagramm. Beachten Sie die Assoziation (bewohnt) zwischen Person und Adresse und deren Multiziplität

✍️ Lösung

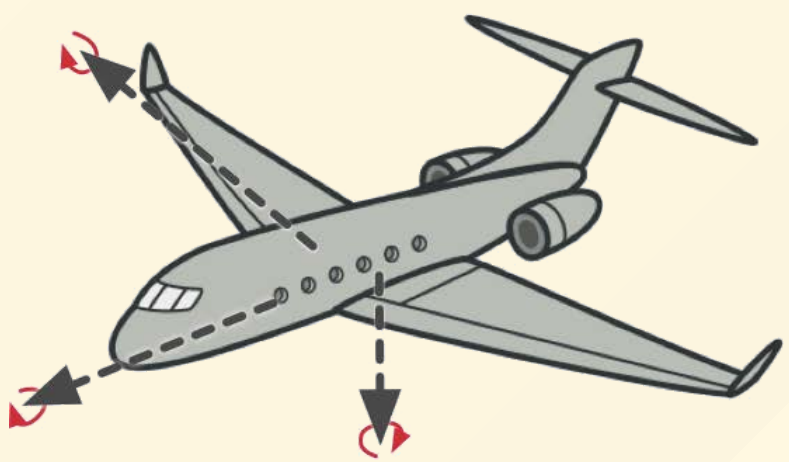

Student
+ int Matrikelnummer
+ float Durchschnittsnote
+SichFuerModulEinschreiben() : boo
Professo
+ float Lohn
Adresse
+ str Strasse
+ str Stadt
+ str PLZ
+ str Land
+ Person Bewohne
+AdresseBestaetigen(Person) : boo
+AlsBeschriftungDrucken()
Persor
+ str Name
+ str Telefonnummer
+ str E-Mail-Adresse
1
0..1

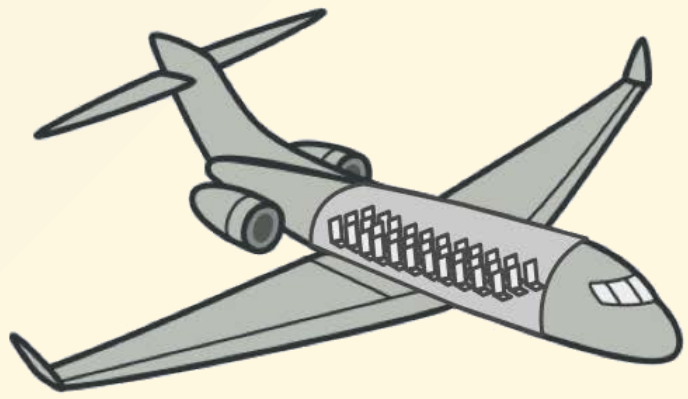
Prinzipien der objektorientierten Programmierung

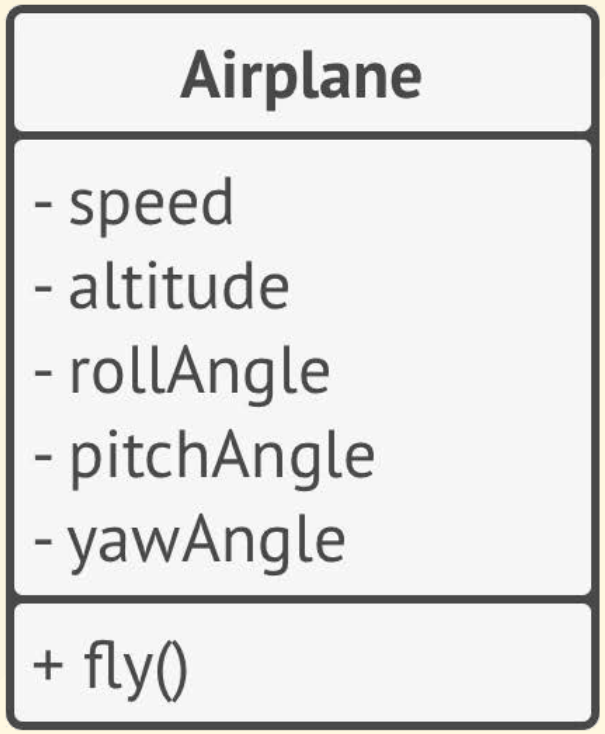
- In der objektorientierten Programmierung wollen wir einige Prinzipien beachten um sinnvolle Klassen und Vererbungen zu erstellen → diese nutzen alle die folgenden Konzepte:
  - Abstraktion
  - Kapselung
  - Vererbung (von abstrakten Klassen)
  - Polymorphismus
- Es existiert eine Vielzahl von Prinzipien die beachtet werden können/sollen → Überlick auf Wikipedia

