CAMPUS 02 GRAZ
FACHHOCHSCHULE DER WIRTSCHAFT

SOFTWARE DESIGN FOLIENSATZ 1

Organisatorisches + Strategy Pattern

Bernhard Fuchs
ZAM - WS 2025/26

basierend auf Unterlagen von Christian Hofer, Christoph Pangerl, Michael Fladischer, angelehnt an das Buch„Head First Design Patterns“

TERMINE

0004VD2005 SOFTWARE DESIGN (29,75UE IL, WS 2025/26)

Gruppe

	Tag	Datum	von	bis	Ort	Ereignis	Termintyp	Lerneinheit	Vortragende*r	Anmerkung
	Standardgruppe									
Heute	Do	22.01.2026	08:15	11:45	CZ106	Abhaltung	fix	**1 Organisation + Strategy Pattern**		
	Do	29.01.2026	08:15	11:45	CZ106	Abhaltung	fix	**2 Decorator Pattern**		
	Do	05.02.2026	12:30	16:00	CZ106	Abhaltung	fix	**3 Observer + Singleton**		
	Fr	06.02.2026	08:15	12:15	CZ106	Abhaltung	fix	**4 Factory**		
	Do	12.02.2026	08:15	13:00	CZ106	Abhaltung	fix	**5 Command + Adapter**		
	Fr	13.02.2026	08:15	12:15	CZ106	Abhaltung	fix	**6 Facade, Template, Iterator**		
	Do	19.02.2026	08:15	11:45	CZ106	Abhaltung	fix	**Wiederholung**		
	Fr	20.02.2026	08:15	12:15	CZ106	Prüfungstermin	fix	**Prüfung**		

(voraussichtlicher Plan)

GROBE LERNZIELE

- Designanforderungen
- SOLID Prinzipien
- Ausgewählte Design Patterns

LEISTUNGSBEURTEILUNG

- 1. und 2. Antritt
  - Schriftliche Klausur
  - > 50% um positive Note zu erhalten
- 3. Antritt
  - Kommissionelle Prüfung
  - Schriftlich und mündlich

LITERATUR

- Head First Design Patterns
  - Mittlerweile zweite Auflage
- Deutsch: Entwurfsmuster von Kopf bis Fuß
- https://refactoring.guru/design-patterns
- https://www.philipphauer.de/study/se/design-pattern.php

SOFTWARE DESIGN

- Prozess der **Planung** einer Software auf **technischer** Ebene
  - Inkludiert das Lösen von Problemen

SOFTWARE DESIGN

- Welche Anforderungen stellen wir an Software Design?

ANFORDERUNGEN SOFTWARE DESIGN

- Erweiterbarkeit
- Wartbarkeit
- Wiederverwendbarkeit
- Testbarkeit

ENTSTEHUNG

- Gang of Four – Überlegung 90er
  - Was unterscheidet erfolgreiche und nicht-erfolgreiche OO(**o**bject-**o**riented)-Designs?
- Aus dem Buch: Design Patterns. Elements of Reusable Object-Oriented Software
  - Design Patterns

DESIGN PATTERN

- Ein Pattern ist eine **Lösung** zu einem **immer wiederkehrenden** Problem.
  - Ein Konzept um ein Problem zu lösen
- Unterteilt in:
  - Verhaltensmuster (Behavioral Pattern)
  - Erzeugungsmuster (Creational Pattern)
  - Strukturmuster (Structural Pattern)

VERHALTENSMUSTER (BEHAVIORAL PATTERN)

- Verhalten von Klassen
  - Zusammenarbeit und Nachrichtenaustausch zwischen Objekten

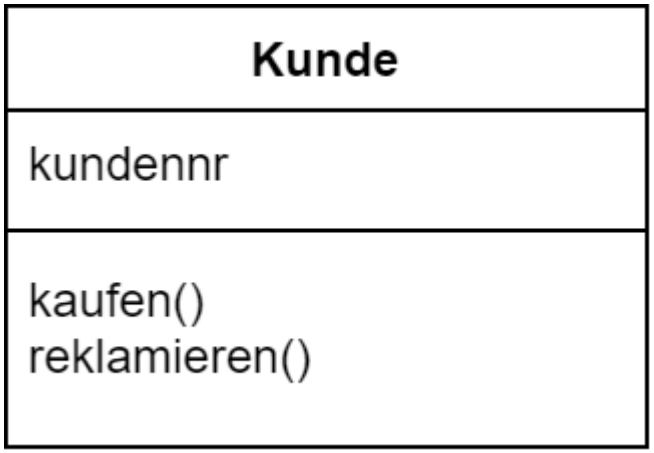
ERZEUNGUNGSMUSTER (CREATIONAL PATTERN)

- Objekterzeugung
  - Variieren welches Objekt erzeugt wird
  - Andere Objekte zur Objekterzeugung verwenden

STRUKTURMUSTER (STRUCTURAL PATTERN)

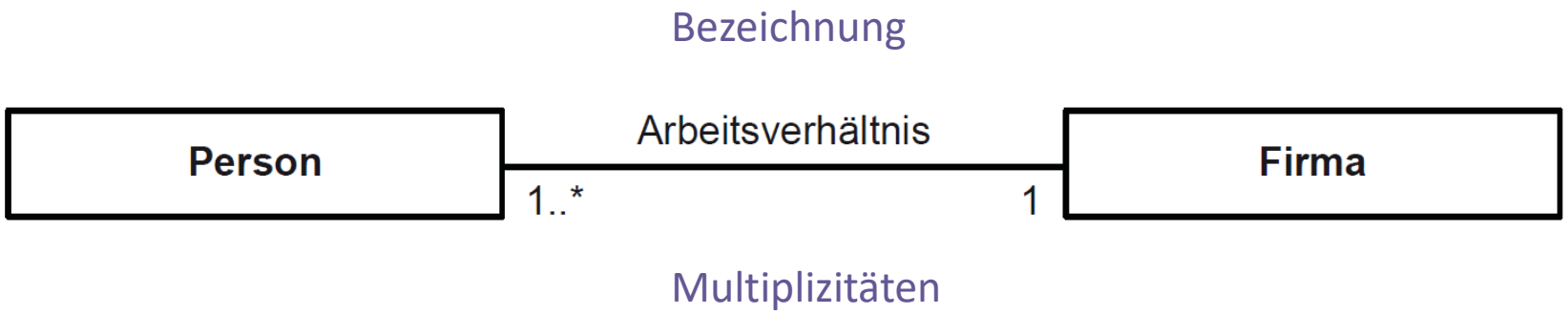
- Vereinfachung Struktur
  - Klassen oder Objekte werden zu einer größeren Struktur kombiniert

WIEDERHOLUNG UML : KLASSE



- Abstrakte Klasse
  - **Kunde {abstract}** oder kursiv
- Interface
  - **<<interface>> Kunde**

ASSOZIATION



- Multiplizitäten:

1: genau 1

3: genau 3

*: beliebig viele

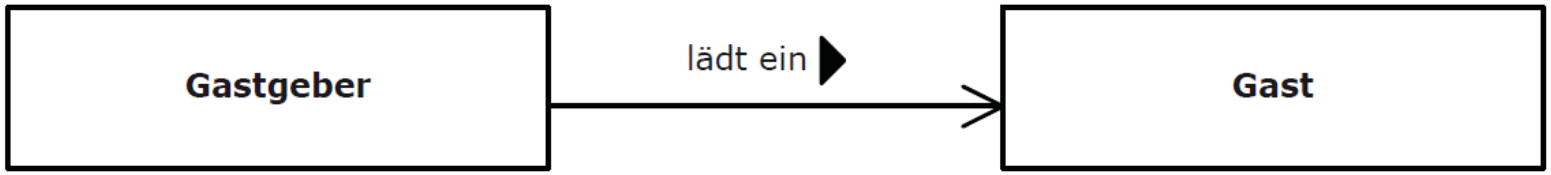
0..1: zwischen 0 und 1

1..4: zwischen 1 und 4

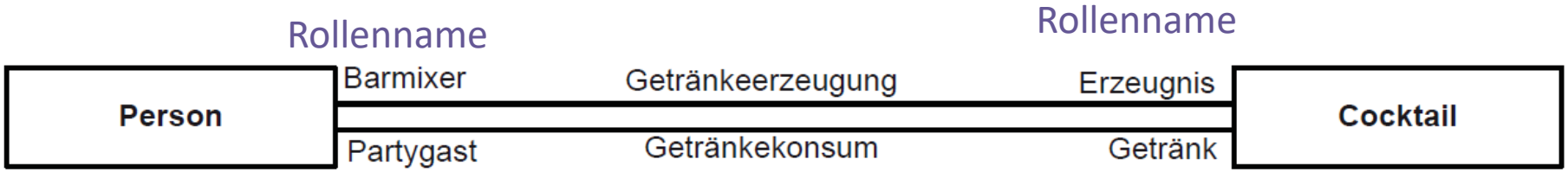
1..*: zwischen 1 und beliebig viele

ASSOZIATION 2

Bezeichnung mit Leserichtung

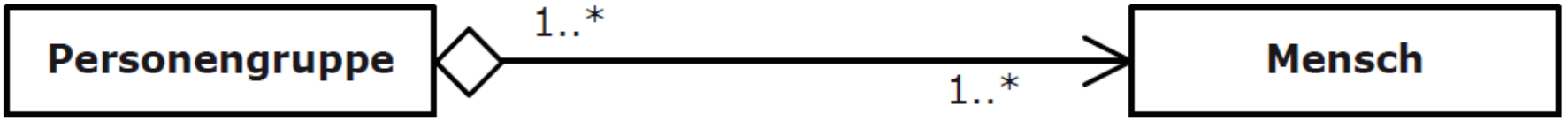


Richtung der Navigierbarkeit



AGGREGATION

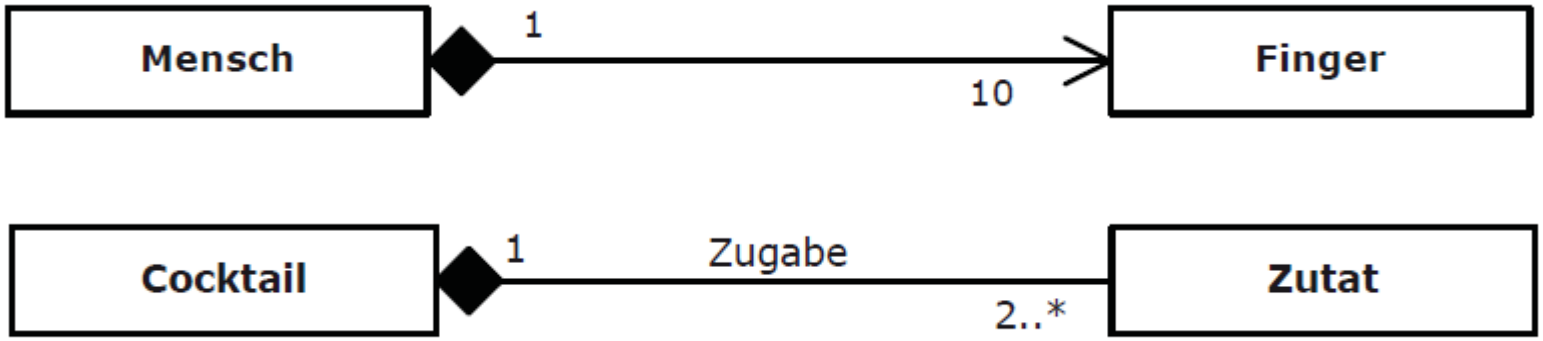
- Teil-Ganzes Beziehung
  - Besteht-aus
  - Ist Teil von



Pfeile eher unüblich

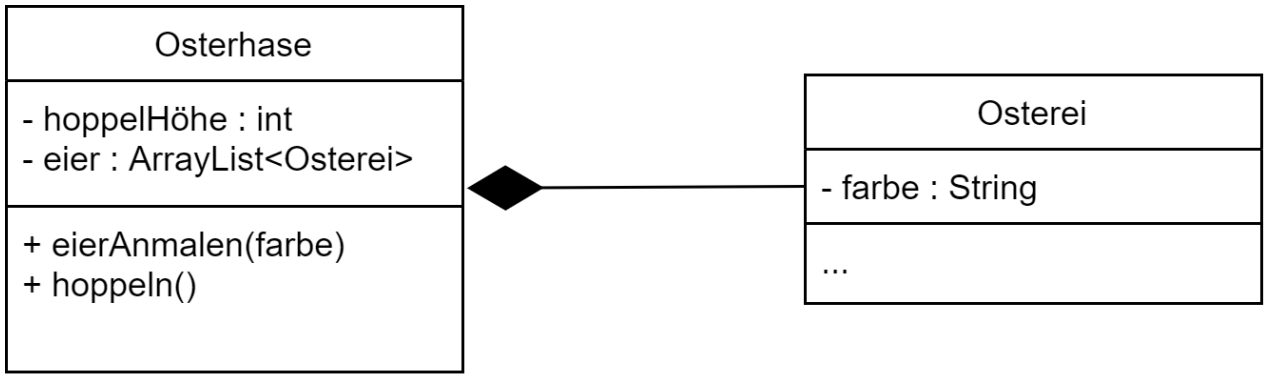
KOMPOSITION (IM UML SINNE)

- Strengerer Zusammenhang
  - Physische Inklusion
    - Ein Teil darf höchstens einem Ganzen zugeordnet sein
    - Gemeinsame Lebensdauer



WIEDERHOLUNG KLASSENDIAGRAMME

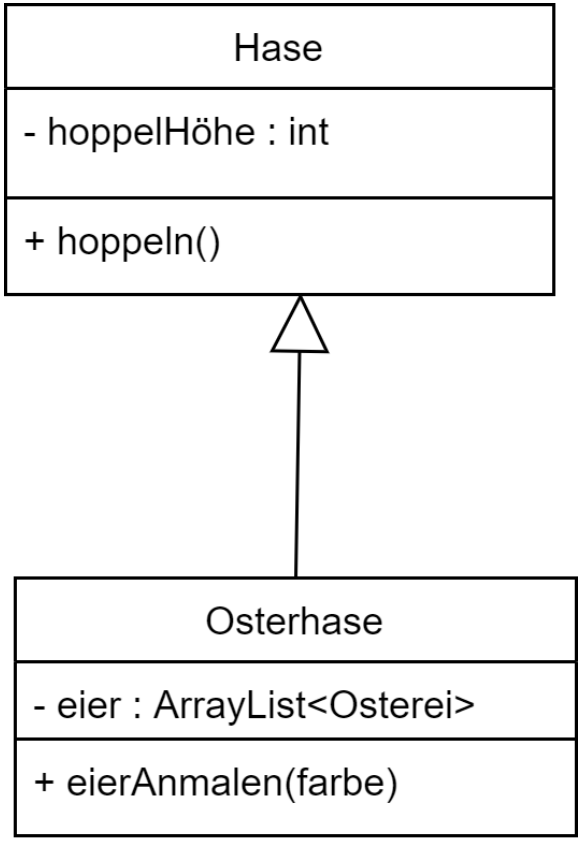
- Komposition (has-a Beziehung)



visual-paradigm.com:
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

WIEDERHOLUNG KLASSENDIAGRAMME

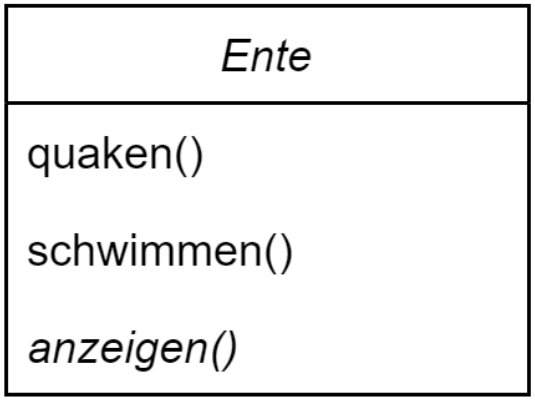
- Vererbung (is-a Beziehung)



