Herzlich Willkommen

Workshop:

Einführung in die OOPM am Beispiel mit Computerspielen

Irina Querbach

Python

Was ist OOPM?

- Programmierparadigma
- *O*bjekt*o*rientiertes *P*rogrammieren und *M*odellieren
- Modellierung der Welt als Objekte (und Klassen) mit Beziehungen
- OOPM vs. imperativem Programmieren („algorithmisch“)
- Vertreter der OO-Programmiersprachen: Java, Python, C++…
- Modellierungssprache: UML (*U*nified *M*odelling *L*anguage)

Computerspiele

- VOR der Modellierung: fachkundiges Hintergrundwissen bei anderen den Schülern nicht geläufigen Prozessen
- Computerspiele bieten einen Themenbereich, in dem sich (die meisten Schüler) auskennen
- Intrinsische Motivation, da aus der Erfahrungswelt und Interessensbereich der Schüler
- In CS finden sich i.d.R. sehr komplexe Programmabläufe → ohne OO (fast) unmöglich
- OO-Konzepte sind in simulierten Welten (=Modellen!) offensichtlich
- Differenzierungsmöglichkeit in der (selbstbestimmten) Wahl der Algorithmik
- Besonders geeignete Spiele: Rollen- und Simulationsspiele

(Die Sims, Minecraft, Hogwarts Legacy, Animal Crossing,…)

Die wichtigsten Konzepte aus der OOPM

- Objekte und Klassen: Spieler, Gegner und Gegenstände
- Attribute und Methoden: Lebensbalken, Ausstattung und Fähigkeiten
- Besondere Methoden(Getter, Setter und Konstruktor): Herstellung, Heilung und „Cheating“
- Beziehungen mit anderen Objekten:
  - (gewöhnliche) Assoziationen (Gegner, Gegenstände, Umgebung und andere Spieler…)
  - Aggregation (Inventar u.Ä.)
  - Vererbung (verschiedene „Gegnermodelle“, NPCs und Charaktere)
- Spielablauf testen/simulieren

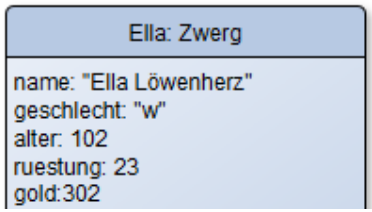
Objekt-und Klassenkarten in UML

Objekte mit gleichen Attributen und Methoden fasst man zu Klassen zusammen.

Ein Objekt besitzt einen eindeutigen Namen und konkrete Attributwerte.

Begrifflichkeiten:
Objekte, Klassen, Attribute und Methoden

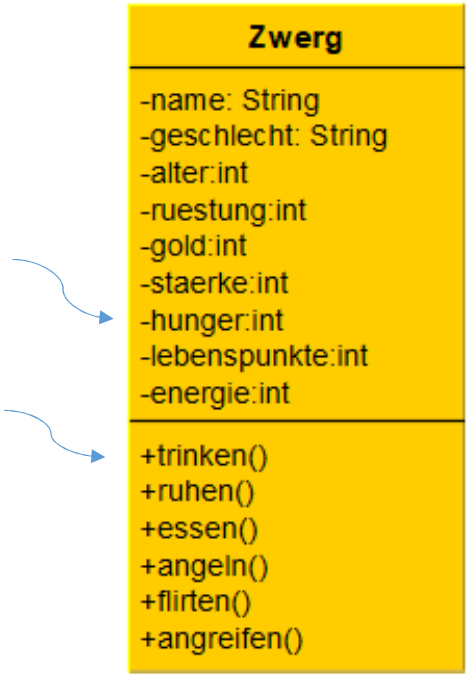
- **Objekte** sind konkrete Exemplare mit Attributwerten…



- **Klassen** sind Baupläne für Objekte…

- **Attribute** sind die Eigenschaften der Objekte…

- **Methoden** sind die Fähigkeiten der Objekte…
  (Algorithmen)



Aufgabe 1:
Objektkarte & Klassenkarte zum „Troll“ modellieren

<u>Anforderung:</u>
mind. 4 Attribute
mind. 4 Methoden

Theo: Troll
name: "Theo Thunfisch" staerke: 350 lebenspunkte:700 gold: 1502 wach:True
angreifen() verteidigen() schreien() schlafen()