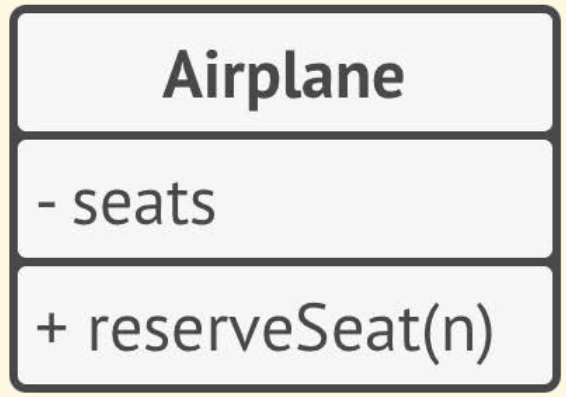
Abstraktion

- Klassen sind Modelle der realen Welt → Nachbildung nur so genau wie nötig
- Wird direkt vom Anwendungsfall diktiert → Flugzeug: dynamische Simulation des Flugverhaltens vs. Ticket-Buchungssystem



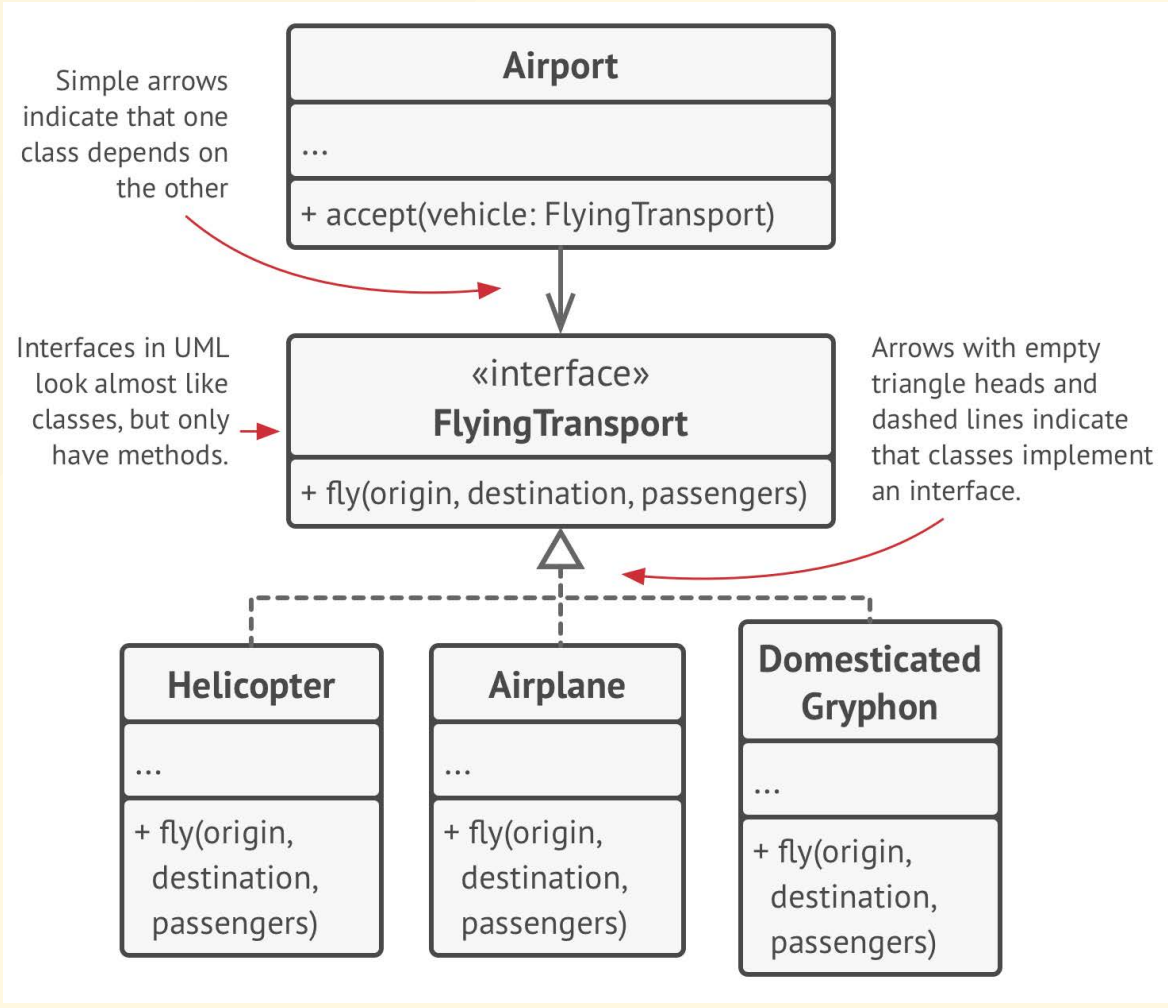






Kapselung

- Nur die Attribute und Methoden, die wirklich zur Interaktion nötig sind, werden nach außen zur Verfügung gestellt (vgl. Sichtbarkeiten)