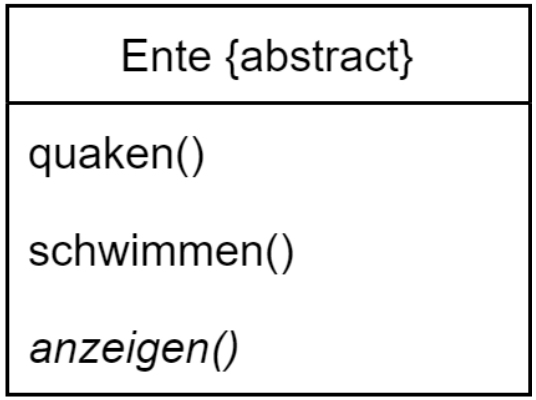
WIEDERHOLUNG KLASSENDIAGRAMME

- Abstrakte Klasse

kursiv geschrieben



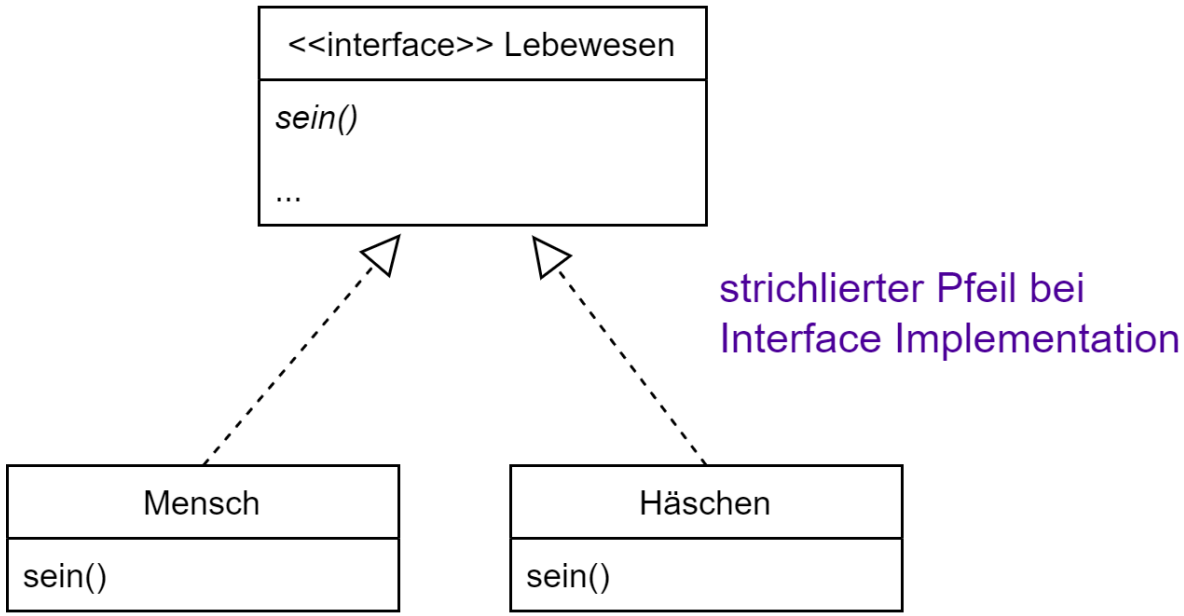
oder



WIEDERHOLUNG KLASSENDIAGRAMME

- Interface und Interface Implementierung

<<interface>> neben dem Klassennamen signalisiert Interface

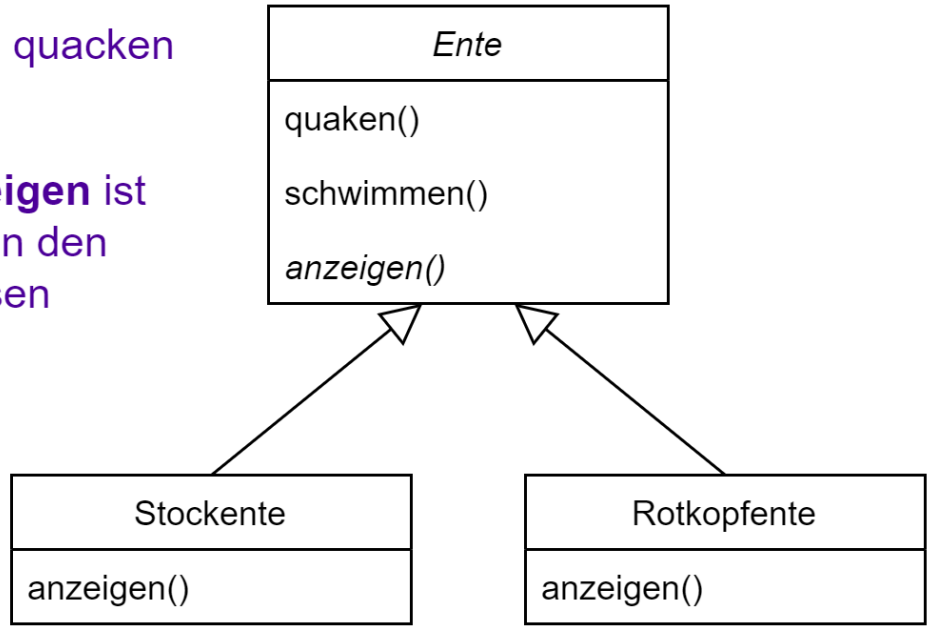


SIMUDUCK -ENTENTEICHSIMULATION

SIMUDUCK KLASSENDIAGRAMM

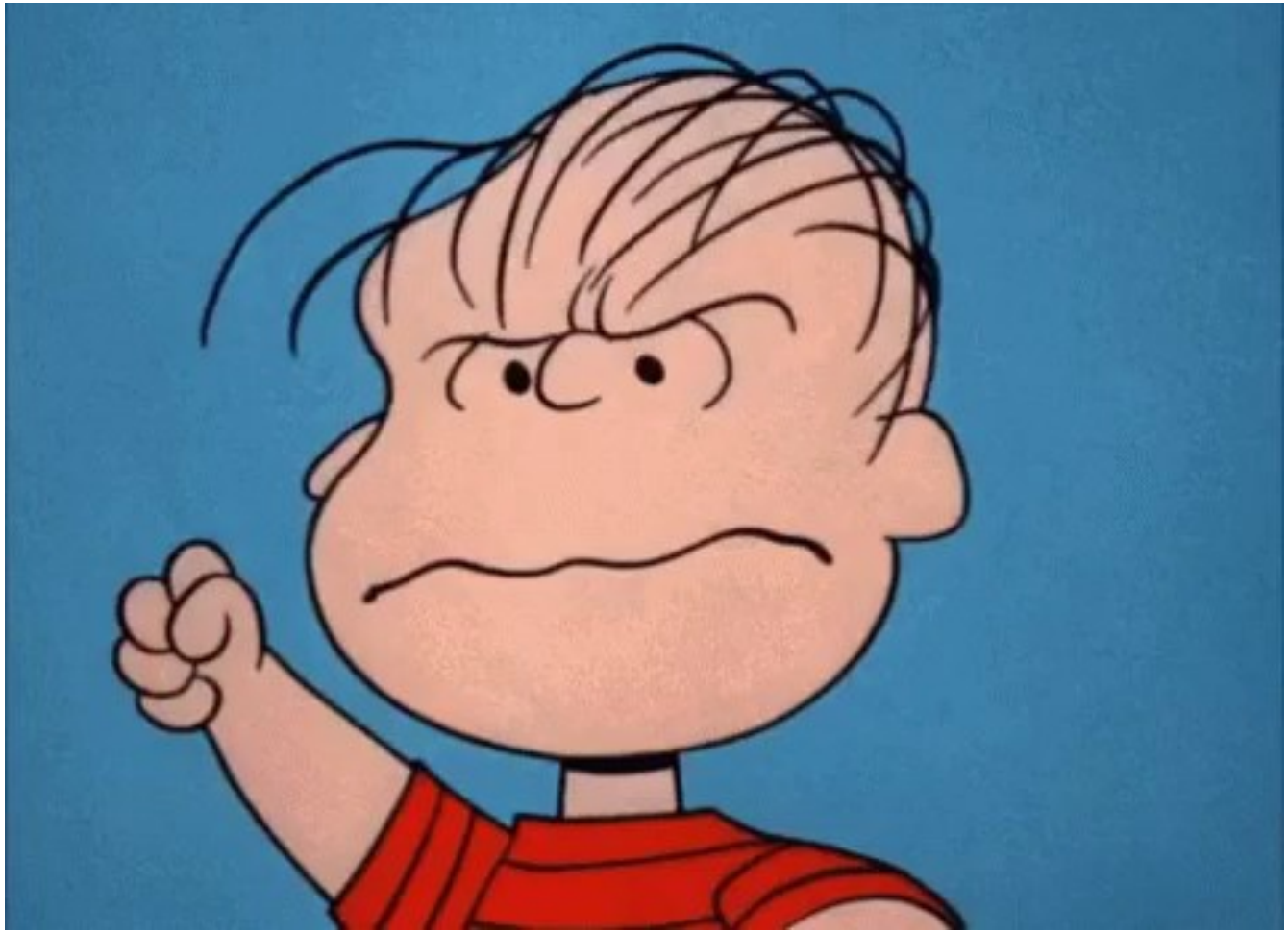
Alle Enten können quacken und schwimmen.