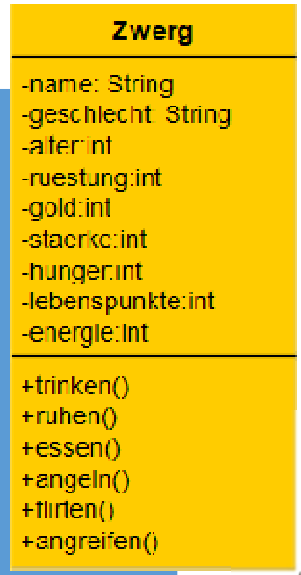
Troll
-name: String -staerke: int -lebenspunkte:int -gold:int -wach:bool
+angreifen() +verteidigen() +schreien() +schlafen()

Konstruktor

Konstruktor als „Ersteller und Initialisierer“ der Attributwerte eines Zwerg-Objektes direkt bei der Erstellung (Startwerte setzen!)

Python:

```
class Zwerg:
    def __init__(self, name, geschlecht, alter, ruestung, staerke):
        # Initialisierung der Attribute
        self.name = name
        self.geschlecht = geschlecht
        self.alter= alter
        self.ruestung=ruestung
        self.gold=0
        self.staerke=staerke
        self.hunger=0
        self.lebenspunkte=1000
        self.energie=100
```



Aufgabe 2: Konstruktor für „Troll“ schreiben

Konstruktor

Konstruktor als „Ersteller und Initialisierer“ der Attributwerte eines Troll-Objektes direkt bei der Erstellung (Startwerte setzen!)

Python:

```
class Troll:
    def __init__(self, name, staerke):
        self.name = name
        self.staerke=staerke
        self.lebenspunkte=1000
        self.gold=0
        self.wach=True
```

Troll
-name: String -staerke: int -lebenspunkte:int -gold:int -wach:bool
+angreifen() +verteidigen() +schreien() +schlafen()

Methoden und Attribute im Zusammenspiel

Methoden „arbeiten“ mit und auf den Attributwerten.

D.h. der Algorithmus im Methodenrumpf wird in Abhängigkeit der Attributwerte definiert und/oder die Attributwerte werden durch die Ausführung des Algorithmus manipuliert.

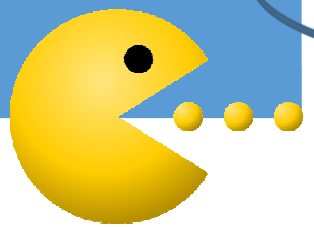
Beispiel in Python:

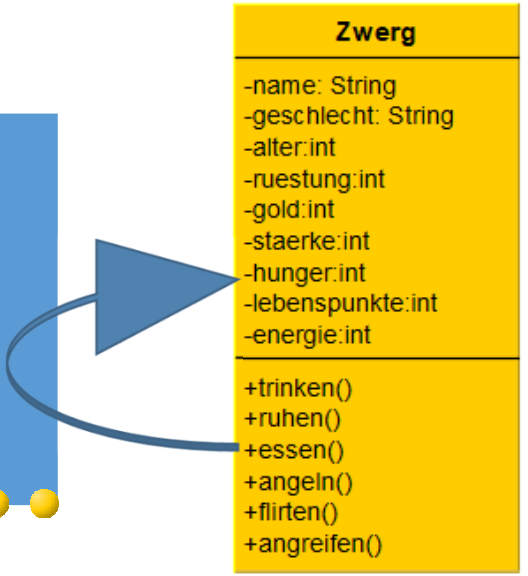
Zwerg:

```
...
def essen(self):
        self.hunger=self.hunger+10
        print("mmmmh")
...
```

Troll:

```
...
    def schlafen(self):
        self.wach=False
        print("zzzz")
...
```