- Interfaces sind wie Klassen, die die Interaktions-Möglichkeiten (Methoden) anbieten
- Python nutzt hierzu sogenannte `Abstract Base Classes (ABCs)`



Kapselung

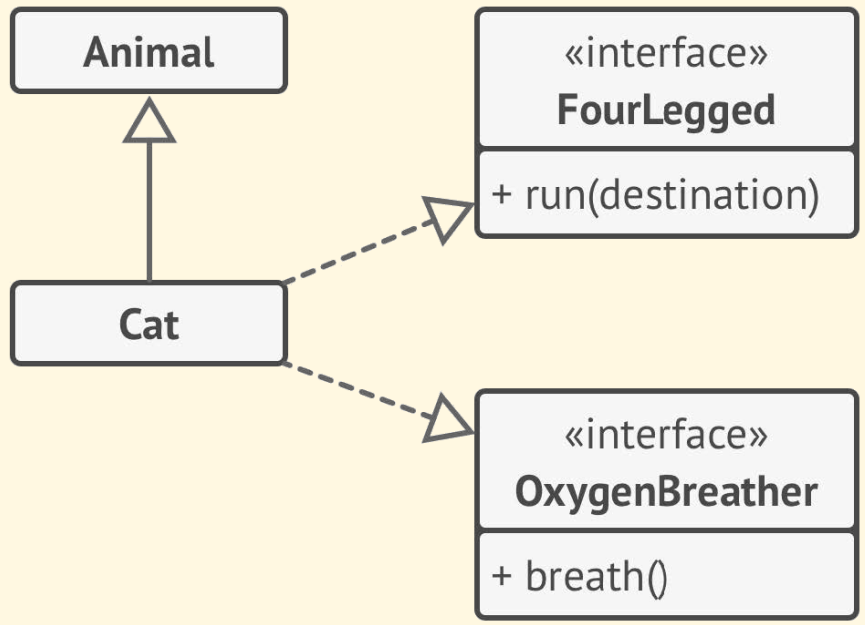
- Interface-Klassen erben von `ABC` → stellen nur Methoden zur Verfügung
- Methoden in Interface-Klassen sind abstrakt → müssen in erbenden Klassen implementiert werden → mit Dekorator `@abstractmethod` versehen

