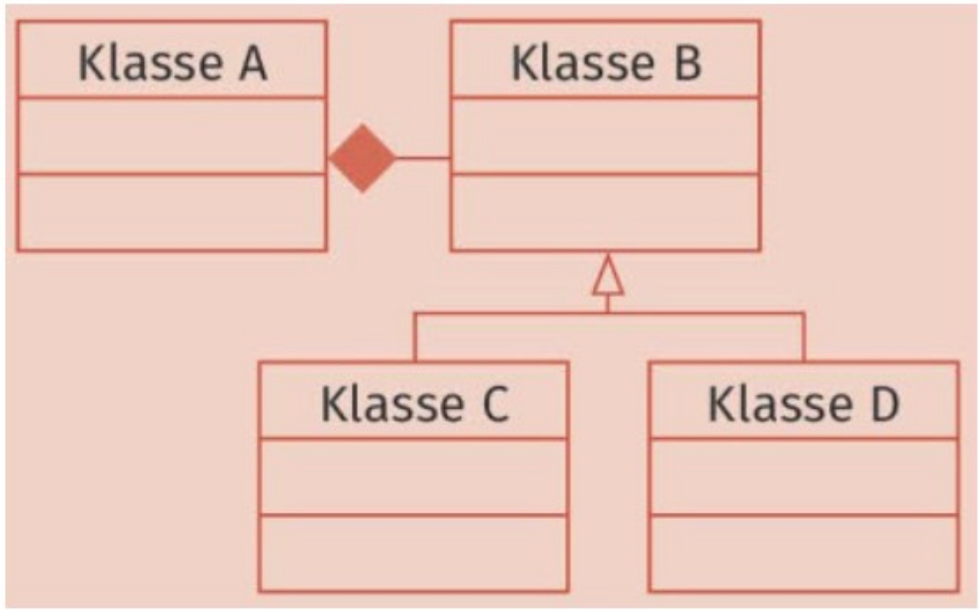
Klassendiagramme

Das Klassendiagramm ist der wichtigste Diagrammtyp der UML. Mit diesem Diagramm werden Klassen beschrieben und die Beziehungen zwischen diesen dargestellt. Es stellt einen Bauplan für die Objekte des Programms dar.



```python
class Charakter:
    def __init__(self, name, level):
        self.name = name
        self.level = level

    def angreifen(self):
        pass

class Krieger(Charakter):
    def __init__(self, name, level, waffe):
        super().__init__(name, level)
        self.waffe = waffe

    def angreifen(self):
        return f"{self.name} greift mit {self.waffe} an!"

class Magier(Charakter):
    def __init__(self, name, level, zauber):
        super().__init__(name, level)
        self.zauber = zauber

    def angreifen(self):
        return f"{self.name} wirkt {self.zauber}!"

class Barde(Charakter):
    def __init__(self,name, level, singen):
        super().__init__(name,level)
        self.singen = singen

    def angreifen(self):
        return f"{self.name} fängt an zu {self.singen}!"

class Dieb(Charakter):
    def __init__(self,name,level,sneak_attack):
        super().__init__(name,level)
        self._sneak = sneak_attack

    def angreifen(self):
        return f"{self.name}greift aus dem Hinterhalt an, mit einem {self._sneak}!"
```

UML Klassendiagramme

Dieser Code definiert eine Reihe von Klassen, die verschiedene Arten von Charakteren in einem Spiel repräsentieren könnten. Jede Klasse hat spezifische **Attribute** und **Methoden**, die ihre einzigartigen Eigenschaften und Fähigkeiten darstellen.

Die Charakter-Klasse ist die Basisklasse. Sie hat zwei Attribute: **name** und **level**. Sie hat auch eine Methode namens **angreifen**, die in den Unterklassen überschrieben wird.
Die **Krieger, Magier, Barde** und **Dieb** Klassen erben von der Charakter-Klasse und fügen jeweils ein zusätzliches Attribut hinzu (**waffe**, **zauber**, **singen** und **_sneak**), das ihre spezifische Art zu angreifen repräsentiert. Sie überschreiben auch die angreifen Methode, um ihre einzigartige Art zu angreifen zu definieren.