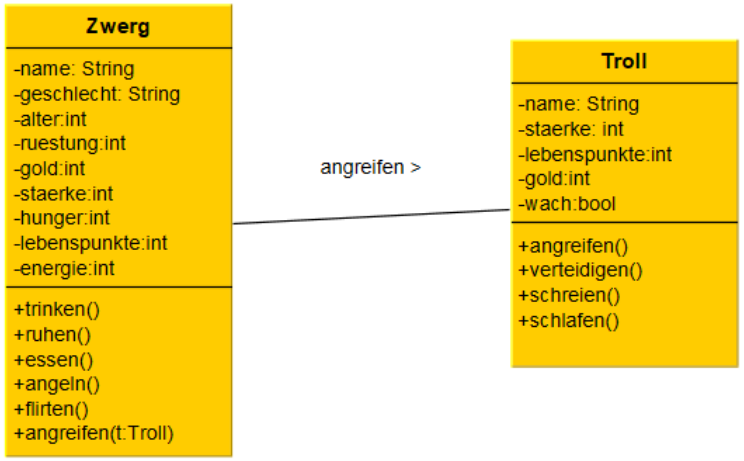
Erste Tests!

Python:

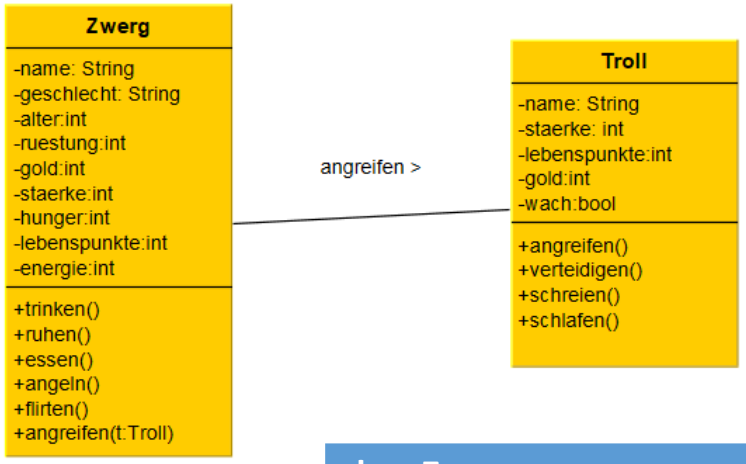
```
>>> ella = Zwerg("Ella", "w", 102, 23, 115)
>>> ella.essen()
mmmmh
>>> print(ella.hunger)
10
>>> ella.essen()
mmmmh
>>> print(ella.hunger)
20
>>>
```

```
>>> theo = Troll("Theo", 350)
>>> theo.schlafen()
zzzz
>>> print(theo.wach)
False
>>>
```

Beziehungen:
1. (einfache) Assoziation („kennt“-Beziehung)



Beziehungen:
1. (einfache) Assoziation („kennt“-Beziehung)




```
class Troll:
...
def verteidigen(self, angriff):
    if self.lebenspunkte>angriff:
        self.lebenspunkte = self.lebenspunkte-angriff
    else:
        self.lebenspunkte=0
    print("oje")
```

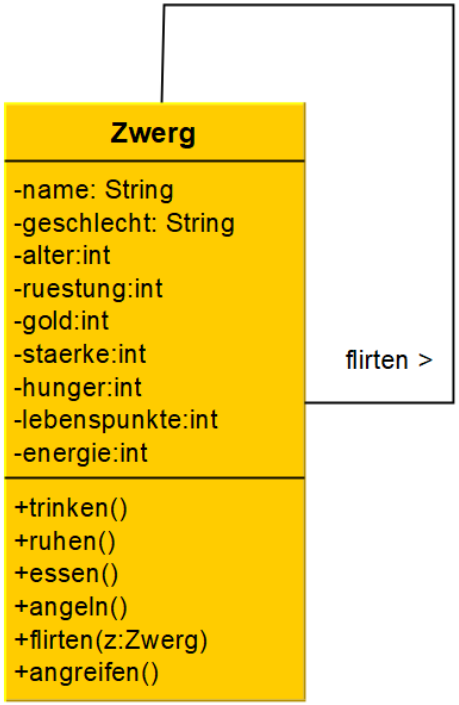
```
class Zwerg:
...

def angreifen(self, ptrollo):
    t.lebenspunkte=t.lebenspunkte-self.staerke
    print("zack")
```

```
class Zwerg:
...

def angreifen(self, ptrollo):
    #besser
    t.verteidigen(self.staerke)
    print("zack")
```

Beziehungen:
1. (einfache) Assoziation („kennt“-Beziehung)
<u>reflexiv</u>

```
def flirten(self, pzwergo):
    print("hello " + pzwergo.name)
    pzwergo.flirten(self)
```

```
def flirten(self, pzwergo, start):
    print("hello " + pzwergo.name)
    #besser
    if start == 0:
        pzwergo.flirten(self, 1)
```



Tests!