Beispiel - `abc_example.py`

```python
from abc import ABC, abstractmethod

# Abstract Class
class FlyingTransport(ABC):
    @abstractmethod
    def fly(self, origin, destination, passengers):
        pass # Abstract Method has no implementation!

# Non-Abstract Class
class Helicopter(FlyingTransport):
    def fly(self, origin, destination, passengers):
        print("Helicopter flying")

# Non-Abstract Class
class Airplane(FlyingTransport):
    def fly(self, origin, destination, passengers):
        print("Airplane flying")

helicopter1 = Helicopter()
airplane1 = Airplane()

helicopter1.fly("INN", "QOJ", 2) #> Helicopter flying
airplane1.fly("INN", "BER", 200) #> Airplane flying
```

Vererbung

- 🔁 ermöglicht es neue Klassen basierend auf existierenden zu erstellen → vermeidet mehrfaches Schreiben von Code
- → Sub-Klassen das selbe Interface wie die Super-Klasse
- `Python` erlaubt Vererbung von mehreren Klassen → Multiple Inheritance
- Es müssen alle abstrakten Methoden des Interfaces implementiert werden, auch wenn diese nicht benutzt werden



Vererbung

- `Cat` erbt von den drei Klassen `Animal`, `FourLegged` und `OxygenBreather` → letztere beide sind Interfaces
- Abstrakte Methoden der Interfaces müssen implementiert werden, Methoden aus `Animal` sind direkt nutzbar

Beispiel - `abc_cat_dog.py`

```python
class Animal():
    def __init__(self, name):
        self.name = name
    def eat(self):
        print(f"Animal {self.name} eats")

class FourLegged(ABC):
    @abstractmethod
    def run(self, destination):
        pass

class OxygenBreather(ABC):
    @abstractmethod
    def breath(self):
        pass

class Cat(Animal, FourLegged, OxygenBreather):
    def __init__(self, name):
        super().__init__(name)

    def run(self, destination):
        print(f"Cat {self.name} runs to {destination}")

    def breath(self):
        print(f"Cat {self.name} breaths")
```

Vererbung - Konstruktoren

- Wie auch schon in C++ müssen wir in Python bei Vererbungen auf die Konstruktoren achten
- Wird der Konstruktor einer Sub-Klasse angepasst, so muss dieser nicht komplett neu geschrieben werden
- Es ist auch möglich den Konstruktor der Super-Klasse aufzurufen und nur die zusätzlichen Attribute zu ergänzen
- `super()` Funktion erlaubt Zugriff auf Methoden der Superklasse

Beispiel - inherit_constructor.py

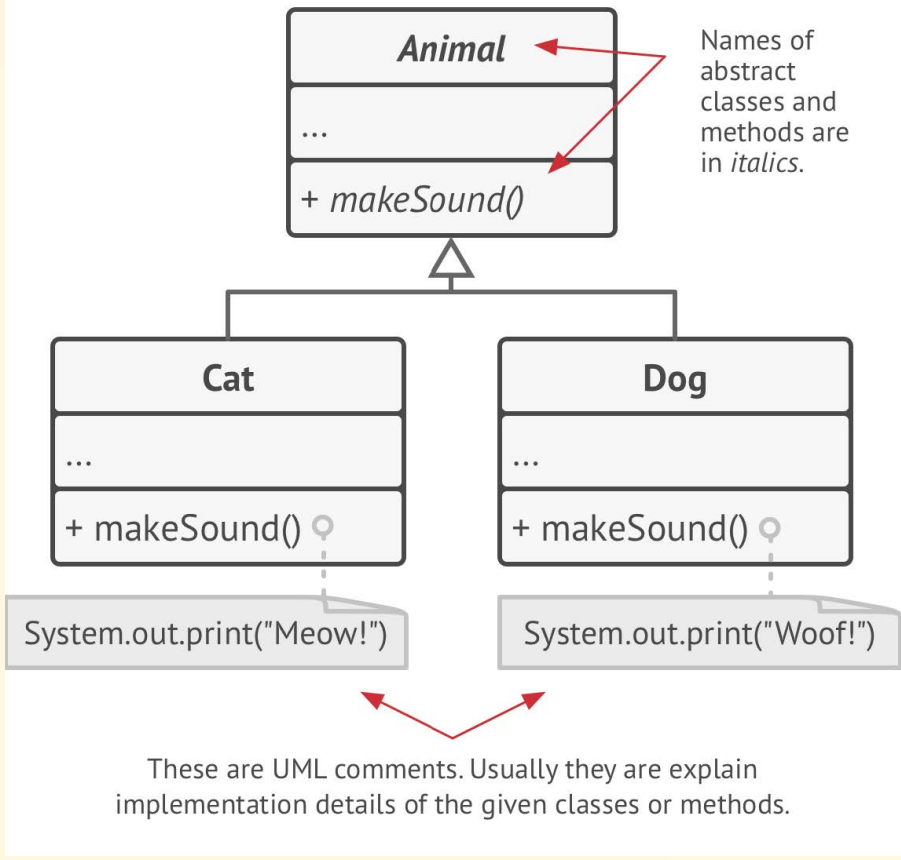
```
class Vehicle():
    def __init__(self, name, max_speed):
        self.name = name
        self.max_speed = max_speed

class Hovercraft(Vehicle):
    def __init__(self, name, max_speed, prop_size):
        super().__init__(name, max_speed) #calls super constructor
        self.prop_size = prop_size

hc1 = Hovercraft("SR.N4 Mk II", 130, 3.5)
```