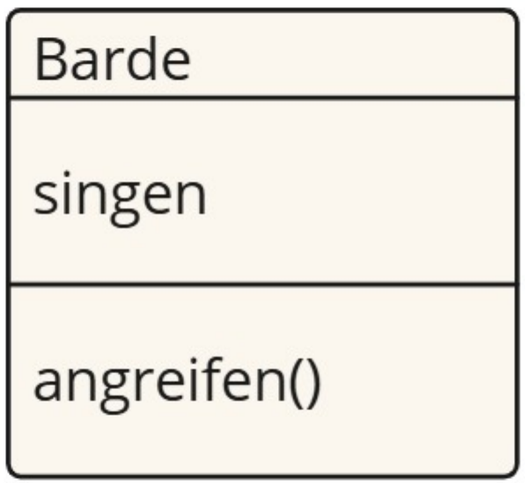
Übertragen der Klasse in die UML Schreibweise

```python
class Charakter:
    def __init__(self, name, level):
        self.name = name
        self.level = level

    def angreifen(self):
        pass
```

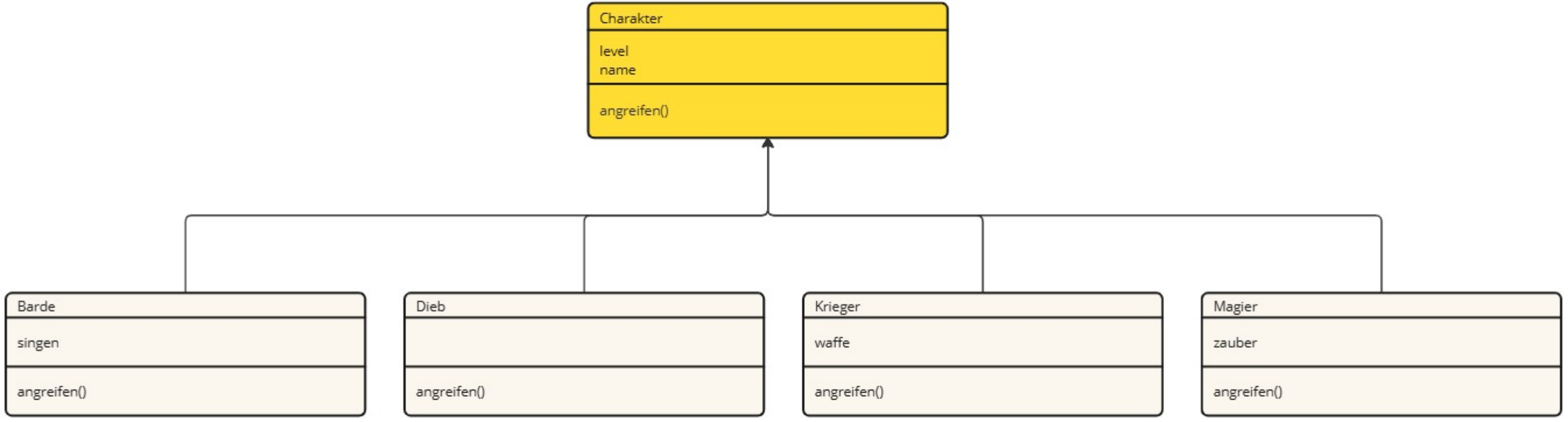
Klassen werden in der UML durch Rechtecke dargestellt, die den Namen der Klasse und/oder die Eigenschaften und Methoden der Klasse enthalten.
Klassename Klassenname, Eigenschaften und Methoden werden durch eine horizontale Linie getrennt. Der KlassenMethoden (Funktionen) steht im Singular und beginnt mit einem Großbuchstaben. Eigenschaften können näher beschrieben werden, z.B. durch ihren Typ. Methoden können ebenfalls durch Parameter usw. ergänzt werden.