Python:

```
>>> ella = Zwerg("Ella", "w", 102, 23, 115)
>>> theo = Troll("Theo", 350)
>>> ella.angreifen(theo)
zack
>>> print(theo.lebenspunkte)
885
>>> ella.angreifen(theo)
zack
>>> ...
-35
```

```
...
#bessere Variante
>>> ella.angreifen(theo)
oje
zack
>>> print(theo.lebenspunkte)
885
>>> ...
0
```

Tests!

Python:

```
>>> ella = Zwerg("Ella", "w", 102, 23, 115)
>>> angi = Zwerg("Angi ", "w", 102, 23, 115)
>>> ella.flirten(angi)
hello Angi Löwenherz
hello Ella Löwenherz
hello Angi Löwenherz
...
hello Ella Löwenherz
hello Angi Löwenherz

Process ended with exit code 3221225725.
```

```
...
#bessere Variante
>>> ella.flirten(angi,0)
hello Angi
hello Ella
```

Aufgabe 3: Assoziation für die Methode schreien(ptrollo) der Klasse „Troll“ implementieren.

```
...
def schreien(self, ptrollo):
        print("whoaaa")
        ptrollo.angreifen()
        self.verteidigen(ptrollo.staerke)
...
```

```
>>> theo = Troll("Theo Thunfisch", 350)
>>> tunidor = Troll("Tunidor Tau", 550)
>>> theo.schreien(tunidor)
whoaaa
wupp
oje
>>> print(theo.lebenspunkte)
450
```

Das Geheimnisprinzip („no Cheating!“)

- **Geheimnisprinzip:** Attribute sind privat und Methoden öffentlich
- **Setter**-Methoden zum ändern mancher Spieler-Eigenschaften nachträglich bzw. während des Spiels unter den eigenen Bedingungen!
- **Getter**-Methoden um manche Attributwerte anderer Objekte abzufragen

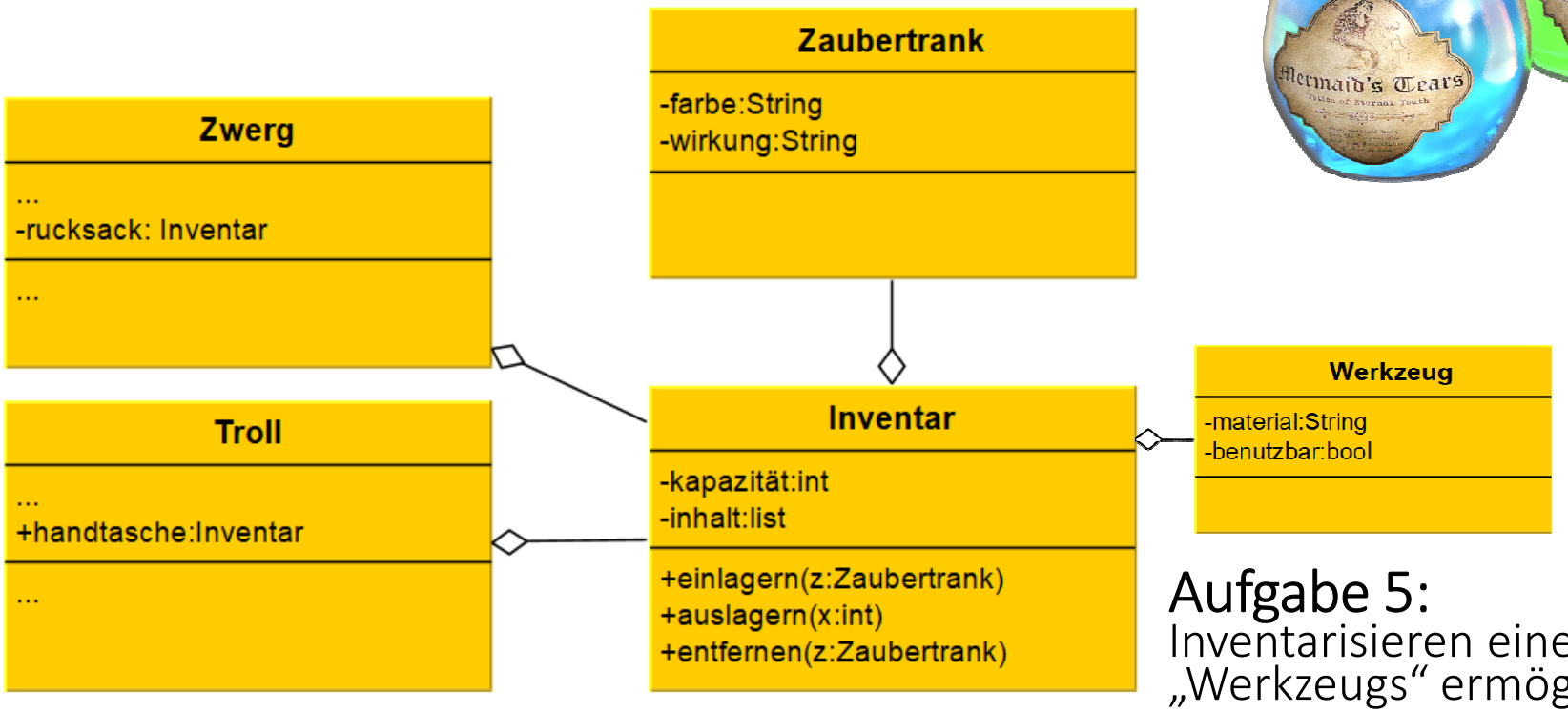
Python:

```
class Zwerg:
    def __init__(self, name, geschlecht, alter, ruestung, staerke):
        self.__name = name
…
        self.__staerke=staerke
        self.__hunger=0
        self.__lebenspunkte=1000
…
```