Die Methode **anzeigen** ist abstrakt und wird in den abgeleiteten Klassen implementiert:



... und noch viele weitere Enten

SPIELERINNEN WENDEN SICH AB



NEUES FEATURE

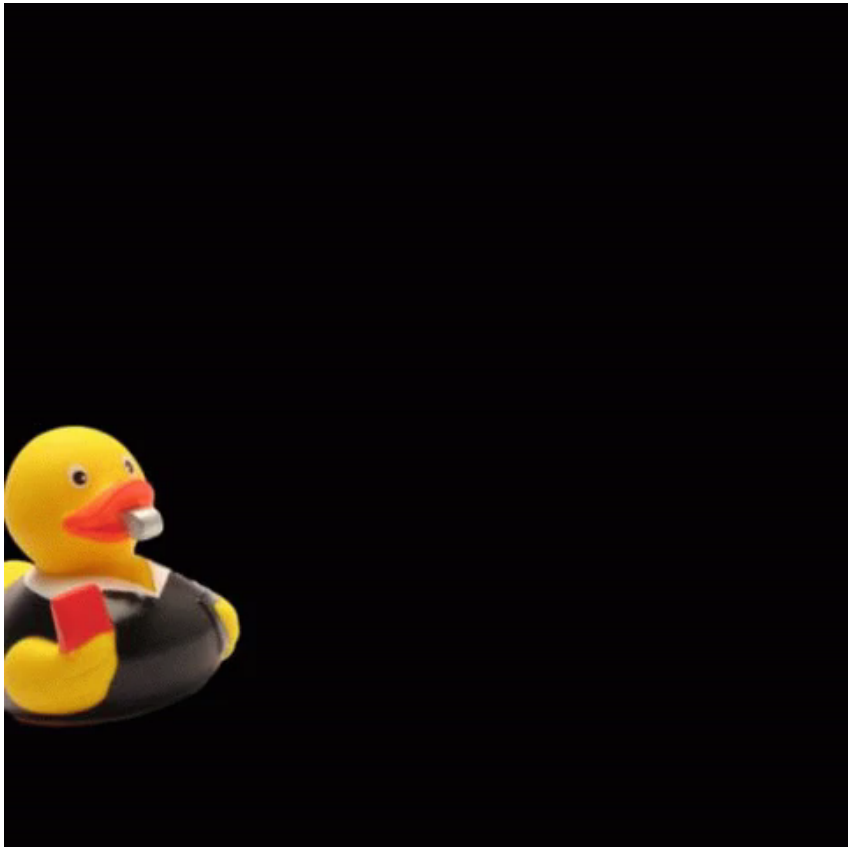
- Fliegende Enten

INNOVATIVE IDEE

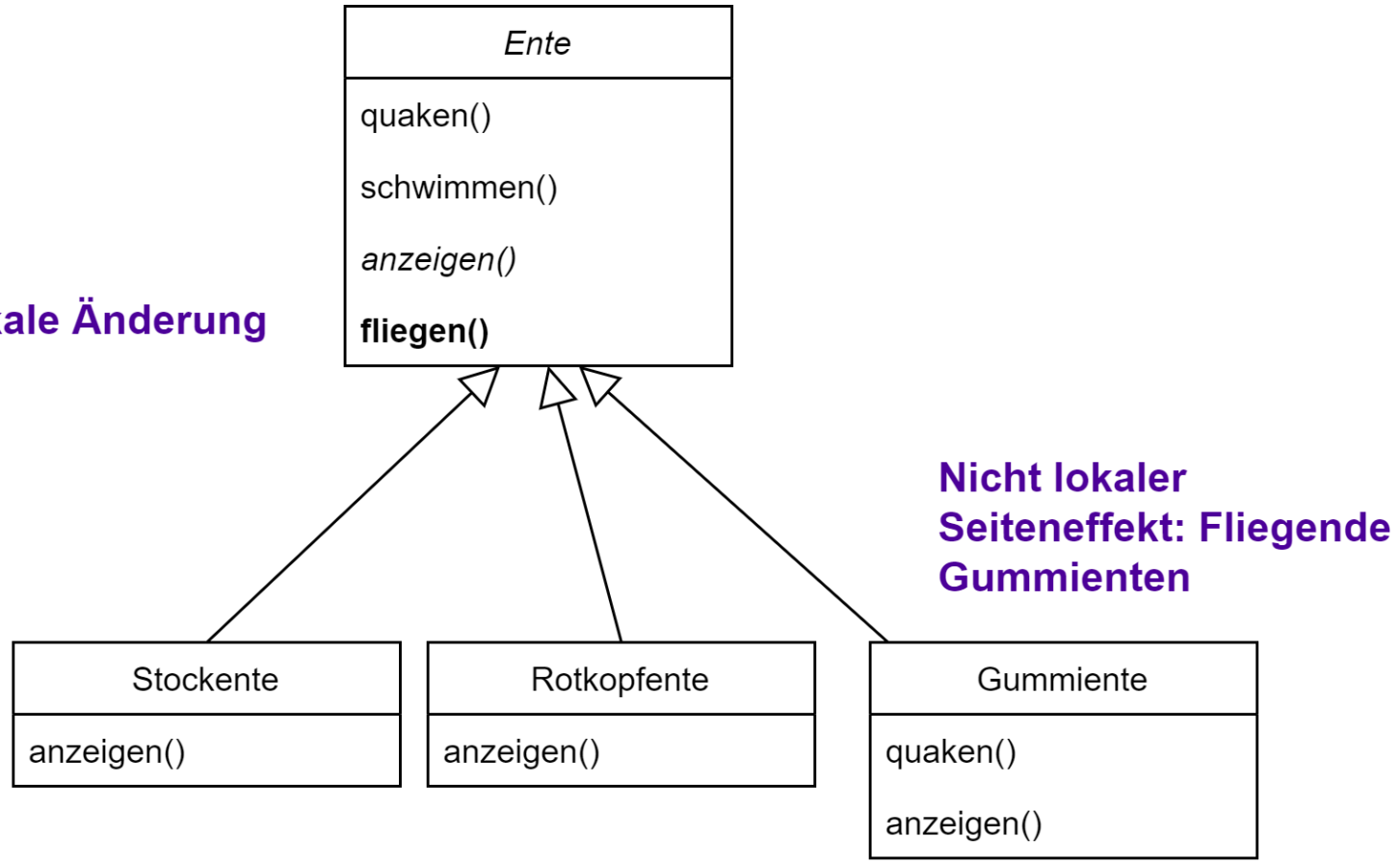
- Neue Methode einführen:
  - fliegen()
- Let's code