Übertragen der Klasse in die UML Schreibweise

```python
class Barde(Charakter):
    def __init__(self,name, level, singen):
        super().__init__(name,level)
        self.singen = singen

    def angreifen(self):
        return f"{self.name} fängt an zu {self.singen}!"
```

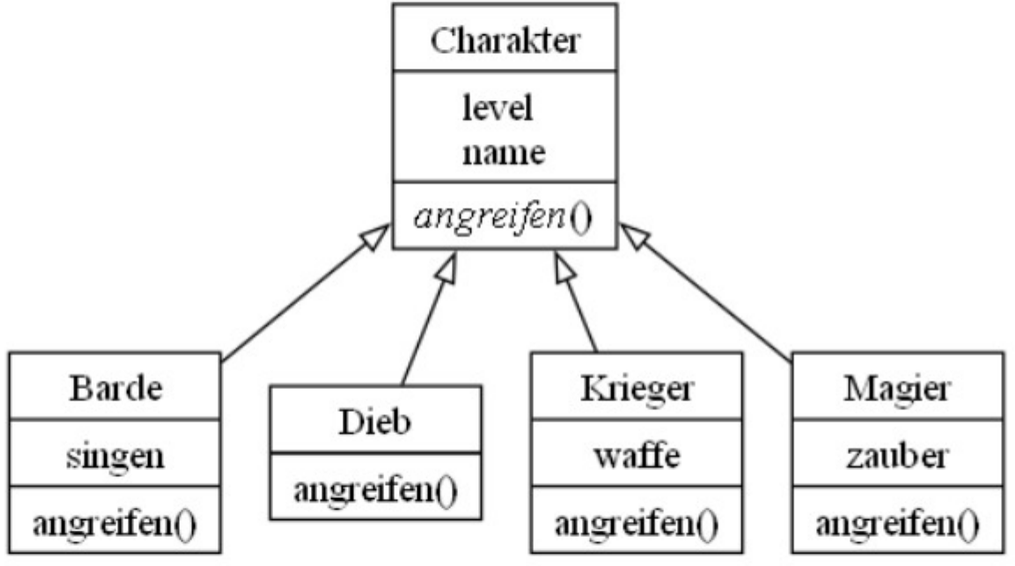


Komplettes Klassendiagramm



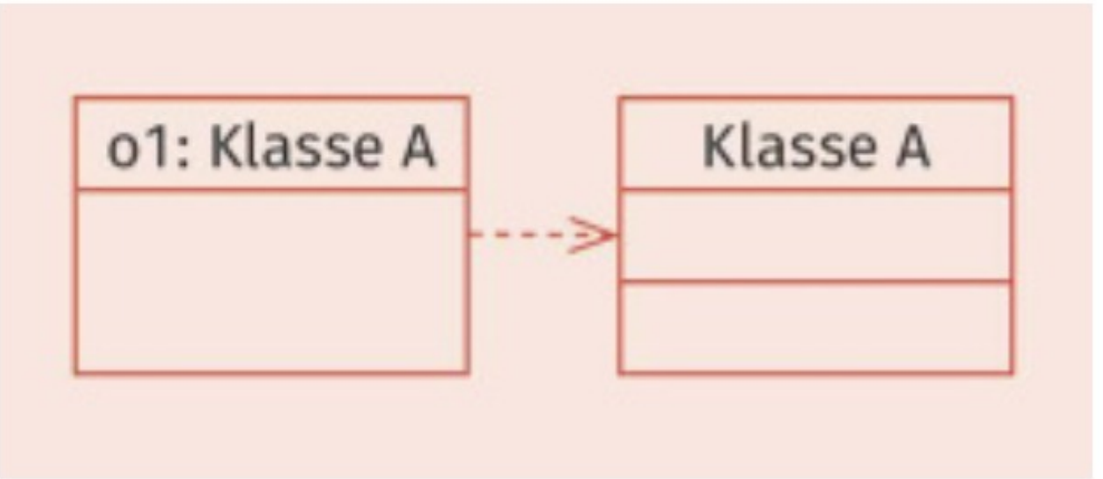
...Tools

Generiert mit Pylint



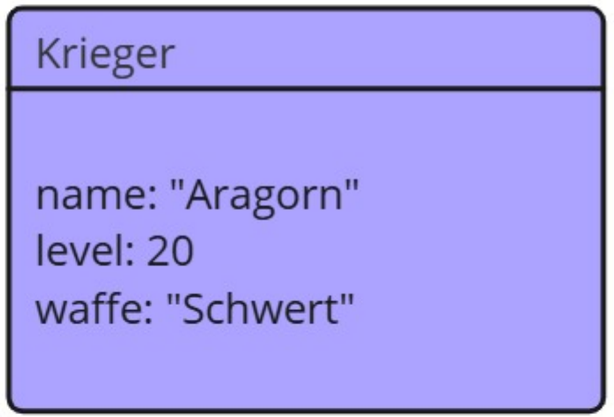
Objektdiagramm

Das Objektdiagramm ist eine Spezifizierung des Klassendiagramms. Es stellt die Beziehungen der tatsächlich erzeugten Objekte zu einem bestimmten Zeitpunkt zur Laufzeit dar. Objektdiagramme können als Ergänzung zu den Klassendiagrammen aufgefasst werden.