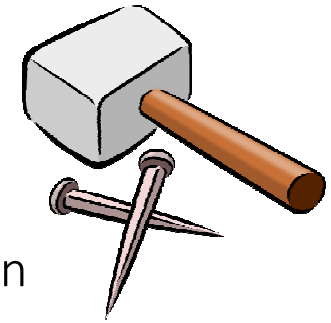
```
…
def get_staerke(self):
        return self.__staerke

…
    def set_lebenspunkte(self, lebenspunkte):
        if lebenspunkte>=0:
            self.__lebenspunkte=lebenspunkte
…
```

Aufgabe 4: Privatisierung der Attribute und Getter/Setter in der Klasse „Troll“ implementieren!

Beziehungen:
2. Aggregation(„hat“-Beziehung)



Aufgabe 5:
Inventarisieren eines „Werkzeugs“ ermöglichen



Python:

```
class Zaubertrank:
    def __init__(self, farbe, wirkung):
        self.__farbe=farbe
        self.__wirkung=wirkung

    def get_wirkung(self):
        return self.__wirkung
```

```
class Zwerg:
    def __init__(self,…):
        ….
        self.__rucksack=Inventar(50)
```

```
class Troll:
    def __init__(self, …):
        …
        self.__handtasche=Inventar(50)
```

```
class Inventar:
    def __init__(self, kapaz):
        self.__kapaz=kapaz
        self.__inhalt=[]

    def get_inhalt(self):
        return self.__inhalt

    def einlagern(self, pzaubertranko):
        self.inhalt.append(pzaubertranko)
        print(pzaubertranko.get_wirkung(), "eingelagert")

    def auslagern(self, x):
        if len(self.__inhalt) > x:
            self.__inhalt.pop(x)

    def entfernen(self, pzaubertranko):
        if pzaubertranko in self.__inhalt:
            self.__inhalt.remove(pzaubertranko)

    def drucken(self):
        print(self.__inhalt)
```

Tests!