PRÄSENTATION VOR CEO

GUMMIENTEN FLIEGEN DURCH DIE SIMULATION



UNSERE ÄNDERUNG



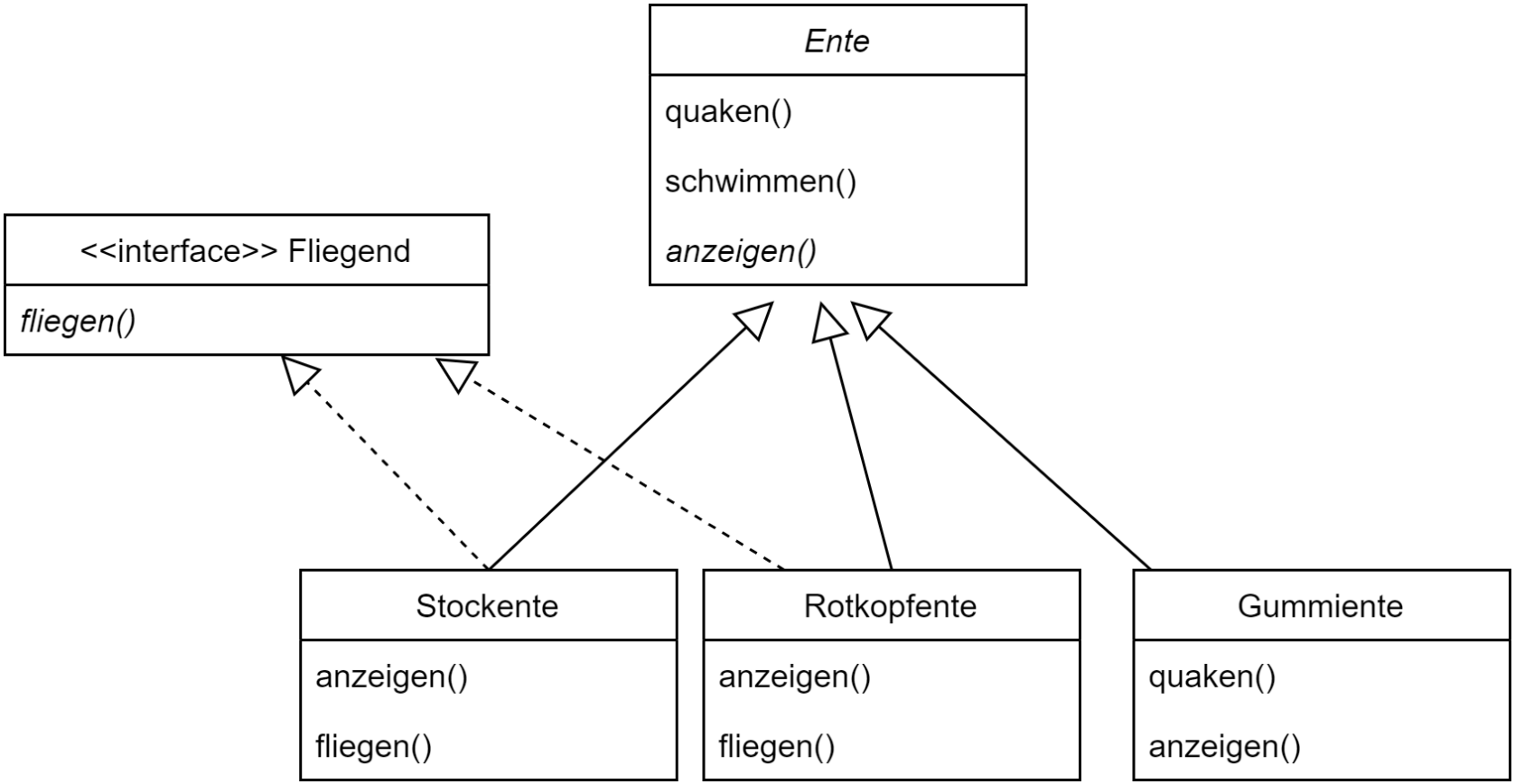
ALTERNATIVE (INNOVATIVE) IDEE

- fliegen() für Gummiente überschreiben
- Nachteile:
  - Duplizierter Code
  - Laufzeitverhalten nicht veränderbar
  - Keine zentrale Übersicht alle Verhalten
  - Wie zuvor nicht lokale Seiteneffekte

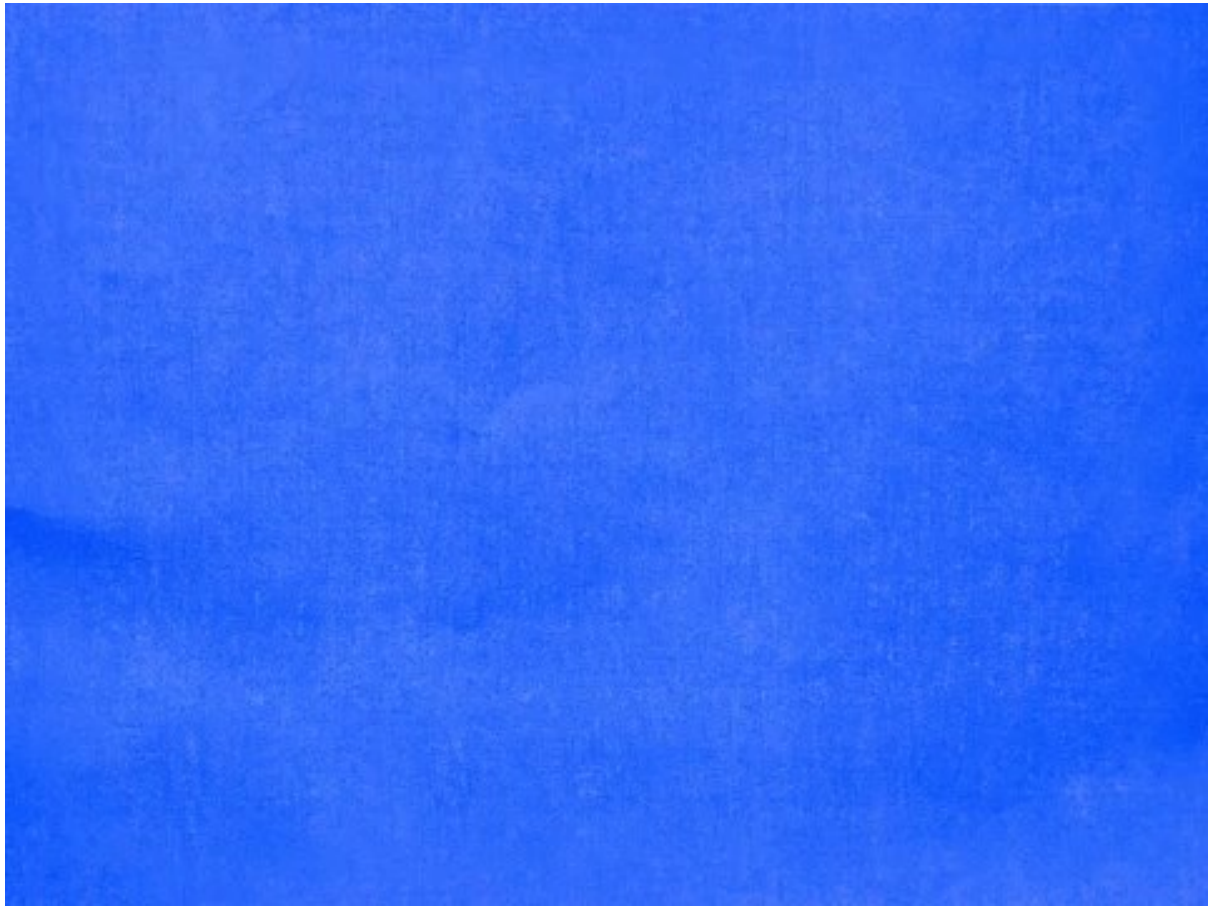
ALTERNATIVE (INNOVATIVE) IDEE

- switch in fliegen()
- Let's code :)