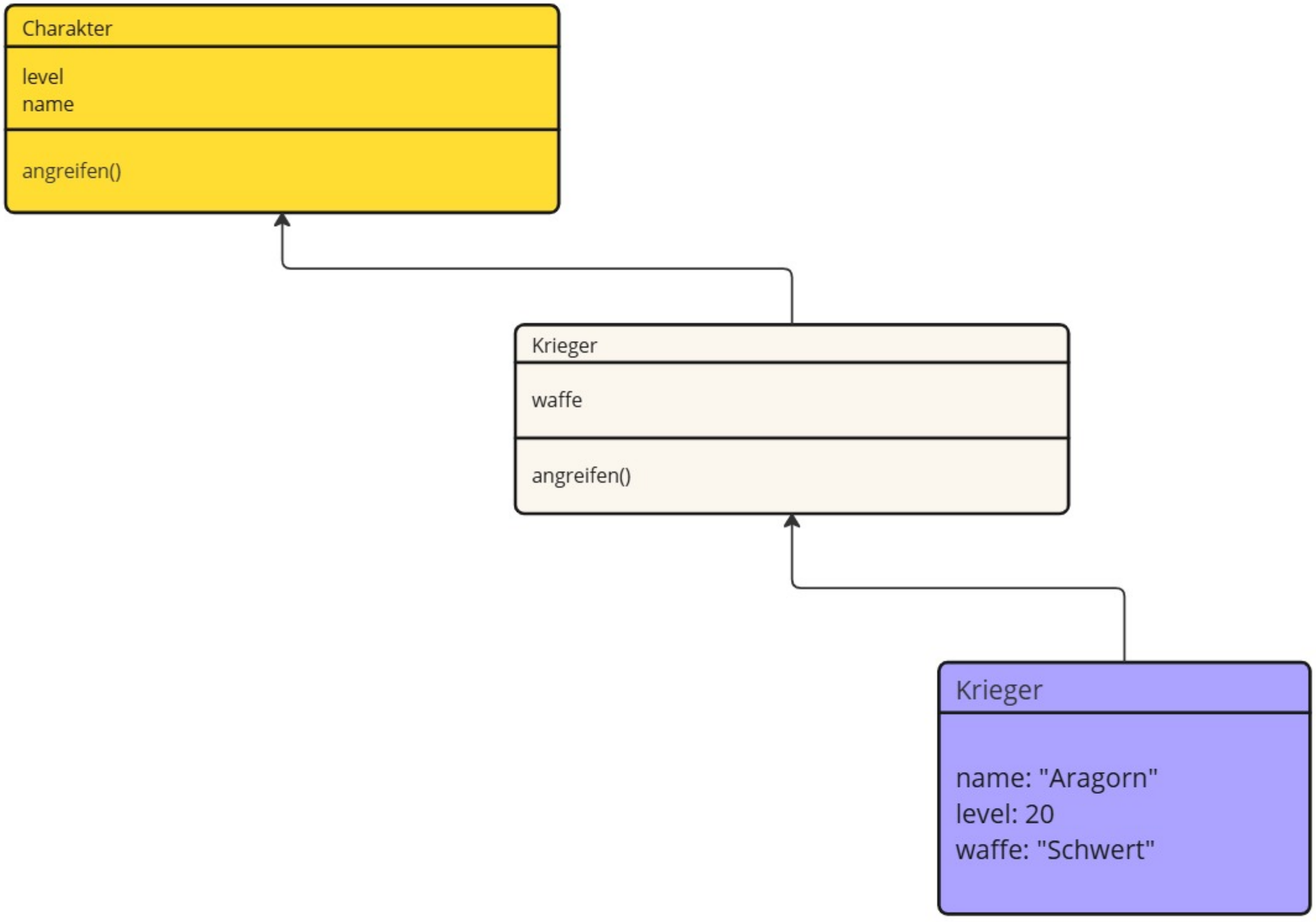


Objekte erstellen

```python
krieger = Krieger("Aragorn", 20, "Schwert")
print(krieger.angreifen())  # Aragorn greift mit Schwert an!
```

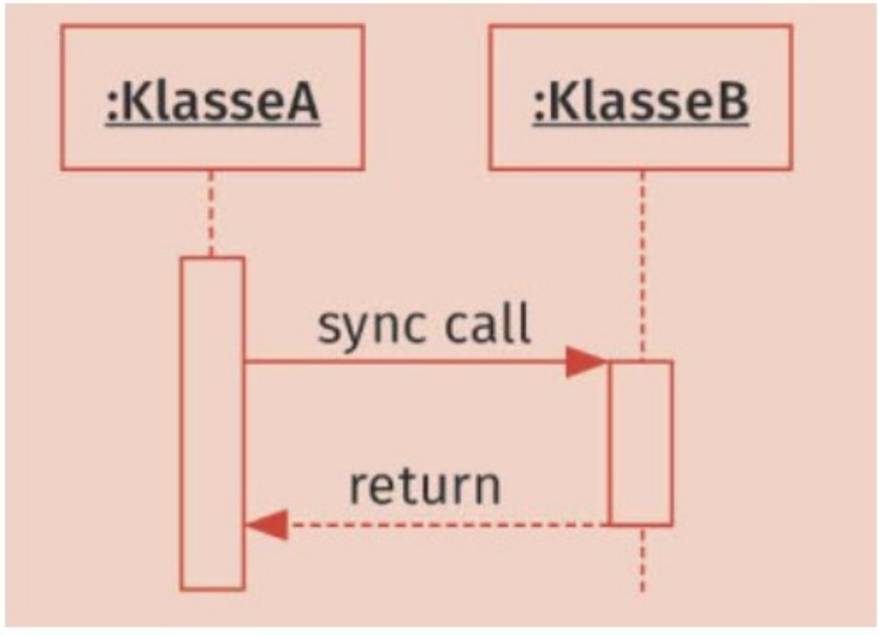


Objektdiagramm



Sequenzdiagramm

Im Sequenzdiagramm liegt der Fokus auf den Interaktionen zwischen Objekten. Hier wird vor allem der Nachrichtenfluss und Nachrichtenaustausch zwischen den Objekten betrachtet. Dabei spielt vor allem der zeitliche
Ablauf der Nachrichten eine entscheidende Rolle. Informationen über die Beziehungen der Objekte untereinander sind im Sequenzdiagramm nicht vorhanden.



```python
class Charakter:
    def __init__(self, name):
        self.name = name

    def sprechen(self, nachricht):
        print(f"{self.name} sagt: {nachricht}")

    def frage_stellen(self, frage, optionen):
        print(f"{self.name} fragt: {frage}")
        for i, option in enumerate(optionen, 1):
            print(f"Option {i}: {option}")

    def antworten(self, option):
        print(f"{self.name} antwortet: {option}")
        return option

def konsequenzen(charakter, antwort):
    if antwort == "Durch die Minen von Moria":
        print(f"{charakter.name}, das ist ein gefährlicher Weg. Es gibt dort viele Gefahren.")
    elif antwort == "Über den Pass von Caradhras":
        print(f"{charakter.name}, das könnte eine gute Option sein, aber es ist ein sehr steiler Aufstieg.")
    else:
        print(f"{charakter.name}, das ist der längste Weg, aber vielleicht auch der sicherste.")

# Charaktere erstellen
gandalf = Charakter("Gandalf")
frodo = Charakter("Frodo")

# Dialog starten
gandalf.sprechen("Frodo!")
frodo.sprechen("Ja?")
gandalf.frage_stellen("Welchen Weg sollten wir nehmen?",
                      ["Durch die Minen von Moria",
                       "Über den Pass von Caradhras",
                       "Um den Berg herum"])
antwort = frodo.antworten("Durch die Minen von Moria")
konsequenzen(frodo, antwort)
```