Polymorphismus

- In Python deutlich einfacher als in C++
- Gleichnamige Methoden können bei unterschiedlichen Klassen verschieden implementiert sein

Beispiel - polymorphism.py

```
class FlyingTransport(ABC):
    @abstractmethod
    def fly(self, origin, destination, passengers):
        pass # Abstract Method has no implementation!

# Non-Abstract Class
class Helicopter(FlyingTransport):
    def fly(self, origin, destination, passengers):
        print("Helicopter flying")

# Non-Abstract Class
class Airplane(FlyingTransport):
    def fly(self, origin, destination, passengers):
        print("Airplane flying")

# Could make use of typ hing List[FlyingTransport]
flying_vehicles = [Airplane(), Helicopter()]
for vehicle in flying_vehicles:
    vehicle.fly("INN", "BER", 5)

#> Airplane flying
#> Helicopter flying
```



✍️ Aufgabe

- Wir wollen einen Roboter entwickeln, der sowohl fahren als auch fliegen kann → beim Zustandswechsel wird eine Statusmeldung ausgegeben
- Erstellen Sie dazu ein UML-Klassen-Diagramm, welches mindestens zwei abstrakte Base Classes (`Driveable` & `Flyable`) enthält.
- Logischer Weise kann der Roboter nur fahren, wenn er gerade nicht fliegt und umgekehrt. Den Wechsel zwischen den Zuständen definieren wir über eine Methode `change_status()`. Schreiben Sie auch den Python Code dazu und Kapseln Sie alle Methoden und Attribute, die nicht zur Interaktion benötigt werden.

Musterlösung - `flying_robot.py`

- Klassen `Driveable`, `Flyable` als Basisklassen
- `Robot` als erbende Klasse

✍️ 🤓 Aufgabe

- Um das Verständnis für die OOP weiter zu vertiefen kann das hier verlinkte Notebook bearbeitet werden → Jupyter Notebook
- Im Appendix "01_A1_Python_Grundlagen_ABCs_und_versteckte_Methoden" finden Sie weitere Informationen zu Abstract Base Classes und versteckten (`__<name>`) Methoden

🏆 Hausaufgabe

- Erweitern Sie das Grundgerüst an Klassen zur Datenverarbeitung aus der letzten Hausübung durch eine sinnvolle Klassenhierarchie
- Anhand des Klassendiagramms im Jupyternotebook sollen die Klassen `DataProcessor`, `DataProcessorTerminal`, `MovingAverageProcessor` und `RMSEProcessor` implementiert werden. Beachten Sie dabei folgende Punkte:
  - `DataProcessor` & `DataProcessorTerminal` sollen abstrakte Klassen sein
  - Integrieren Sie die bestehenden Klassen `MovingAverageProcessor` und `RMSEProcessor` in die neue Klassenhierarchie
  - Erstellen Sie die neue Klasse `MAPEProcessor` die den *Mean Absolute Percentage Error* berechnet