NÄCHSTE IDEE: INTERFACE



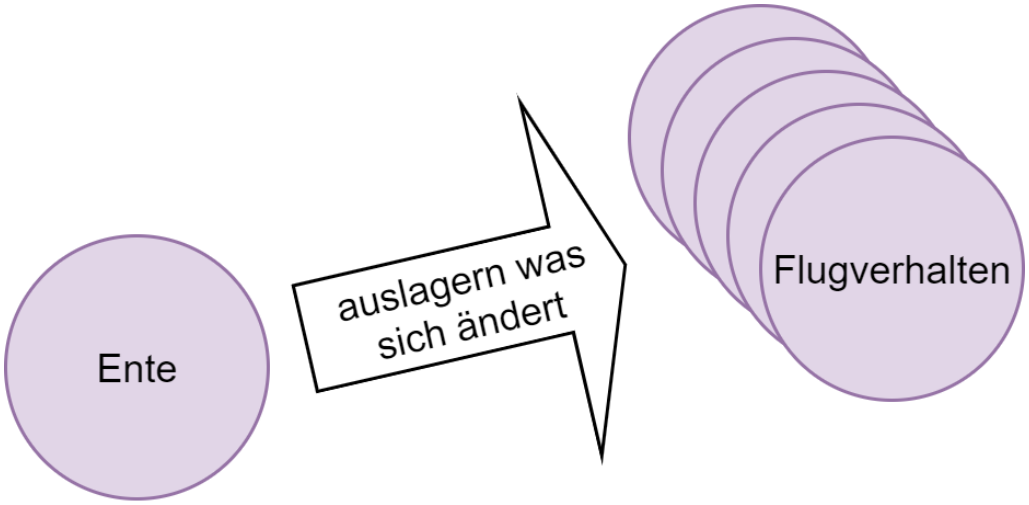
DIE EINZIGE KONSTANTE IN DER SOFTWAREENTWICKLUNG



(DAS) DESIGN PRINZIP

Identifiziere jene Aspekte einer Applikation, welche sich häufig ändern und trenne diese von jenen, welche gleich bleiben.

DESIGN PRINZIP



ZWEITES DESIGN PRINZIP

- Zuvor waren wir abhängig von der Implementation in Ente oder einer abgeleiteten Klasse davon.
  - Das Flugverhalten können wir nur verändern in dem wir Code schreiben.
- Lagern das Flugverhalten in ein eigenes Interface aus, Ente selbst muss nicht von der Implementation seines Verhaltens wissen.
  - Konkrete Klassen für jede Art des Flugverhalten

ZWEITES DESIGN PRINZIP

Programmiere gegen ein Interface (Supertyp) an statt gegen eine konkrete Implementierung.

- Deklarierte Typ ist eine abstrakte Klasse/Interface
- Weisen Implementierung zu
- Klasse muss nichts über das implementierende Objekt wissen

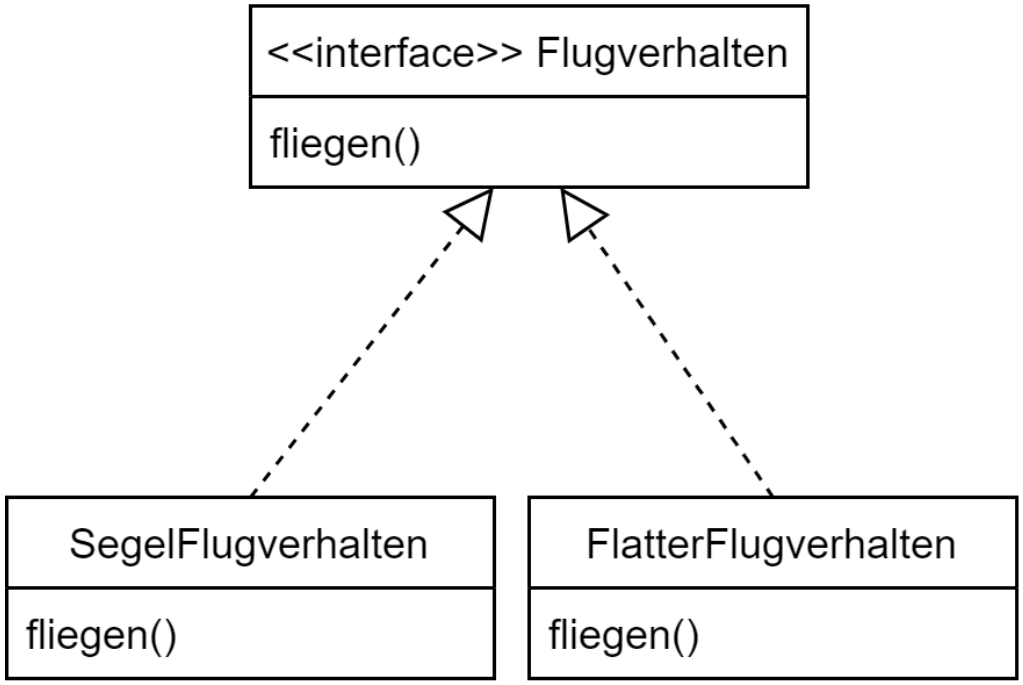
ZWEITES DESIGN PRINZIP

```
// Programmiere gegen Implementierung
Hund h = new Hund();
h.bellt();
// Programmiere gegen Interface
Tier t = new Hund(); //noch besser getTier()
t.machGeraeusch();
```

ZWEITES DESIGN PRINZIP: NOCH BESSER

```
// Programmiere gegen Implementierung
Hund h = new Hund();
h.bellt();
// java Programmiere gegen Interface (noch besser)
Tier t = getTier() // statt new Hund()
t.machGeraeusch();
```

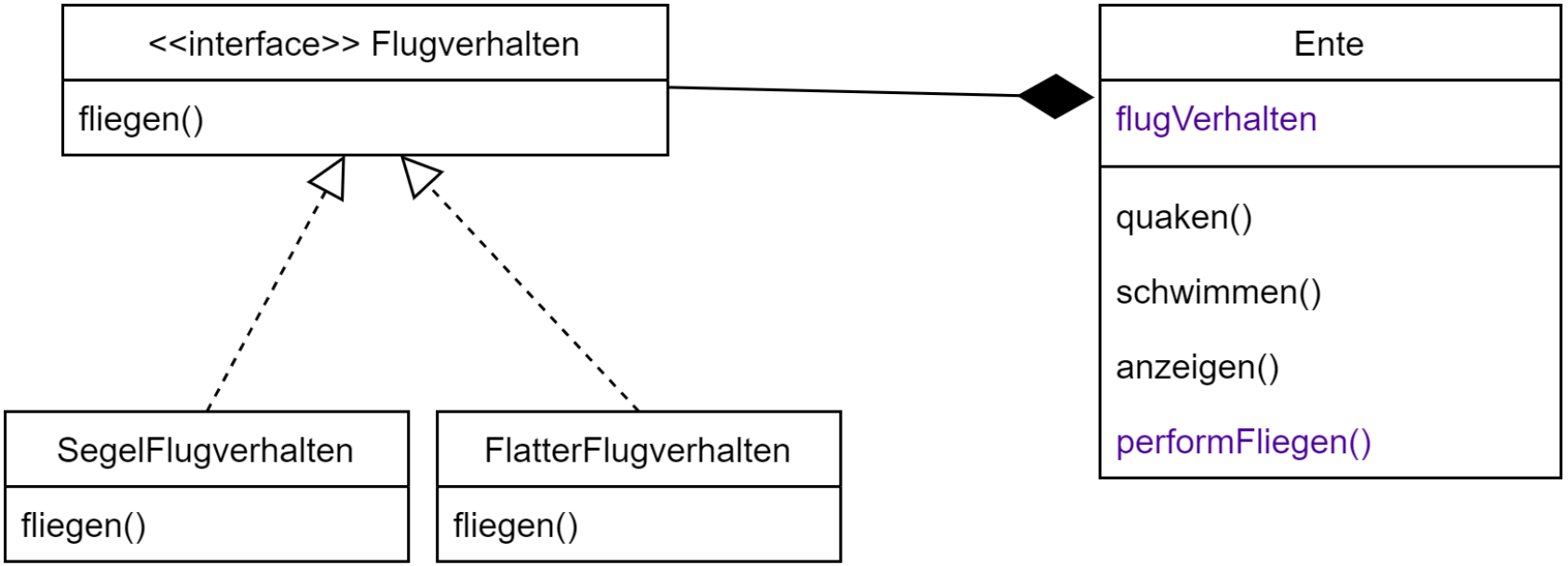
LÖSUNG



Andere Objekte können Flugverhalten verwenden, da nicht mehr in Entenklassen versteckt.