1. Es wird eine Klasse `Charakter` definiert, die drei Methoden hat: `**sprechen**`, `**frage_stellen**` und `**antworten**`. Jeder Charakter hat einen Namen, der beim Erstellen des Charakter-Objekts festgelegt wird.
Die Methode `**sprechen**` ermöglicht es dem Charakter, eine Nachricht auszugeben.

Die Methode `**frage_stellen**` ermöglicht es dem Charakter, eine Frage zu stellen und eine Liste von Optionen zur Auswahl zu präsentieren.

Die Methode `**antworten**` ermöglicht es dem Charakter, auf eine Frage zu antworten.

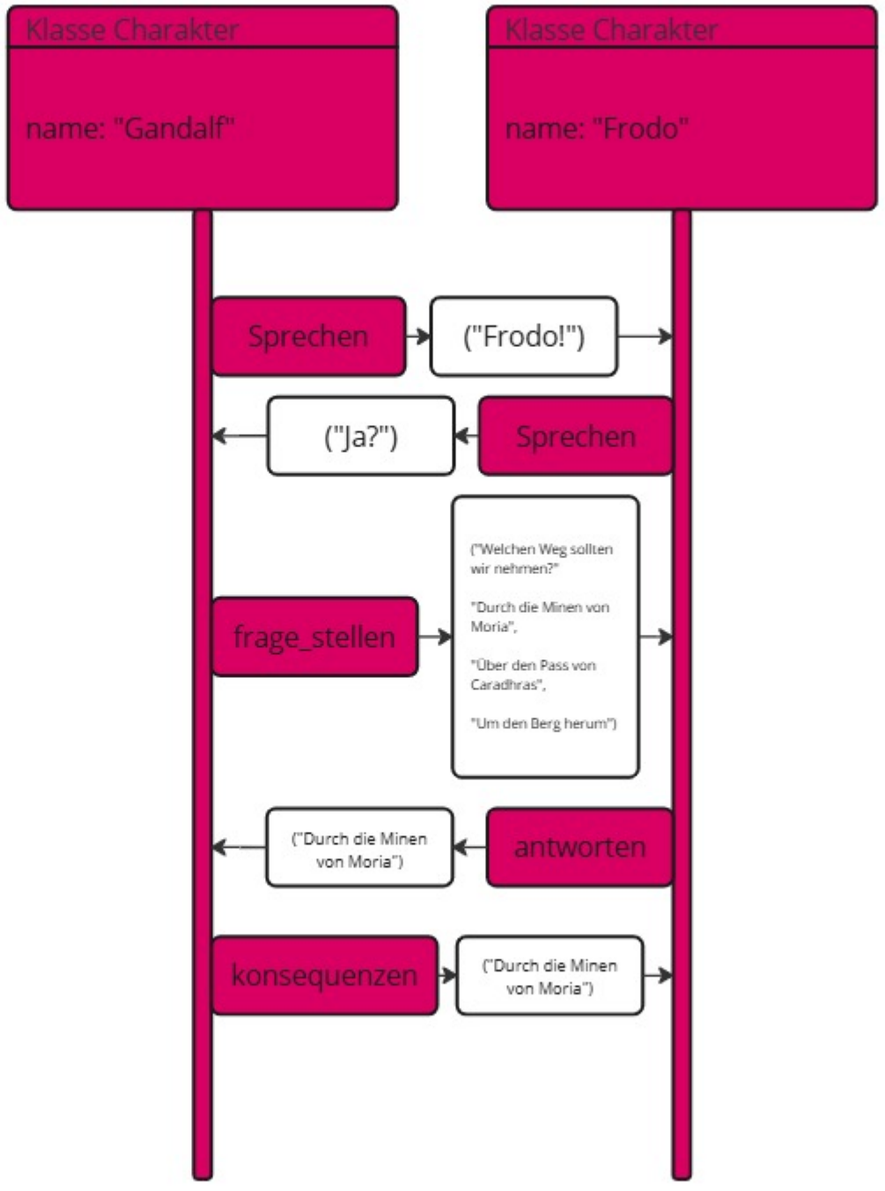
2. Es wird eine Funktion `**konsequenzen**` definiert, die die Konsequenzen der Antwort eines Charakters auf eine Frage ausgibt. Die Konsequenzen hängen von der gegebenen Antwort ab.