Python:

```
ella = Zwerg("Ella", "w", 102, 23, 115)
angi = Zwerg("Angi ", "w", 102, 23, 115)
theo = Troll("Theo", 350)

heiltrank=Zaubertrank("gelb", "Heilung")
gift=Zaubertrank("grün", "Gift")
doping=Zaubertrank("blau", "Stärke")

ella.get_rucksack().einlagern(heiltrank)
ella.get_rucksack().einlagern(gift)

 angi.get_rucksack().einlagern(gift)
 angi.get_rucksack().einlagern(doping)

 theo.get_handtasche().einlagern(doping)
 theo.get_handtasche().einlagern(heiltrank)

 print("\n Ella: ")
 print(self.ella.get_rucksack())

print("\n Angi: ")
 print(self.angi.get_rucksack().drucken())



print("\n Theo: ")
print(self.theo.get_handtasche().drucken())
```

```
Heilung eingelagert
Gift eingelagert
Gift eingelagert
Stärke eingelagert
Stärke eingelagert
Heilung eingelagert



Ella:
<Inventar.Inventar object at 0x000001D2DB3D90C0>

 Angi:
[<Zaubertrank.Zaubertrank object at 0x000001D2DB3D9210>,
<Zaubertrank.Zaubertrank object at 0x000001D2DB3D8E50>]
None

 Theo:
[<Zaubertrank.Zaubertrank object at 0x000001D2DB3D8E50>,
<Zaubertrank.Zaubertrank object at 0x000001D2DB3D8D30>]
None
>>>
```

Interessantes:

- Eine Aggregation ist eine **spezielle Assoziation**, die den Zugriff auf das Objekt aus einer anderen Klasse vereinfacht.
- Es gibt noch eine stärkere „hat“-Beziehung, die man modellieren kann. Sie nennet sich **Komposition**. In UML ist dann die Raute schwarz gefüllt. Sie bindet den „Besitz“ so sehr an die „Besitzerklasse“, dass der Besitz ohne die Besitzerklasse nicht existieren kann.
- Das Inventar bietet viele Möglichkeiten um über Sortierverfahren, Suchalgorithmen und über Datenstrukturen zu sprechen!

Aufgabe 5:

Tierische Begleiter modellieren!

```
class Begleiter:
    def __init__(self, name, rasse, faehigkeit):
        self.__name=name
        self.__rasse=rasse
        self.__faehigkeit=faehigkeit

    def faehigkeit_anwenden(self):
        print ("Das kann ich:", self.__faehigkeit, "!!!!!")
```

```
class Zwerg:
    def __init__(self, name, geschlecht, alter, ruestung, staerke, begleiter):
        …
        self.__rucksack=Inventar(50)
        self.__freundchen= begleiter
```

```
tier1=Begleiter("Furi", "Drache", "Feuer speien")
tier2= Begleiter("Leuchti", "Stern", "scheinen")
ella = Zwerg("Ella", "w", 102, 23, 115, tier1)
angi = Zwerg("Angi ", "w", 102, 23, 115, tier2)
…
ella.get_freundchen().faehigkeit_anwenden()
angi.get_freundchen().faehigkeit_anwenden()
```

```
Das kann ich: Feuer speien !!!!!
Das kann ich: scheinen !!!!!
```