Wir können Neue hinzufügen ohne bestehende Flugverhalten oder Enten Klassen verändern zu müssen.

LÖSUNG TEIL 2



- Let's code

BEVORZUGE HAS-A (KOMPOSITION) GEGENÜBER IS-A (VERERBUNG)

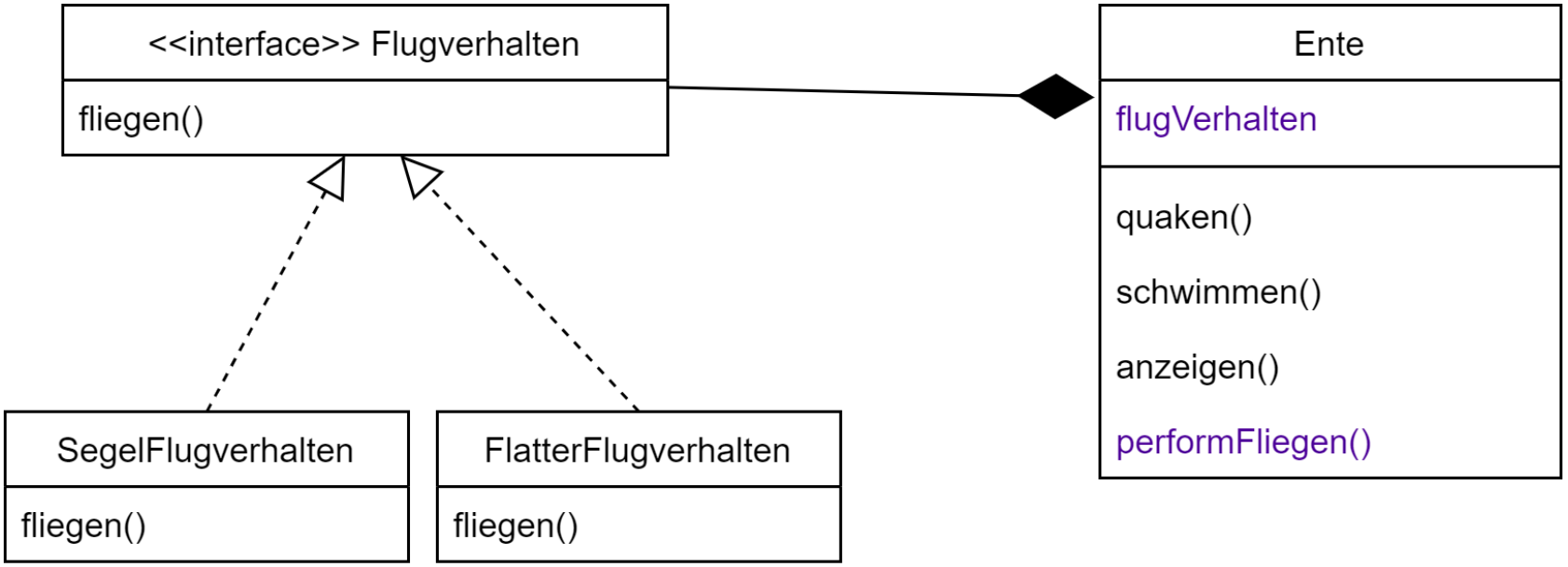
- Warum?
  - Flexibilität
    - Familie von Algorithmen während Laufzeit austauschen
- Komposition wird oft in Patterns verwendet

GRATULATION

STRATEGY PATTERN

Das Strategy Pattern erlaubt es einen **Algorithmus** (Familie von Algorithmen) unabhängig von der ihn nutzenden Klasse (= Client) zu **variieren**.

STRATEGY PATTERN

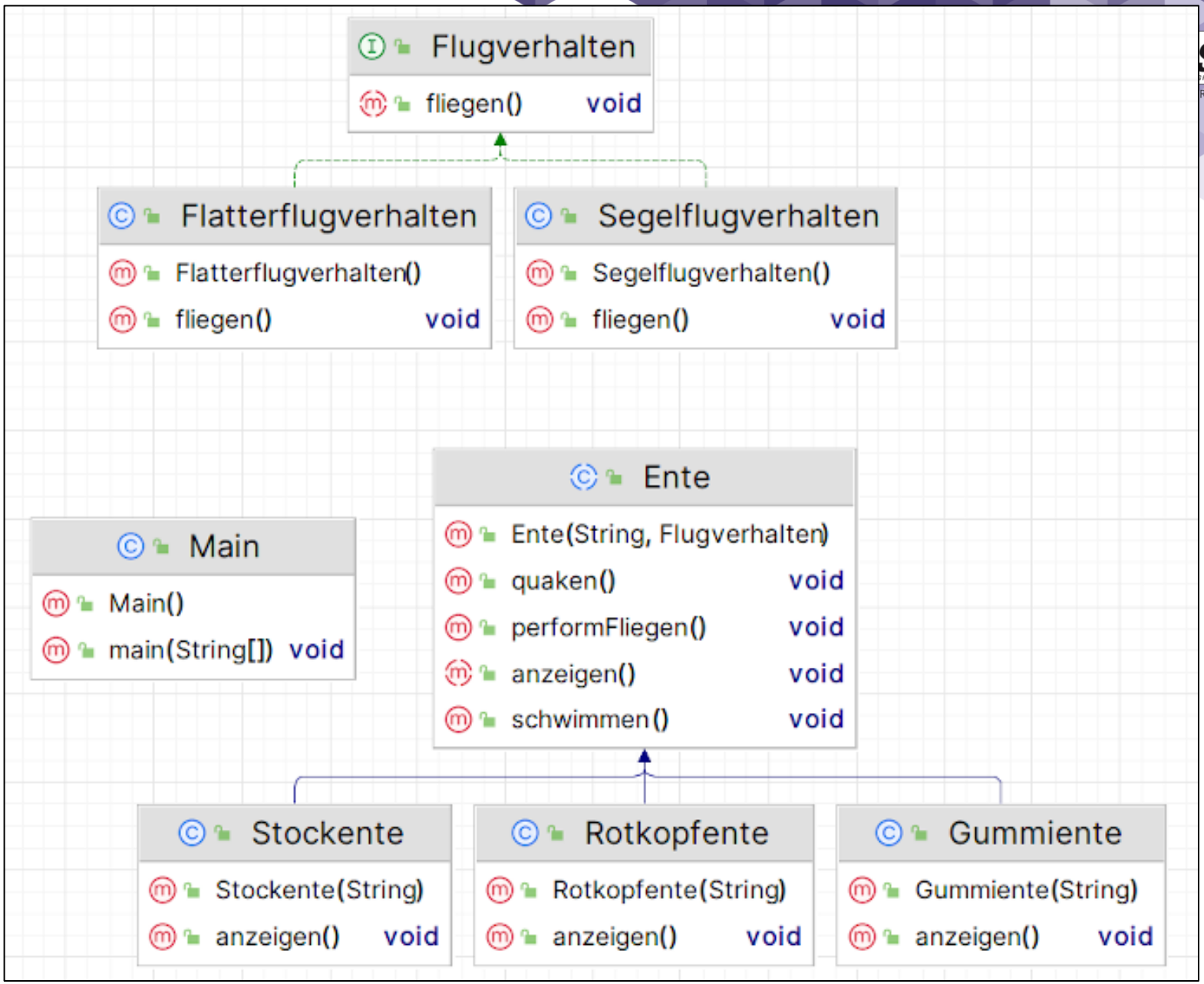


WEITERE DESIGN PRINZIPIEN: LOSE KOPPLUNG

Teilsysteme sind lose gekoppelt, wenn ihre Abhängigkeiten zueinander minimal sind. Die Klasse Ente kennt beispielsweise lediglich das Interface Flugverhalten (und Quackverhalten), jedoch nicht deren konkreten Implementierungen. Somit können letztere einfach ausgetauscht werden, ohne dass eine Änderung der Klasse Ente von Nöten wäre.