3. Zwei Charakter-Objekte, **Gandalf** und **Frodo**, werden erstellt.

4. Ein Dialog zwischen Gandalf und Frodo wird gestartet. Gandalf spricht zuerst Frodo an, Frodo antwortet, dann stellt Gandalf eine Frage mit mehreren Auswahlmöglichkeiten. Frodo antwortet auf die Frage und die Konsequenzen seiner Antwort werden ausgegeben.

Erstellen eine Sequenzdiagramms

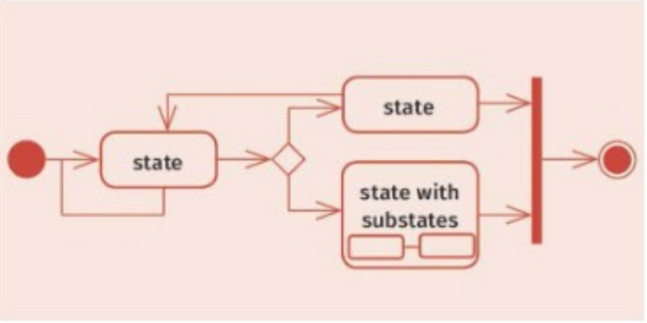
```python
# Charaktere erstellen
gandalf = Charakter("Gandalf")
frodo = Charakter("Frodo")

# Dialog starten
gandalf.sprechen("Frodo!")
frodo.sprechen("Ja?")
gandalf.frage_stellen("Welchen Weg sollten wir nehmen?",
                      ["Durch die Minen von Moria",
                       "Über den Pass von Caradhras",
                       "Um den Berg herum"])
antwort = frodo.antworten("Durch die Minen von Moria")
konsequenzen(frodo, antwort)
```



Zustandsdiagramme

Ein **Zustandsdiagramm**, auch bekannt als Zustandsautomat, bietet eine visuelle Darstellung der verschiedenen Zustände, in denen sich ein System befinden kann, und der Übergänge zwischen diesen Zuständen.

Im Gegensatz zu anderen Diagrammtypen, die eine Aneinanderreihung von Aktionen darstellen, konzentriert sich das Zustandsdiagramm auf die Zustandsänderungen. Es zeigt, wie das System auf bestimmte Ereignisse oder Bedingungen reagiert, indem es von einem Zustand in einen anderen wechselt.



Jeder **Zustand** in einem Zustandsdiagramm repräsentiert eine bestimmte Konfiguration oder einen bestimmten Zustand des Systems. Ein Zustand kann durch Eigenschaften wie Werte von Variablen, aktive Prozesse oder interne Zustände charakterisiert werden.

Ein **Zustandsübergang** ist der Prozess, bei dem das System von einem Zustand in einen anderen wechselt. Zustandsübergänge werden normalerweise durch Ereignisse oder Bedingungen ausgelöst, die im System auftreten.

Ereignisse sind Vorfälle oder Aktionen, die einen Zustandsübergang auslösen können. Sie können intern (innerhalb des Systems) oder extern (von außerhalb des Systems) sein.

Das Zustandsdiagramm ist besonders nützlich für die Modellierung von Systemen, die eine komplexe Logik oder Verhalten haben, das stark von ihrem aktuellen Zustand abhängt.

Ampel.py > ...

```python
class Ampel:
    def __init__(self):
        self.state = "Rot"

    def zustandswechsel(self):
        if self.state == "Rot":
            self.state = "Gelb"
        elif self.state == "Gelb":
            self.state = "Grün"
        else:
            self.state = "Rot"

    def zeige_zustand(self):
        print("Die Ampel ist jetzt", self.state)

# Erstellen Sie ein Ampelobjekt und führen Sie einige Zustandswechsel durch
ampel = Ampel()
ampel.zeige_zustand()
ampel.zustandswechsel()
ampel.zeige_zustand()
ampel.zustandswechsel()
ampel.zeige_zustand()
ampel.zustandswechsel()
ampel.zeige_zustand()
```

Dieser Code definiert eine Klasse namens `Ampel`, die ein Modell einer Verkehrsampel darstellt. Die Ampel hat drei Zustände: "Rot", "Gelb" und "Grün".

Die Klasse `Ampel` hat drei Methoden:
- `__init__`: Dies ist der Konstruktor der Klasse, der aufgerufen wird, wenn ein neues Objekt der Klasse erstellt wird. Er initialisiert den Zustand der Ampel auf "Rot".

- `zustandswechsel`: Diese Methode ändert den Zustand der Ampel. Wenn der aktuelle Zustand "Rot" ist, wechselt er zu "Gelb". Wenn der Zustand "Gelb" ist, wechselt er zu "Grün". Und wenn der Zustand "Grün" ist, wechselt er zurück zu "Rot".

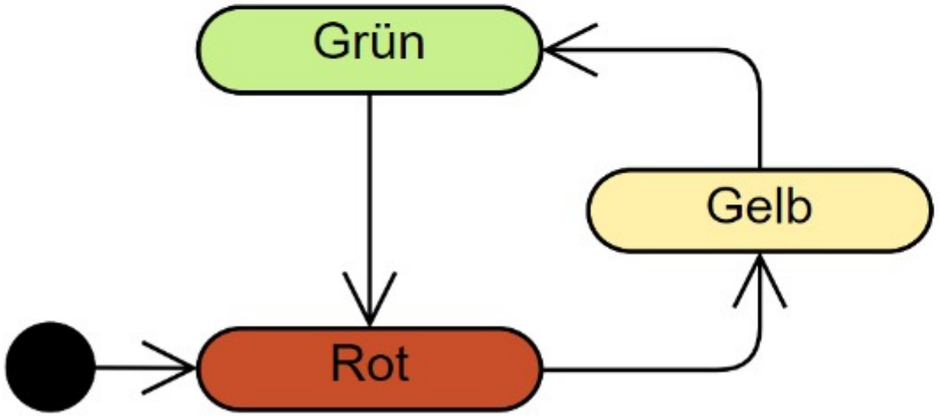
- `zeige_zustand`: Diese Methode gibt den aktuellen Zustand der Ampel aus.

Nach der Definition der Klasse wird ein Ampelobjekt erstellt und seine Methoden werden aufgerufen, um den Zustand der Ampel zu ändern und anzuzeigen.

Initial Pseudo State (schwarzer Kreis): Beginnen Sie mit einem schwarzen Kreis, der den Anfangszustand darstellt.

State (abgerundetes Rechteck): Zeichnen Sie ein abgerundetes Rechteck für jeden Zustand, den Ihre Ampel haben kann: “Rot”, “Gelb” und “Grün”.

Transition (schwarzer Pfeil): Zeichnen einen schwarzen Pfeil von einem Zustand zum nächsten, entsprechend der Logik in der zustandswechsel Methode.
Also von “Rot” zu “Gelb”,
von “Gelb” zu “Grün”
und von “Grün” zurück zu “Rot”.



Final State
(weiß eingerahmter schwarzer Kreis)

Da die Ampel ständig zwischen den Zuständen wechselt und keinen Endzustand hat, benötigen Sie keinen Final State.

Tool: https://online.visual-paradigm.com

Weitere Beispiele