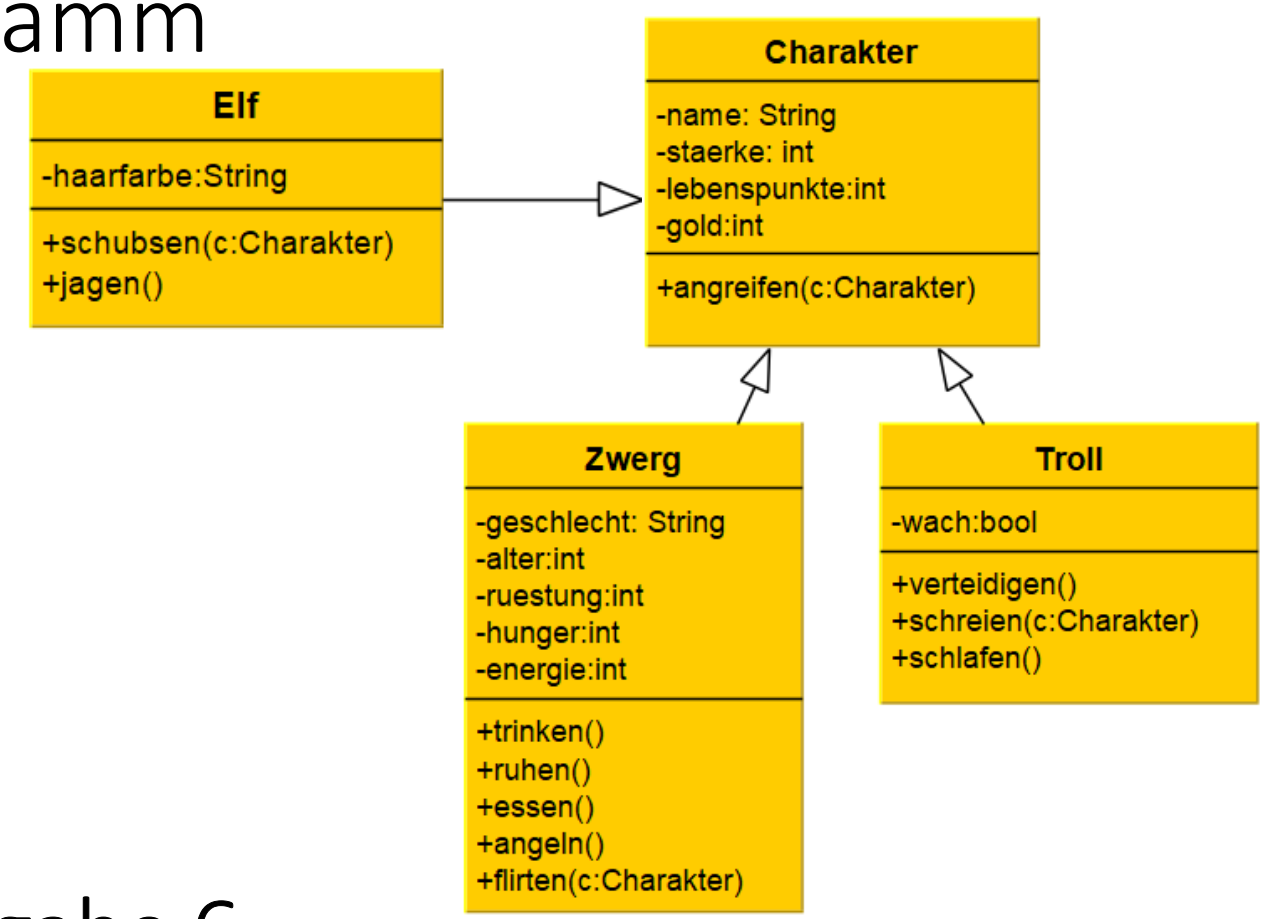
Beziehungen:
3. Vererbung/Spezialisierung („ist“-Beziehung)

Ideen:

- Zwerge dürfen auch mit Trollen flirten (und Trolle dürfen auch Zwerge anschreien)
- Doppelten Code in den Klassen Zwerg und Troll vermeiden

Die Unterklassen erben (ausnahmslos) alle Attribute und Methoden der Oberklasse! Methoden können überschrieben werden (Polymorphie).

Klassendiagramm



Aufgabe 6:

Integration des „Elf“ sinnvoll ins Klassendiagramm

Python:

```
class Character:
    def __init__(self, name, staerke):
        self._name = name
        self._staerke=staerke
        self._lebenspunkte=1000
        self._gold=0
        self._tasche=Inventar(50)

    def get_name(self):
        return self._name

    def get_staerke(self):
        return self._staerke

    def get_tasche(self):
        return self._tasche

    def get_lebenspunkte(self):
        return self._lebenspunkte

    def angreifen(self):
        print("wupp")
```

```
class Elf(Character):
    def __init__(self, name, staerke, haarfarbe):
        super().__init__(name, staerke)
        self.__haarfarbe=haarfarbe

    def schubsen(self, pcharactero):
        print("wumms")
        print(pcharactero.get_name(), "greift", self._name, "an !")
        pcharactero.angreifen()

    def jagen(self):
        print ("schwups")
```

```
>>> elfi=Elf("Oridor", 330, "grün")
>>> elfi2=Elf("Hubidos", 430, "gelbgrünlich")
>>> elfi.schubsen(elfi2)
wumms
Hubidos greift Oridor an!
Wupp
```

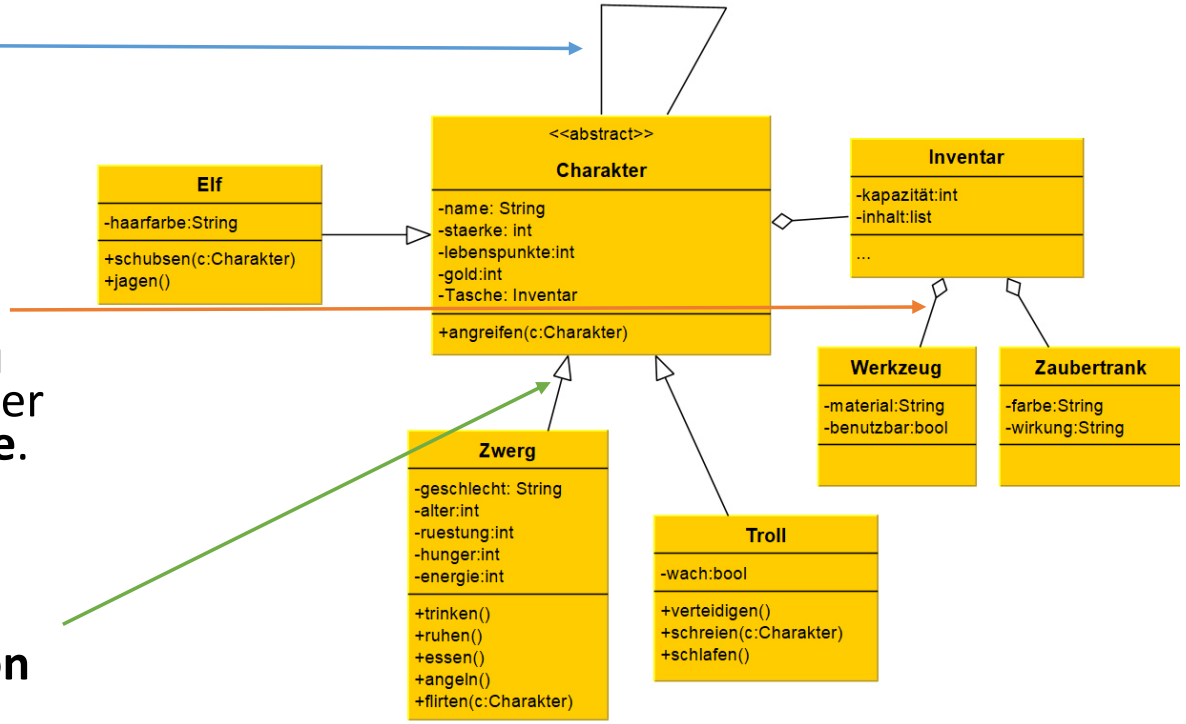
Fehler, wenn Attribute name private ist:
AttributeError: 'Elf' object has no attribute '_Elf__name

Interessantes:

- Die Unterklasse erbt stets **alle Attribute und alle Methoden** der Oberklasse (=Superklasse)
- Die Oberklasse nennt man **Generalisierung** und die Unterklasse **Spezialisierung**
- **Attribute der Oberklasse** sollten **protected** statt private sein
- Eine **Abstrakte Klasse** ist eine Klasse, von der keine Objekte erzeugt werden können. Andere Klassen können von der abstrakten Klasse allerdings erben
- Man nennt das Überschreiben der Oberklassenmethoden durch eigene (Unterklassen-)Methoden **Polymorphie**

Beziehungen Zusammenfassung

- **Kennt**-Beziehungen (einfache *Assoziation*en) zu einem Objekt werden nur in **Methoden** (z.B. als Parameter) implementiert.
- Eine **Hat**-Beziehung (*Aggregation*) zu einem Objekt implementiert man über die Definition innerhalb der **Attribute**.
- Eine **Ist**-Beziehung (Vererbung/*Spezialisierung*) ist insbesondere in der **Klassendefinition** und im (Klassen-)Konstruktor implementiert.



DANKE!

Quellen und Programme:

- alle Bilder von Pixabay inkl. Pixabay Inhaltslizenz
- Programm Thonny, University of Tartu (Thonny, Python IDE for beginners)
- Programm yEd von yWorks GmbH (www.yWorks.com)