WEITERE DESIGN PRINZIPIEN: KOHÄRENZ

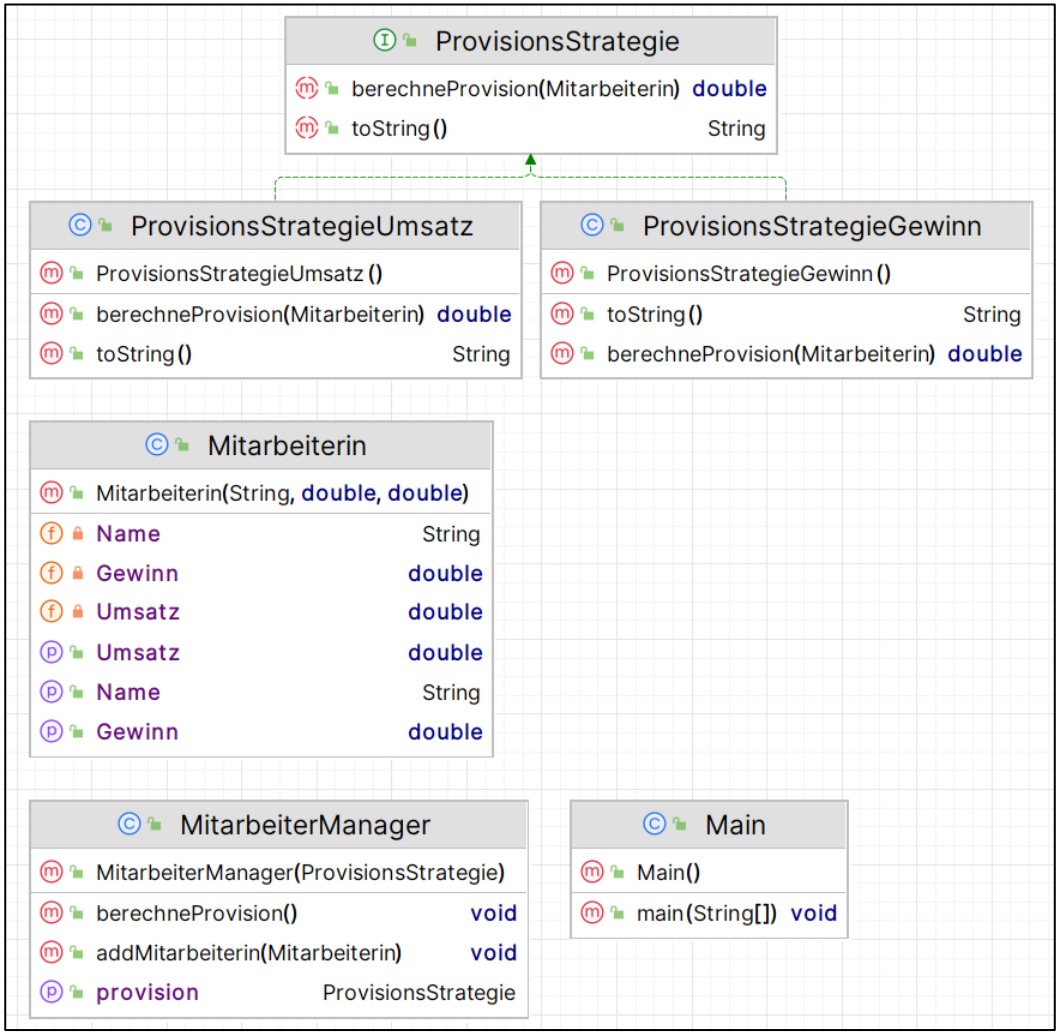
- Jedes Teilsystem sollte eine klar umrissene Verantwortung haben.
  - Dies vermindert die Komplexität des Systems und vereinfacht Wartung


Flugverhalten
fliegen() void
Flatterflugverhalten
Flatterflugverhalten()
fliegen() void
Segelflugverhalten
Segelflugverhalten()
fliegen() void
Ente
Ente(String, Flugverhalten)
quaken() void
performFliegen() void
anzeigen() void
schwimmen() void
Main
Main()
main(String[]) void
Stockente
Stockente(String)
anzeigen() void
Rotkopfente
Rotkopfente(String)
anzeigen() void
Gummiente
Gummiente(String)
anzeigen() void

ÜBUNGSBEISPIEL

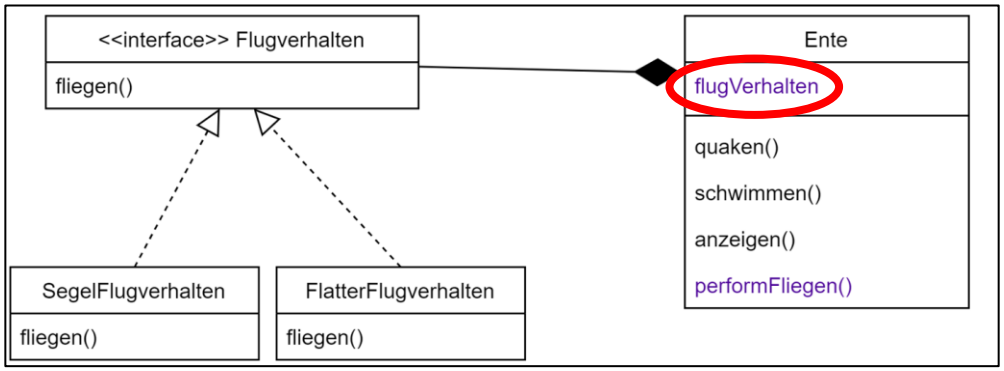
- Mitarbeiter*in besitzen die Eigenschaften Name, Umsatz und Gewinn.
- MitarbeiterManager verwaltet Mitarbeiter*innen in einer Liste und bietet weiters eine Methode um ihre Provisionen zu berechnen (=einfach ausgeben).
  - Die Provision für alle Mitarbeiter*innen ist entweder 10% des Gewinns oder 5% des Umsatzes.
  - Manager soll eine Methode aufweisen um die Provision währen der Laufzeit des Programms zu verändern.

MÖGLICHE LÖSUNG ÜBUNGSBEISPIEL „PROVISION“



ÜBUNGSBEISPIEL

- Mitarbeiter*in besitzen die Eigenschaften Name, Umsatz und Gewinn.
- MitarbeiterManager verwaltet Mitarbeiter*innen in einer Liste und bietet weiters eine Methode um ihre Provisionen zu berechnen (=einfach ausgeben).
  - Die Provision für alle Mitarbeiter*innen ist entweder 10% des Gewinns oder 5% des Umsatzes.
  - Manager soll eine Methode aufweisen um die Provision während der Laufzeit des Programms zu verändern. (nur der Manager hat eine Provisions-Strategie, MA haben nur Name, Umsatz und Gewinn)



ÜBUNG „TRANSPORTER“

- Angabe siehe Moodle
- Klassendiagramm der Lösung: