Software Engineering

Entwurfsmuster

Prof. Dr. Bodo Kraft

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Lernziele

Wir verstehen den **Zweck von Entwurfsmustern**

Wir kennen **ausgewählte Entwurfsmuster** und wissen, wie wir diese in unsere Entwürfe **integrieren** können

Wir können für unsere **spezifische Situation passende Muster suchen**, diese verstehen und verwenden

Architekturmuster – Entwurfsmuster – Idiome
Allgemeines

Muster bei der Software-Architektur

- Wie wird das System in Subsysteme/Module unterteilt?

Muster im Software-Design

- Welche Klassen/Objekte werden erstellt?
- Welche Schnittstellen werden definiert?
- Wie kommunizieren die Objekte?

Muster bei der Implementierung

- Programmiersprachen-spezifisch
- Namenskonventionen für Klassen, Attribute, Methoden
- Formatierung von Quellcode zur besseren Les- und Wartbarkeit

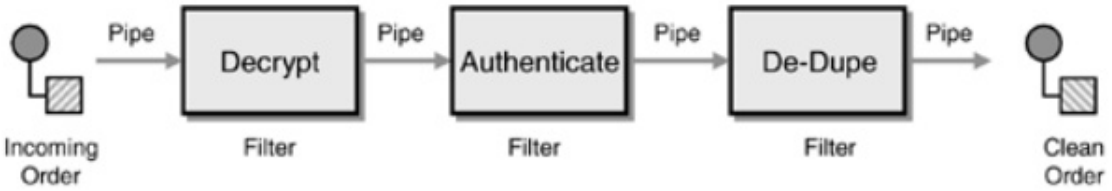
Architekturmuster – Entwurfsmuster – Idiome
Allgemeines

Muster bei der Software-Architektur

- Wie wird das System in Subsysteme/Module unterteilt?

Beispiele

- Model-View-Controller
- Layers
- **Pipes and Filters**
- Broker



Quelle: Pipes & Filters Architectural Pattern, Fredrik Kivi
https://www.slideshare.net/FredrikKivi/pipes-filters-architectural-pattern

Architekturmuster – Entwurfsmuster – Idiome
Allgemeines

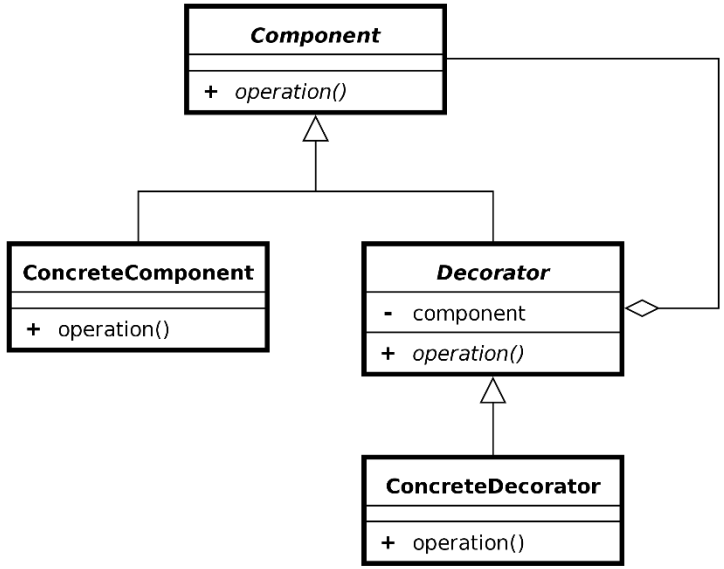
Muster im Software-Design

- Welche Klassen/Objekte werden erstellt?
- Welche Schnittstellen werden definiert?
- Wie kommunizieren die Objekte?

Beispiele

- Strategy (Strategie)
- Singleton (Einzelstück)
- Observer (Beobachter)
- Iterator (Iterator)

- **Decorator (Dekorierer)**



Quelle: Design Pattern, E. Gamma et. al.

Architekturmuster – Entwurfsmuster – Idiome
Allgemeines

Muster bei der Implementierung

- Programmiersprachen-spezifisch
- Namenskonventionen für Klassen, Attribute, Methoden
- Formatierung von Quellcode zur besseren Les- und Wartbarkeit

Beispiele

- CamelCaseBennenung `class StockNewsAnalyser`
- Verb + Substantiv `berechneMahnkosten()`
- Standardisierte Präfixe `interface ICalculate`
- Javadoc `/** …. **/`
- Und vieles mehr …

Ziele eines ‚guten' SW-Designs
Grundlagen

Leichte Erweiterbarkeit

Gute Wartbarkeit

Modularität

Wiederverwertbarkeit

...

wiederholende Probleme beim konkreten SW-Design

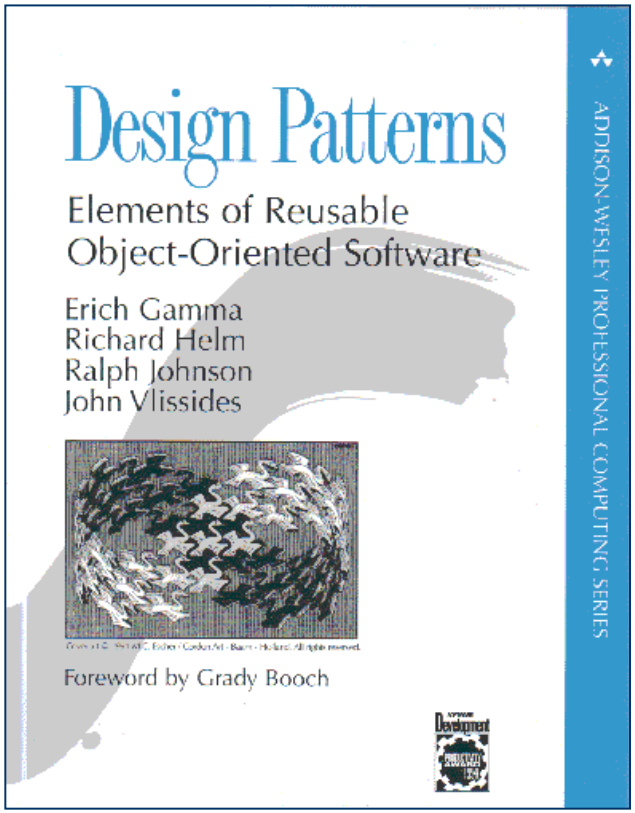
Software-Entwurfsmuster

Definition der SW-Entwurfsmuster
Grundlagen

Entwurfsmuster ...

- sind **wohlüberlegte Vorschläge** für ein „gutes" Software-Design
- beschreiben in **rezeptartiger Weise** das Zusammenwirken von Klassen, Objekten und Methoden
- stellen **bewährte, standardisierte Lösungen** unter einem definierten Namen dar
- sind **keine** SW-Bibliothek mit fertigen Lösungen!

Literatur zu Mustern



1995: Design Patterns

„Gang of Four" (GoF)

Literatur zu Mustern

Sehr gut zum Selbststudium geeignet und gut zu lesen!

1. Auflage Dez 2005
ca. 48,00 €

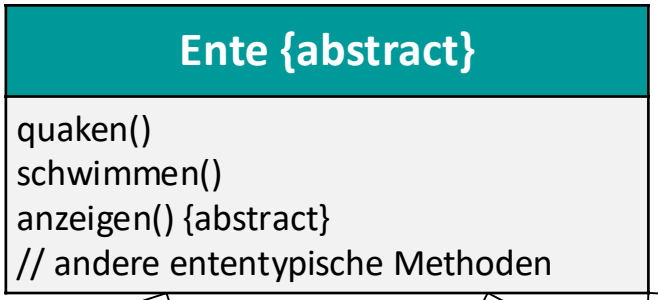
Kategorien von Entwurfsmustern

Erzeugung	Verhalten	Struktur
Fabrikmethode	Schablone	Proxy
Abstrakte Fabrik	**Beobachter**	**Kompositum**
Erbauer	Iterator	Adapter
Prototyp	Besucher	**Fassade**
Singleton	**Strategie**	**Dekorierer**
	Zustand Befehl	

Computerspiel „Duck City“
Einführung in Entwurfsmuster

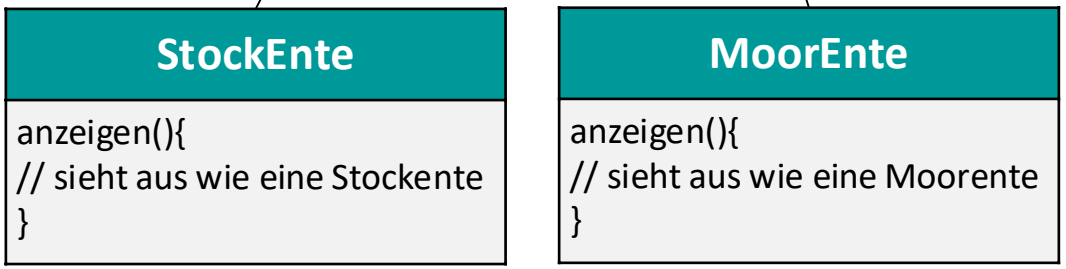
Die Methode `anzeigen()` ist abstrakt, da alle Untertypen jeweils anders aussehen.

Alle Enten quaken und schwimmen. Die Superklasse kümmert sich um den Implementierungscode.



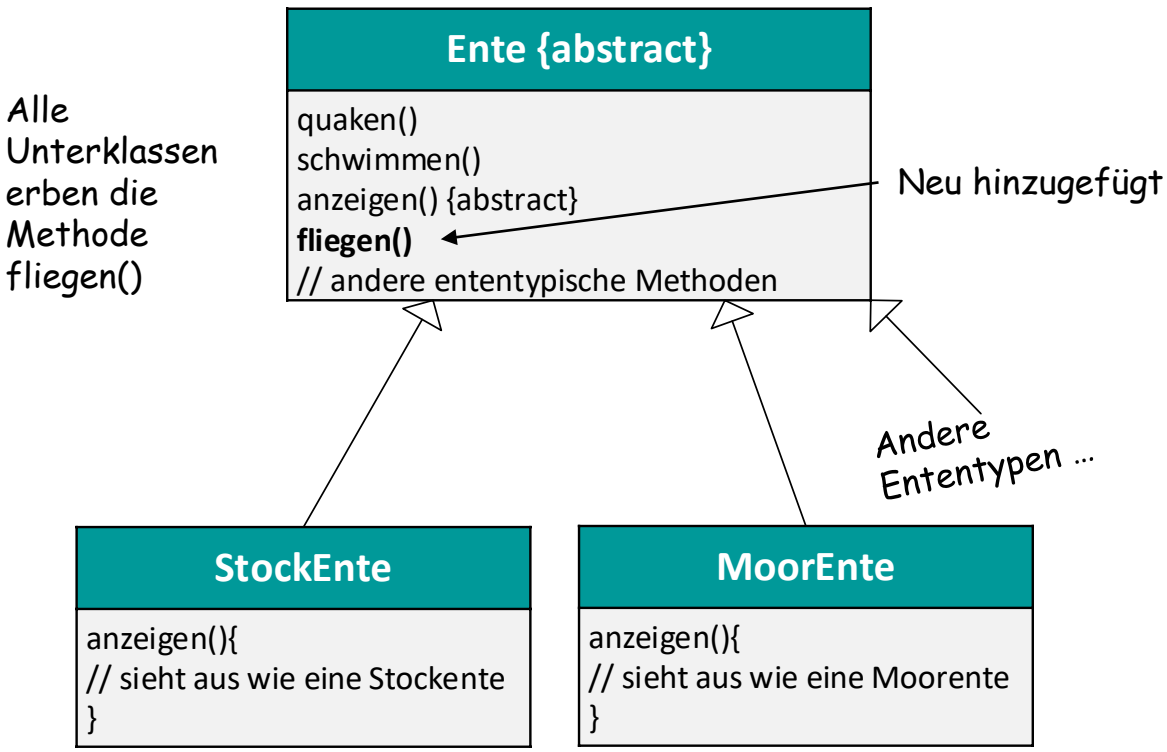
Jeder Untertyp von Ente ist selbst für die Implementierung seines anzeigen()-Verhaltens verantwortlich, das festlegt, wie er auf dem Bildschirm aussieht.

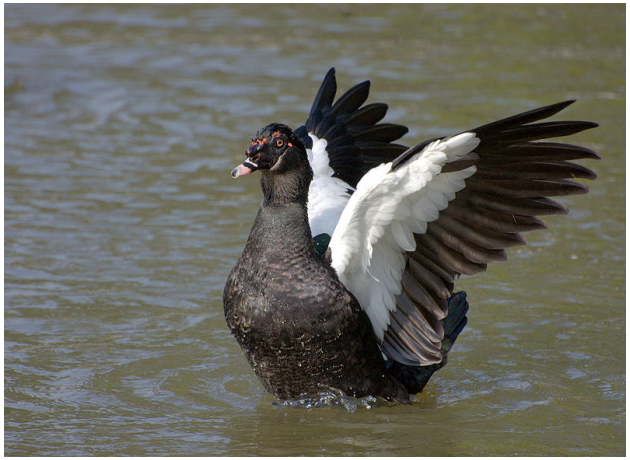
Viele andere Typen von Enten erben von der Superklasse Ente.



Weitere Funktionalität in Oberklasse
Computerspiel „Duck City“

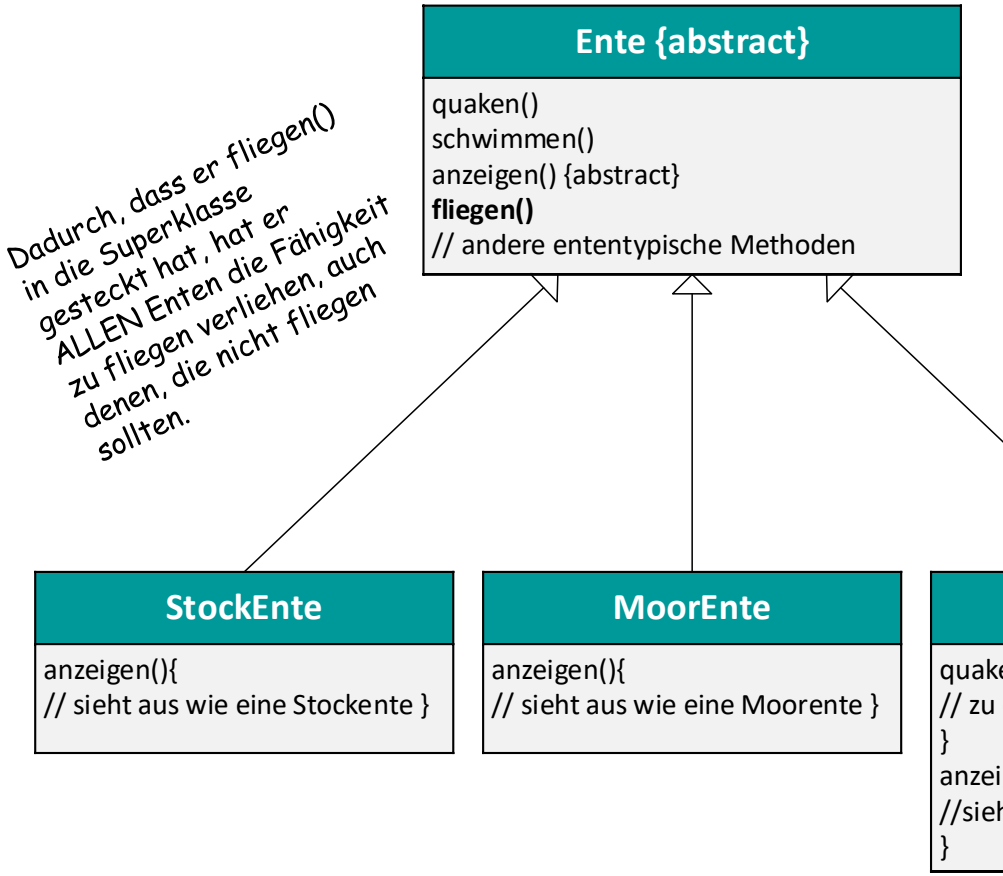
Neue Anforderung des Auftraggebers
→ Enten müssen fliegen können!





Auswirkung bei Hinzunahme weiterer Unterklassen
Computerspiel „Duck City“

Neue Anforderung des Auftraggebers
→ Es soll auch Gummi-Enten geben



Dadurch, dass er fliegen() in die Superklasse gesteckt hat, hat er ALLEN Enten die Fähigkeit zu fliegen verliehen, auch denen, die nicht fliegen sollten.

Bewertung
\+ Wiederverwendung

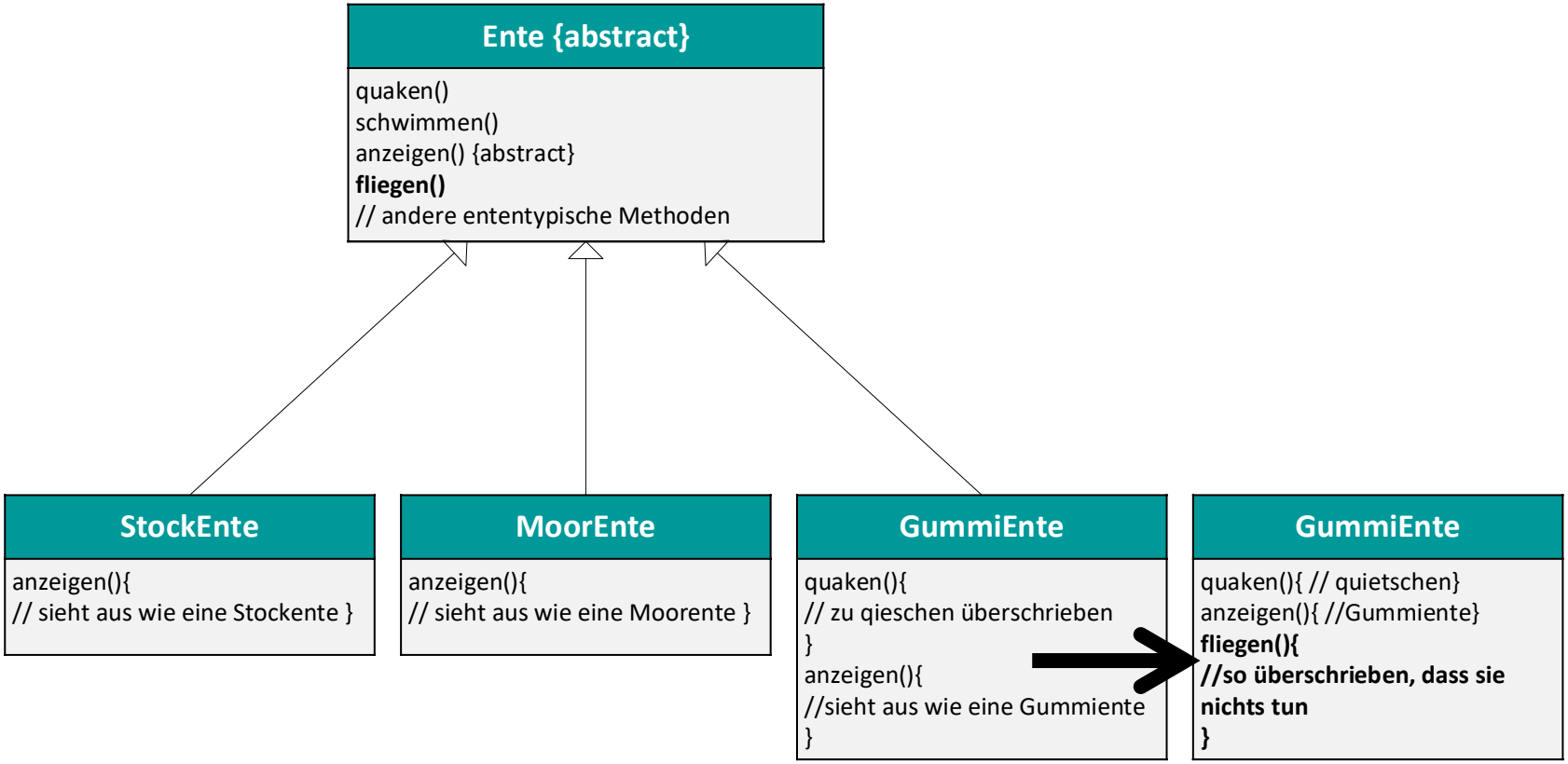
- Unklares Verhalten `fliegen`
- Unklares Verhalten `quaken`

- Unsaubere Implementierung

Gummienten quaken nicht, also wurde quaken() zu >>quietschen<< überschrieben

Auswirkung bei Hinzunahme weiterer Unterklassen
Computerspiel „Duck City“

Modellierung

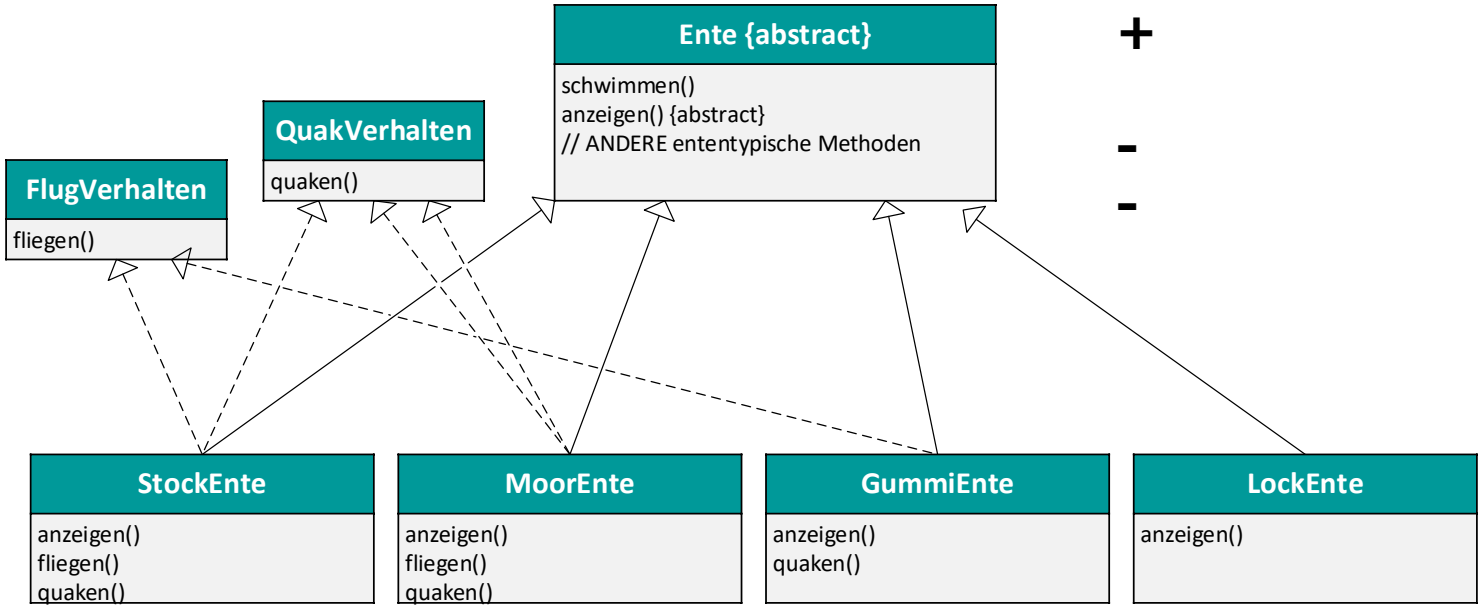


Alternative: Methode fliegen() wird auch überschrieben

Besser?

Alternative mit Interfaces
Computerspiel „Duck City“

Modellierung



Bewertung

\+

\+

\-

\-

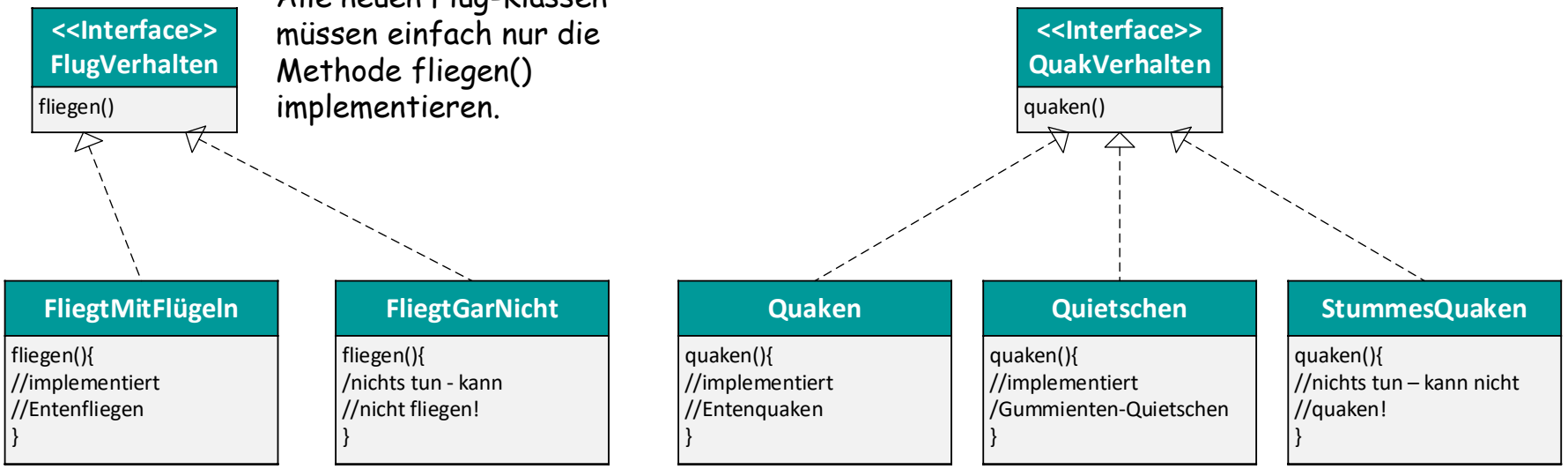
Was halten SIE von diesem Entwurf?

Auf dem Weg zur Lösung
Computerspiel „Duck City“

Modellierung der „besonderen Funktionalität“

Hier haben wir ein Interface, das alle Flug-Klassen implementiert. Alle neuen Flug-Klassen müssen einfach nur die Methode fliegen() implementieren.

Hier haben wir das Gleiche für das Quakverhalten. Wir haben ein Interface, das nur eine quaken()-Methode enthält, die implementiert werden muss.



Implementierung für das Fliegen für alle Enten, die Flügel haben.

Implementierung für alle Enten, die **nicht** fliegen können.

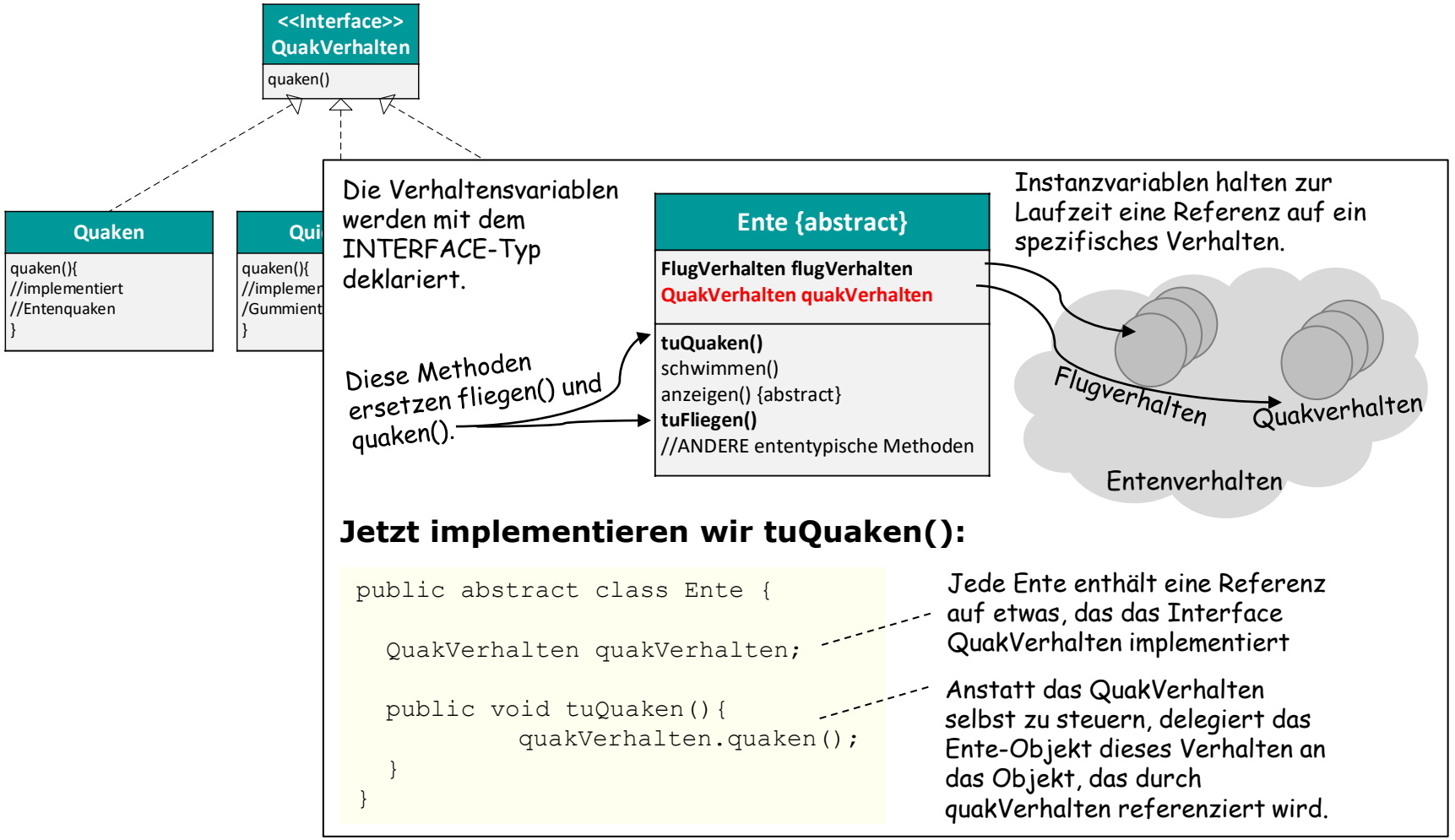
Ein Quaken, das richtig quakt.

Ein Quaken, das quietscht.

Ein Quaken, das kein Geräusch macht.

Auf dem Weg zur Lösung – Delegation
Computerspiel „Duck City“

Modellierung der „besonderen Funktionalität“



Jetzt implementieren wir tuQuaken():

```
public abstract class Ente {

  QuakVerhalten quakVerhalten;

  public void tuQuaken(){
          quakVerhalten.quaken();
  }
}
```

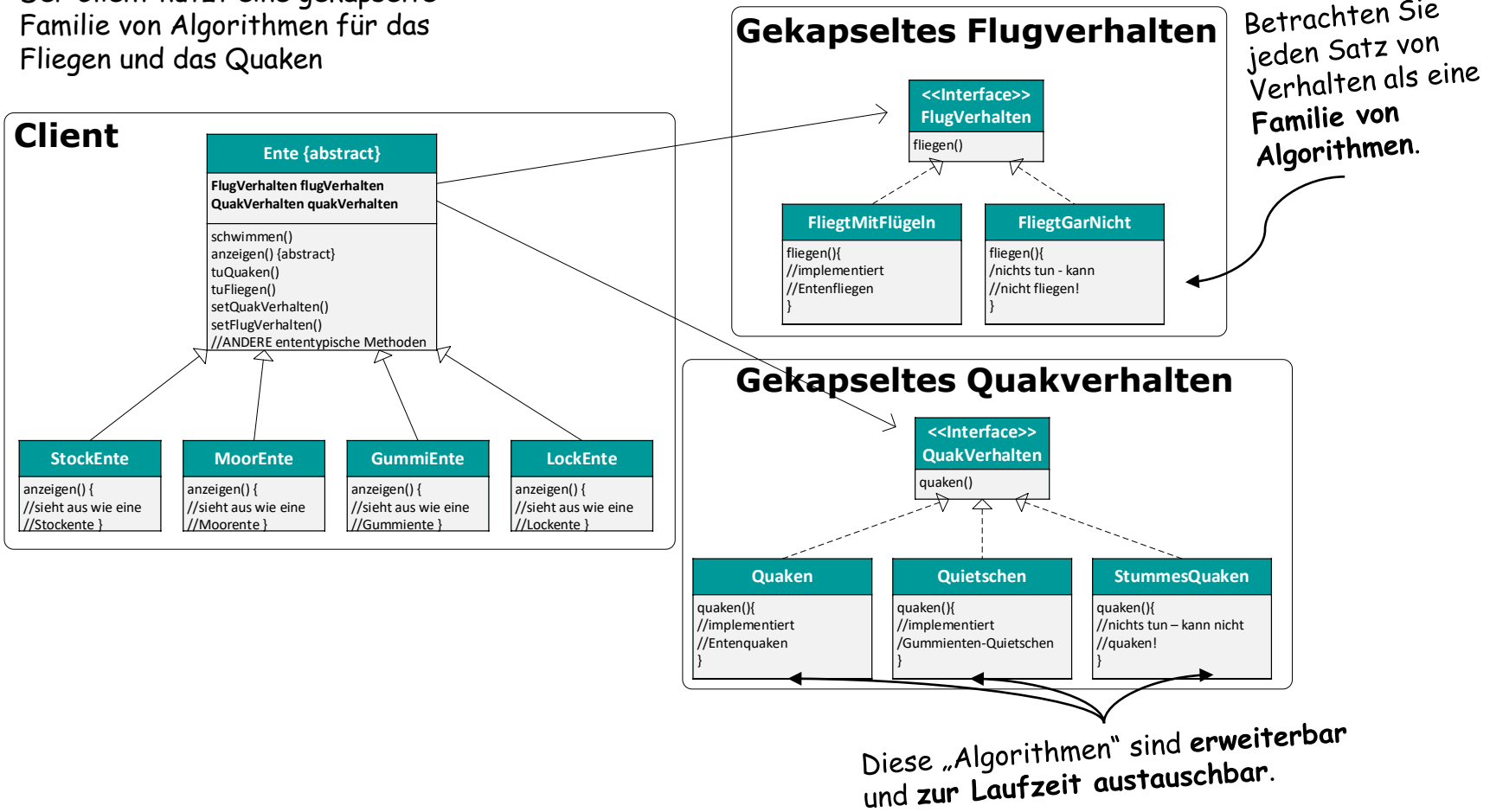
Jede Ente enthält eine Referenz auf etwas, das das Interface QuakVerhalten implementiert

Anstatt das QuakVerhalten selbst zu steuern, delegiert das Ente-Objekt dieses Verhalten an das Objekt, das durch quakVerhalten referenziert wird.

Auf dem Weg zur Lösung
Computerspiel „Duck City“

Vollständige Architektur

Der Client nutzt eine gekapselte Familie von Algorithmen für das Fliegen und das Quaken

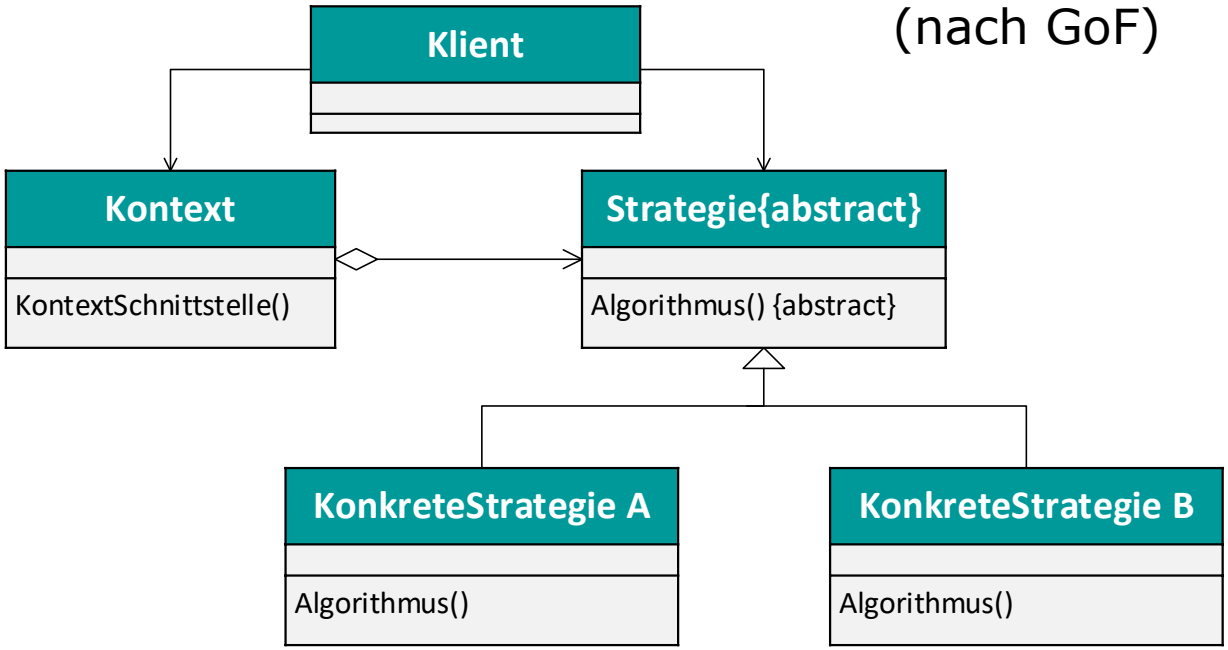


Das Strategie Pattern
Einführung in Entwurfsmuster

„Das Strategie-Muster definiert eine **Familie von Algorithmen**, kapselt sie einzeln und macht sie austauschbar.“

„Das Strategie-Muster ermöglicht es, den **Algorithmus unabhängig von ihn nutzenden Klienten** (auch zur Laufzeit!) **zu variieren**.“

(nach GoF)



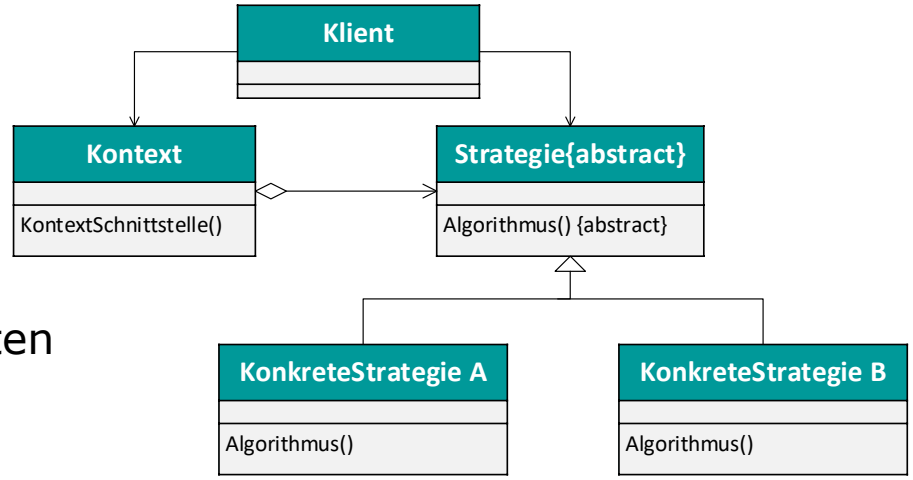
Zusammenfassung: Strategie Pattern
Einführung in Entwurfsmuster

Verwendung

- viele verwandte Klassen, die sich nur in ihrem Verhalten unterscheiden
- unterschiedliche (austauschbare) Varianten eines Algorithmus benötigen

Weitere Beispiele

- **Steuerberechnungsprogramm:**
  Berechnung von Steuersätzen auslagern
- **Speicherung:**
  Grafik in verschiedenen Dateiformaten
- **Packer:**
  verschiedene Kompressionsalgorithmen
- **Sudoku:** verschiedene Lösungsstrategien



Prinzipien: Trenne Veränderliches von Unveränderlichem

Einführung in Entwurfsmuster

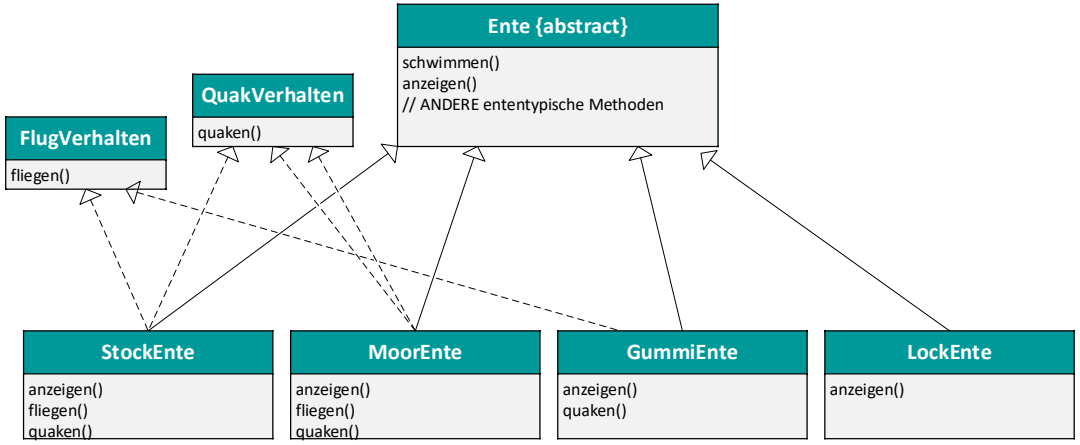
Trenne Veränderliches von Unveränderlichem

Identifizieren Sie die Aspekte Ihrer Anwendung, die sich ändern können und trennen Sie sie von denen, die konstant bleiben

Motivation

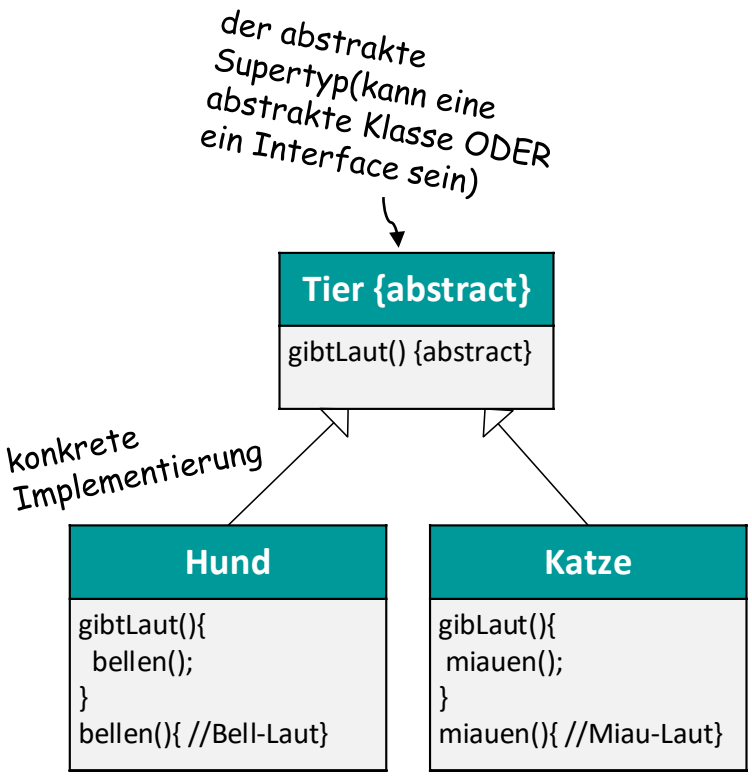
- Keine Verdopplung von Quellcode
- Einfache Erweiterung/Veränderung ihres Codes

Negativ-Beispiel



Prinzipien: Programmiere auf einer Schnittstelle
Einführung in Entwurfsmuster

Programmiere auf einer Schnittstelle



Schnittstelle
→ Konzept des nach außen öffentlichen Teils einer Komponente

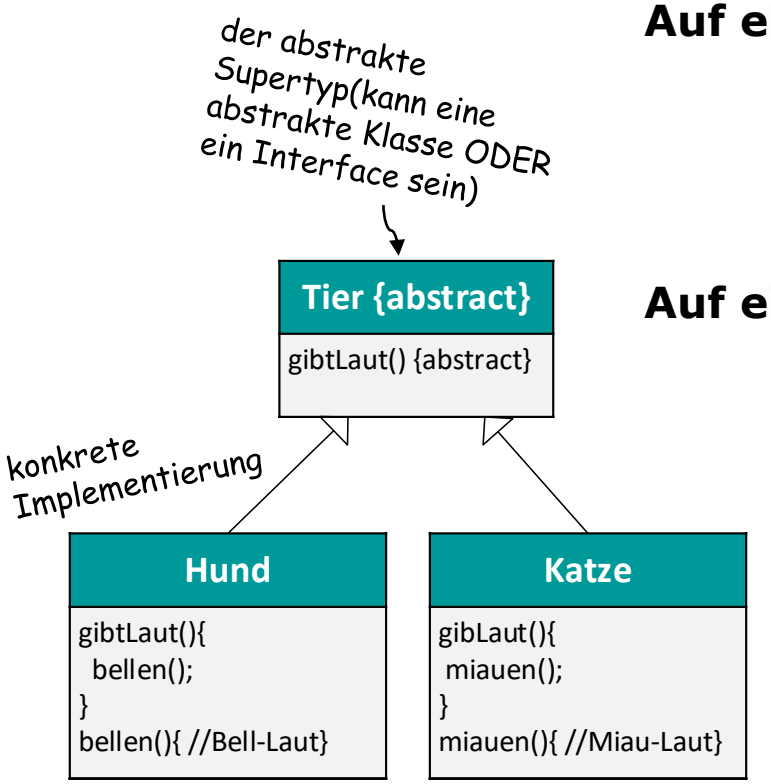
Interface
→ Konkrete Implementierung als Java-Interface

Motivation
Verwendung von Polymorphie durch Programmierung auf Supertyp

Verwendende Klasse kennt konkrete Implementierung nicht

Prinzipien: Programmiere auf einer Schnittstelle
Einführung in Entwurfsmuster

Programmiere auf einer Schnittstelle



Auf einer Implementierung programmieren

```
Hund meinHund = new Hund();
meinHund.bellen();
```

Auf einer Schnittstelle/Supertyp programmieren

```
Tier meinTier = new Hund();
meinTier.gibLaut();
```

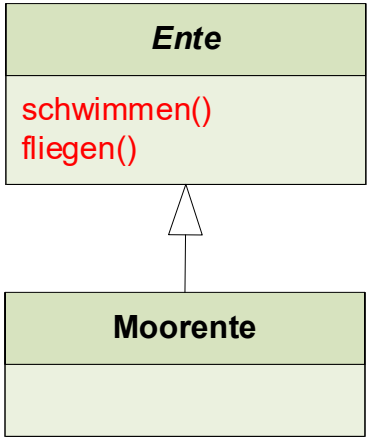
Noch besser...

```
Tier meinTier = null;
meinTier = Factory.getTier();
meinTier.gibLaut();
```

Prinzipien: Ziehe Delegation der Vererbung vor
Einführung in Entwurfsmuster

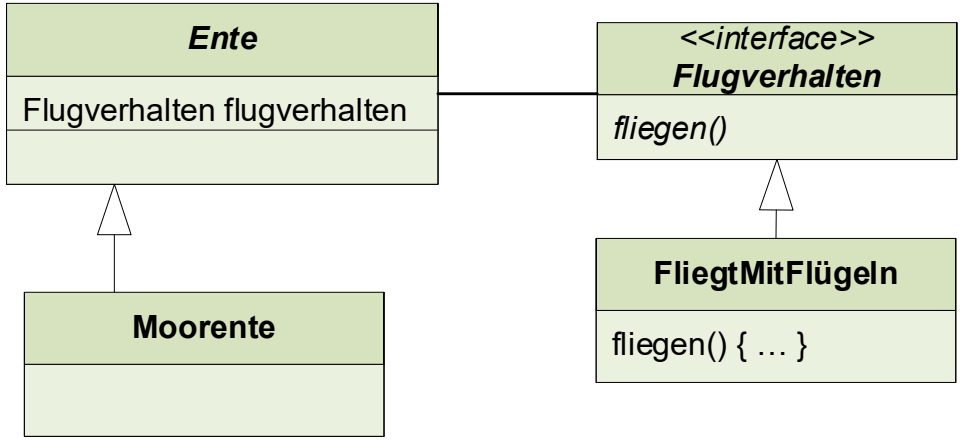
Ziehe Delegation der Vererbung vor

Vererbung



ist ein (is A)

Delegation



hat ein (has A) ← Komposition

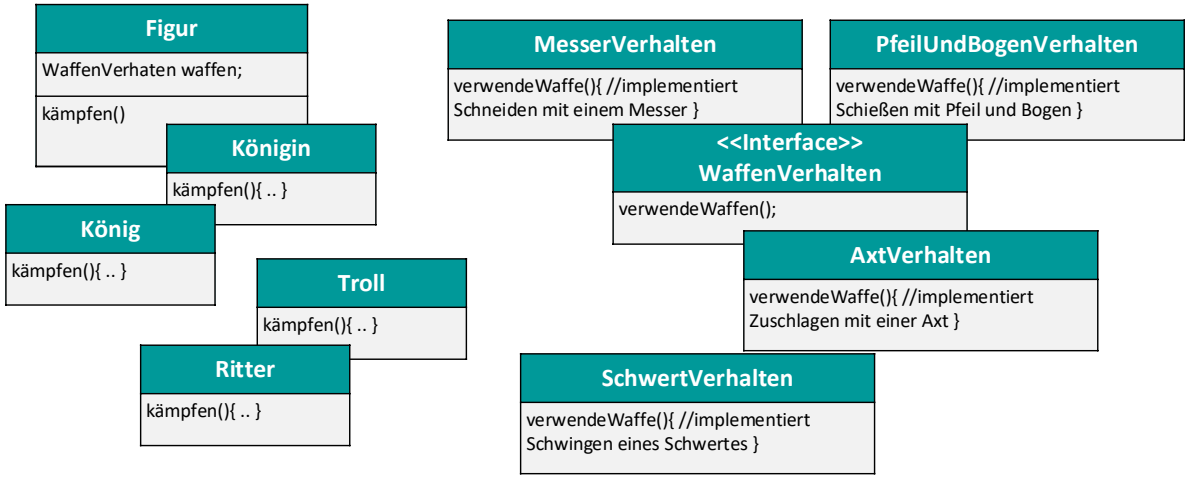
Vorteile
+ Kapselung einer Familien von Algorithmen
+ Erlaubt, das Verhalten zur Laufzeit zu verändern

Kurzübung „Action Adventure“
Einführung in Entwurfsmuster

- Klassen für Spielfiguren und Klassen für Waffenverhalten
- Jede Figur kann jeweils nur eine Waffe verwenden
- Die Waffe kann während des Spiels gewechselt werden

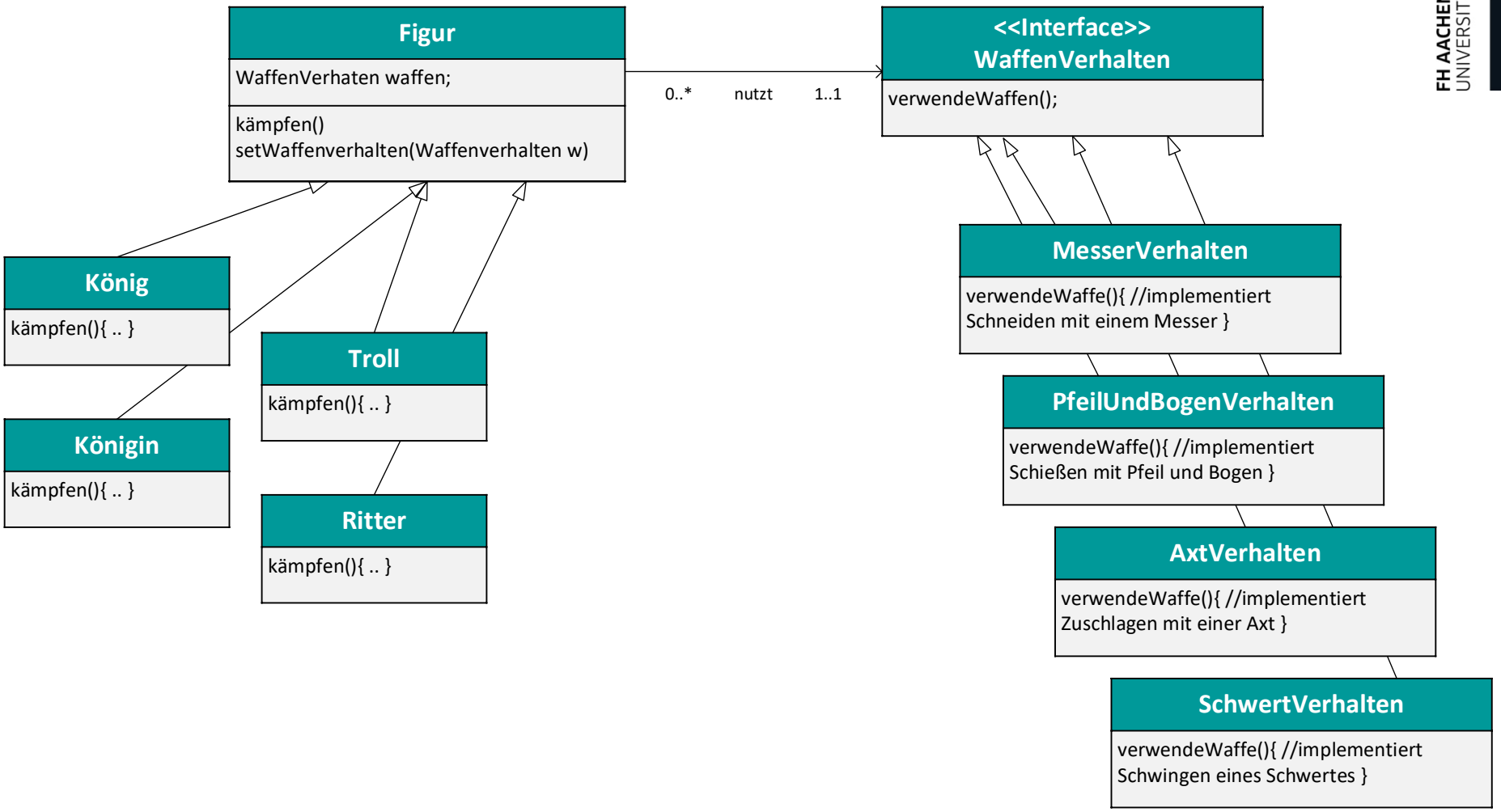
Aufgabe

- Ordnen Sie die Klassen und zeichnen Sie ein Klassendiagramm
- Fügen Sie die Methode setWaffe() in die richtige Klasse ein
- Welches Muster wird verwendet?




```
setWaffe(WaffenVerhalten w){
        this.waffe = w;
}
```

Kurzübung „Action Adventure“
Einführung in Entwurfsmuster



Kategorien von Entwurfsmustern

Erzeugung	Verhalten	Struktur
Fabrikmethode	Schablone	Proxy
Abstrakte Fabrik	**Beobachter**	**Kompositum**
Erbauer	Iterator	Adapter
Prototyp	Besucher	**Fassade**
Singleton	**Strategie**	**Dekorierer**
	Zustand Befehl	

Das Beobachter-Muster (Observer-Pattern)
Verhaltensmuster

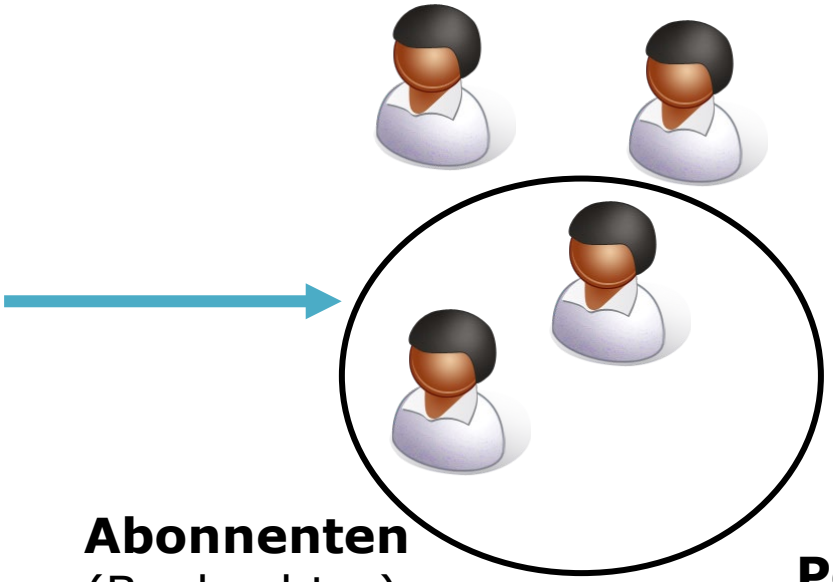
Das Observer-Muster

Definiert eine **Eins-zu-viele-Abhängigkeit** zwischen Objekten in der Art,

- dass alle **abhängigen Objekte benachrichtigt werden**,
- wenn sich der **Zustand des einen Objekts ändert**

Beispiel

Zeitungsverlag
(Subjekt)



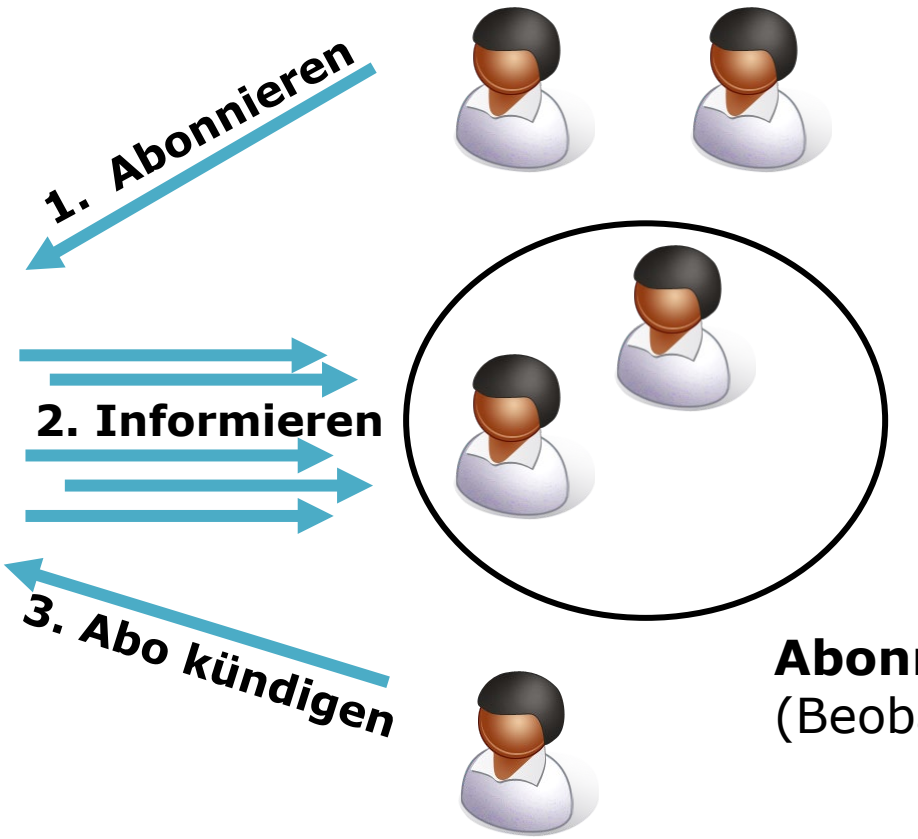
Abonnenten
(Beobachter)

Personen

Das Beobachter-Muster (Observer)
Verhaltensmuster

Beispiel

Zeitungsverlag
(Subjekt)



Abonnenten
(Beobachter)

Auftrag „Internet Wetterstation“
Verhaltensmuster

Gesamtes Szenario

- Unterschiedliche Sensoren liefern Werte
- Unterschiedliche Repräsentationen zur Darstellung

Wird geliefert

Zu Implementieren (weitere Anzeigen später)

Auftrag „Internet Wetterstation“
Verhaltensmuster

Gelieferte Schnittstelle

Diese drei Methoden liefern die neusten Wettermessungen für Temperatur, Luftfeuchtigkeit respektive Luftdruck

Uns interessiert nicht, WIE diese Variablen gesetzt werden. Das WetterDaten-Objekt weiß wie es aktualisierte Informationen von der Wetterstation erhält

Die Entwickler des WetterDaten-Objektes haben uns einen Hinweis hinterlassen, der angibt, was wir hinzufügen müssen …

Denken Sie daran: Aktuelle Wetterbedingungen ist nur einer von drei verschiedenen Anzeigebildschirmen

```
/*
 * Diese Methode wird jedes mal
 * aufgerufen, wenn die Wetter-
 * messung aktualisiert wurde.
 */
public void messwerteGeändert() {
    //Hier kommt Ihr Code rein.
}
```

WetterDaten.java

Auftrag „Internet Wetterstation“
Verhaltensmuster

Erster Entwurf

```
public class Wetterdaten

  // Deklaration der Instanzvariablen

  public void messwerteGeändert() {

    float temperatur = getTemperatur();
    float feuchtigkeit = getFeuchtigkeit();
    float luftdruck = getLuftdruck();

    aktuelleBedingungenAnzeige.aktualisieren(temperatur, feuchtigkeit, luftdruck);
    statistikAnzeige.aktualisieren(temperatur, feuchtigkeit, luftdruck);
    vorhersageAnzeige.aktualisieren(temperatur, feuchtigkeit, luftdruck);
  }

  // andere Wetterdaten-Methoden kommen hier herein.

}
```

Die frischen Messungen werden abgerufen, indem die bereits implementierten Getter-Methoden von WetterDaten aufgerufen werden.

Jetzt die Anzeigen aktualisieren...

Jedes Anzeigeelement wird aufgefordert, seine Anzeige zu aktualisieren. Dabei werden ihm die neuesten Messwerte übergeben.

Diskussion

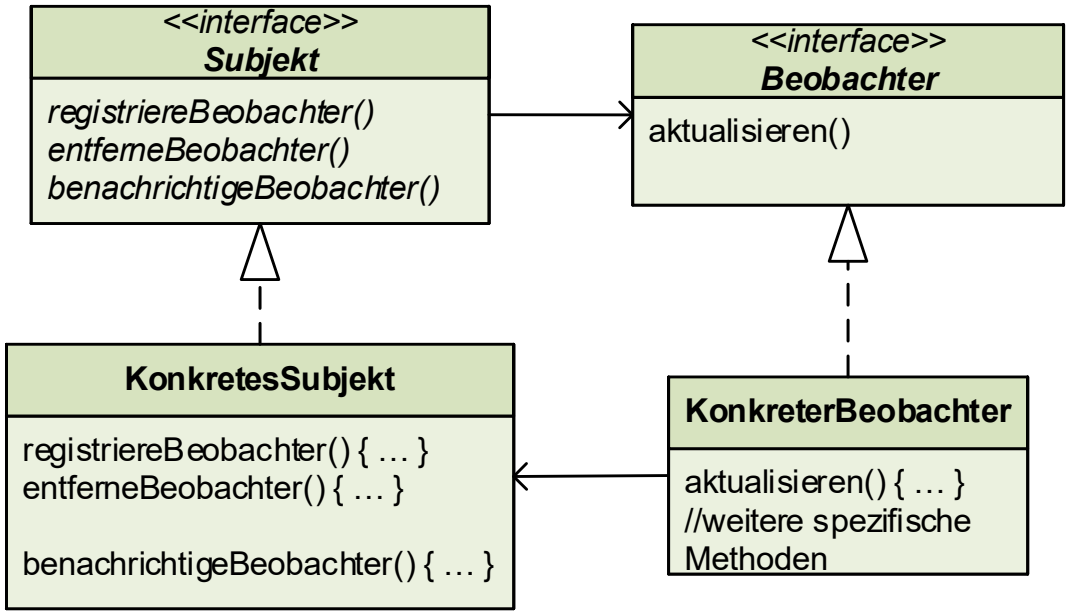
- Implementieren auf Schnittstelle oder Implementierung?
- Änderbarkeit zur Laufzeit gegeben (z.B. neue Anzeigegeräte)?

Beobachter-Muster als Klassendiagramm
Verhaltensmuster

Kapselung über Schnittstelle
- Subjekt
- Beobachter

Subjekt kennt Beobachter
- aktualisieren

KonkreterBeobachter kennt KonkretesSubjekt
- registrieren/entfernen



Wetterstation als Klassendiagramm

Einführung in Entwurfsmuster – Beobachter (Observer)

Wetterstation als Quellcode (1/2)
Einführung in Entwurfsmuster – Beobachter (Observer)


```
public interface Subject {


  public void registerObserver(
              Observer o);


  public void removeObserver(
              Observer o);


  public void notifyObservers();


}
```

```
public class WeatherData implements Subject {
  private List<Observer> observers;
  private float temperature, humidity, pressure;

  public WeatherData() {
    observers = new ArrayList<>();
  }

  public void registerObserver(Observer o) {
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    observers.remove(o);
  }

  public void notifyObservers() {
    for (Observer observer : observes) {
            observer.update
                (temperature, humidity, pressure);
   }


  public void measurementsChanged() {
    notifyObservers();
}
```

Wetterstation als Quellcode (2/2)
Einführung in Entwurfsmuster – Beobachter (Observer)


```
public interface Observer {

   public void update(
       float temperature,
       float humidity,
       float pressure);

}
```

```
public class CurrentConditionsDisplay implements Observer {

 private Subject weatherData;
 private float temperature;

 public CurrentConditionsDisplay(Subject weatherData) {
     this.weatherData = weatherData;
     weatherData.registerObserver(this);
 }

 public void update(float temperature, float humidity,
   float pressure) {

   // humidity, pressure not used here
   this.temperature = temperature;
   display();
 }

 public void display() {
   System.out.println("Currently“ + temperature);
 }

}
```

Zusammenfassung: Observer Pattern
Einführung in Entwurfsmuster

Verwendung

- Änderung des Zustands eines Objekts führt dazu,
  - dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden

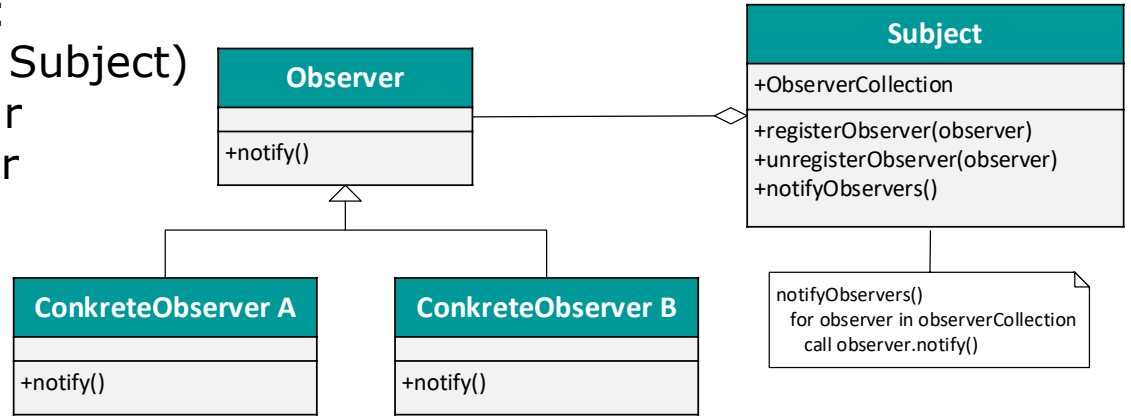
  ohne dass das geänderte Objekt die anderen genauer kennen muss

Weitere Beispiele

- Trennung von Business Logik und Repräsentation

In Java bereits definiert

- Klasse Observable (fürs Subject) mit Implementierung zur Verwaltung der Observer



Strebe nach loser Kopplung

Subjekt und Beobachter sind locker gebunden

- Subjekt weiß vom Beobachter nur, dass es ein Interface implementiert

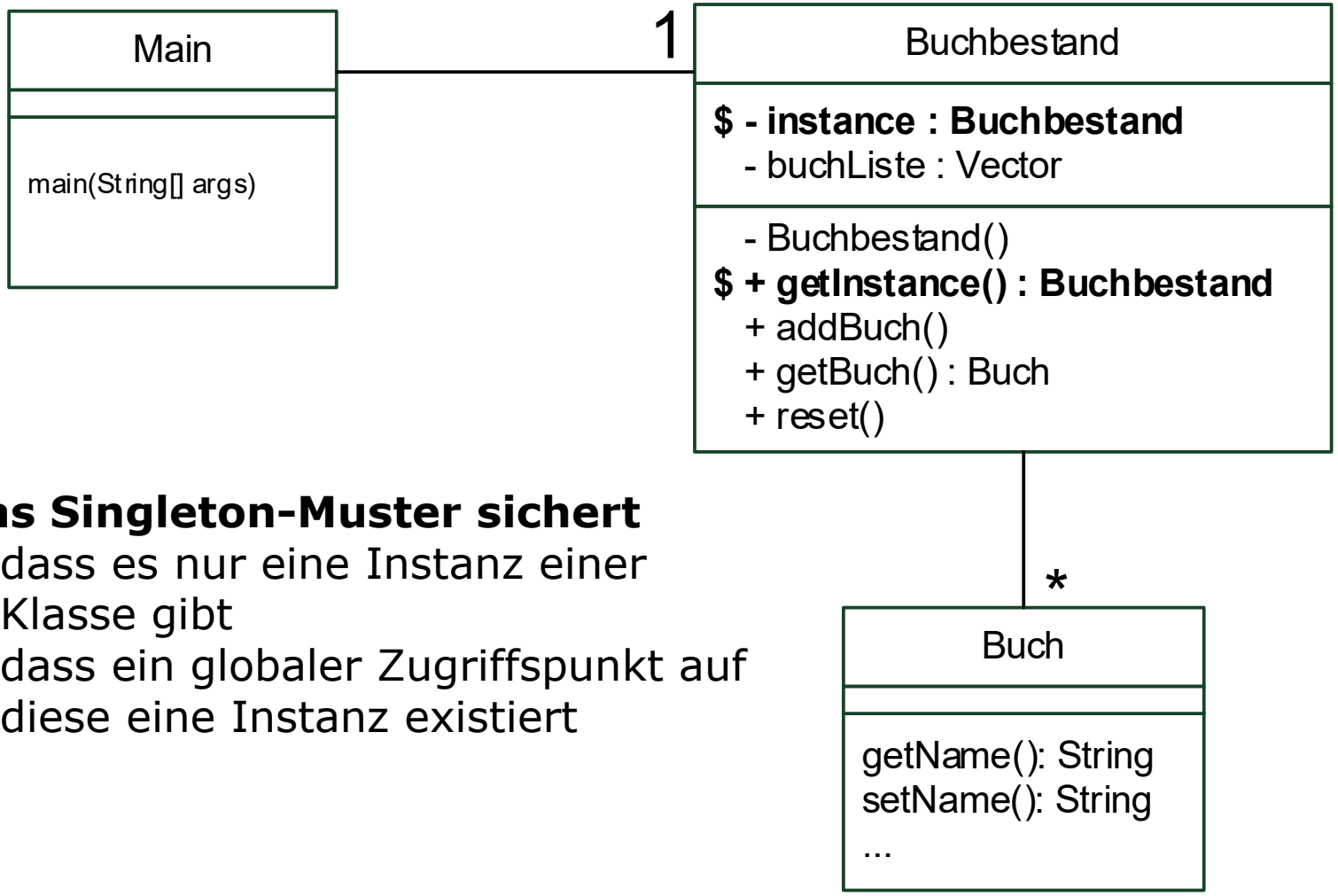
```
private Subject weatherData;
...
weatherData.registerObserver(this);
```

- Subjekt bleibt unverändert, wenn neue Beobachter dazu kommen:
  - Neuer Beobachter implementiert Interface
  - Neuer Beobachter registriert sich selbst
- Änderungen am Subjekt oder einem Beobachter haben keinen Einfluss auf den jeweils anderen
- Subjekt und Beobachter können unabhängig voneinander wieder verwendet werden

Kategorien von Entwurfsmustern

Erzeugung	Verhalten	Struktur
Fabrikmethode	Schablone	Proxy
Abstrakte Fabrik	**Beobachter**	**Kompositum**
Erbauer	Iterator	Adapter
Prototyp	Besucher	**Fassade**
Singleton	**Strategie**	**Dekorierer**
	Zustand Befehl	

Das Singleton-Pattern
Entwurfsmuster der Kategorie Erzeugung

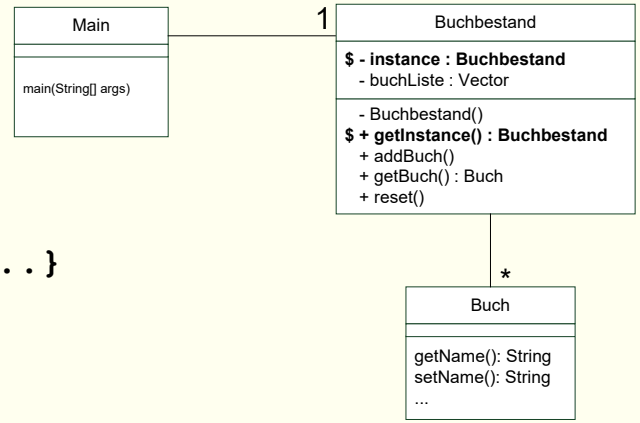


Das Singleton-Muster sichert

- dass es nur eine Instanz einer Klasse gibt
- dass ein globaler Zugriffspunkt auf diese eine Instanz existiert

Singleton – Codesicht des Singletons
Entwurfsmuster der Kategorie Erzeugung

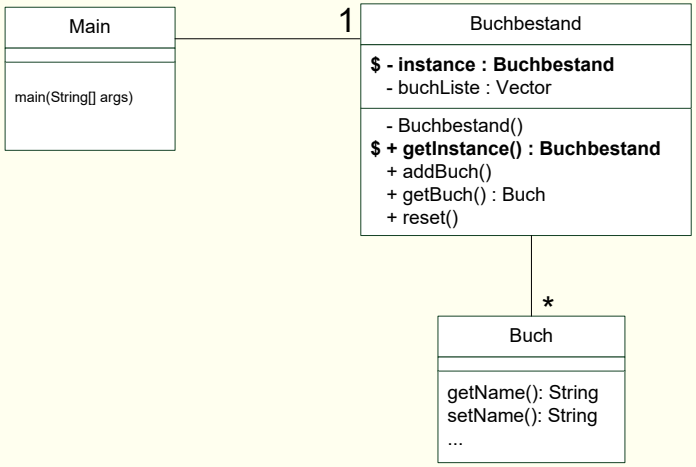

```
public class Buchbestand {

        // Klassenvariable speichert einzige Instanz. Wird direkt erzeugt.
        private static Buchbestand instance = new Buchbestand();

        // Instanzvariable
        private List<Buch> buchListe;

        // privater Konstruktor, kann nur innerhalb der Klasse aufgerufen werden
        private Buchbestand() {
                buchListe = new ArrayList<>();
        }

        // Klassenmethode, die DIE eine Instanz zurückliefert
        public static Buchbestand getInstance() {
                return instance;
        }

        public void addBuch(Buch buch) {
                buchListe.add(buch);
        }

        public Buch getBuch(String title) {...}

        public void reset() {...}

}
```



Singleton – Codesicht des Verwenders

Entwurfsmuster der Kategorie Erzeugung

```
public class Main {

  public static void main(String[] args) {

        Buch buch = new Buch();

        buch.setName("Matse-Klausurlösung 2006 bis 2024");


        // DIE eine Instanz zurückgeben lassen
        Buchbestand instance = Buchbestand.getInstance();

        // mit der Instanz arbeiten
        instance.addBuch(buch);

        ...

  }

}
```

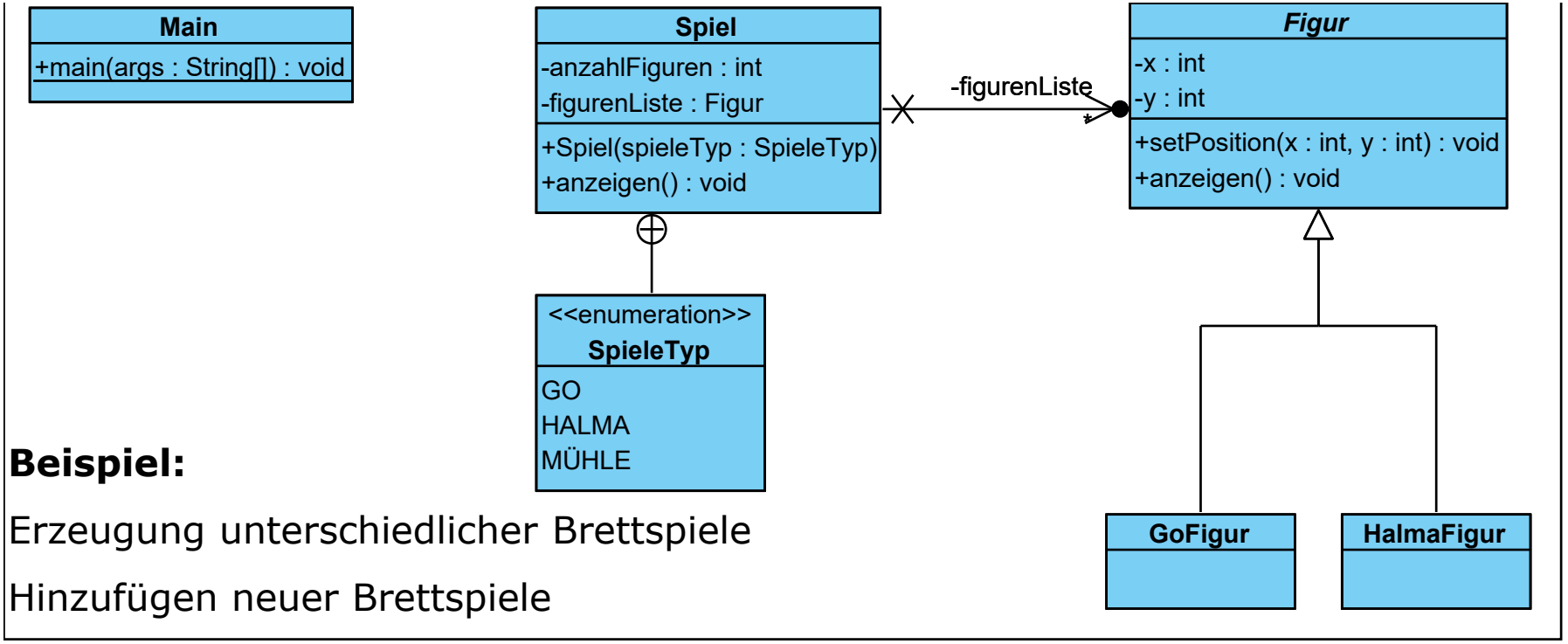


Fabrikmethode - Motivation
Entwurfsmuster der Kategorie Erzeugung

Herkömmliche Objekterzeugung - Entwurfssicht

Problem: Objekterzeugung mit **new**-Operator oft unzureichend

- **Entwurfsänderungen** ziehen **Codeänderungen** an vielen Stellen nach sich



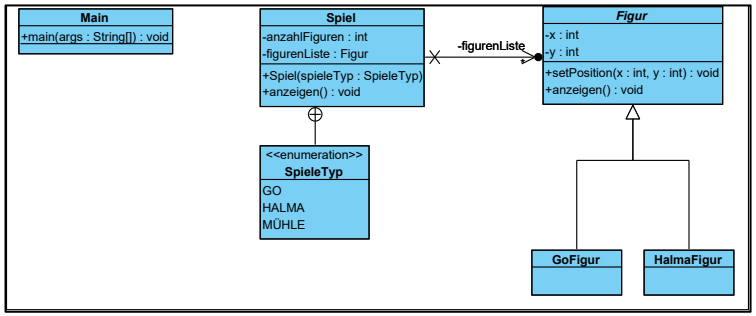
Beispiel:

Erzeugung unterschiedlicher Brettspiele

Hinzufügen neuer Brettspiele

Codesicht bei herkömmlicher Objekterzeugung
Entwurfsmuster der Kategorie Erzeugung

```
public class Spiel {
   public enum SpieleTyp {GO, HALMA, MÜHLE}

   private int anzahlFiguren;
   private List<Figur> figurenListe;

   public Spiel (SpieleTyp spieleTyp) {
      figurenListe = new ArrayList<Figur>();

      switch (spieleTyp) {
         case GO: {
            anzahlFiguren = 30;
            for (int i = 0; i < anzahlFiguren; i++) {
               Figur f = new GoFigur();
               f.setPosition(i, i);
               figurenListe.add(f);
            }
            break;
         }
         case HALMA: { … } …
```



Was ist an diesem Ansatz im Hinblick auf die bisherigen Prinzipien zu verbessern?

```
public static void main(String[] args) {
   Spiel halma = new Spiel(Spiel.SpieleTyp.HALMA);
   halma.anzeigen();

   Spiel go = new Spiel(Spiel.SpieleTyp.GO);
   go.anzeigen();
}
```

Probleme bei herkömmlicher Objekterzeugung
Entwurfsmuster der Kategorie Erzeugung

**Entwürfe sollten für Erweiterungen offen,
aber für Veränderungen geschlossen sein.**

→ Verstoß gegen das Open/Closed-Prinzip:

Das Hinzufügen von neuen Spielen oder anderer Figuren erfordert die Modifikation der Klasse `Spiel`

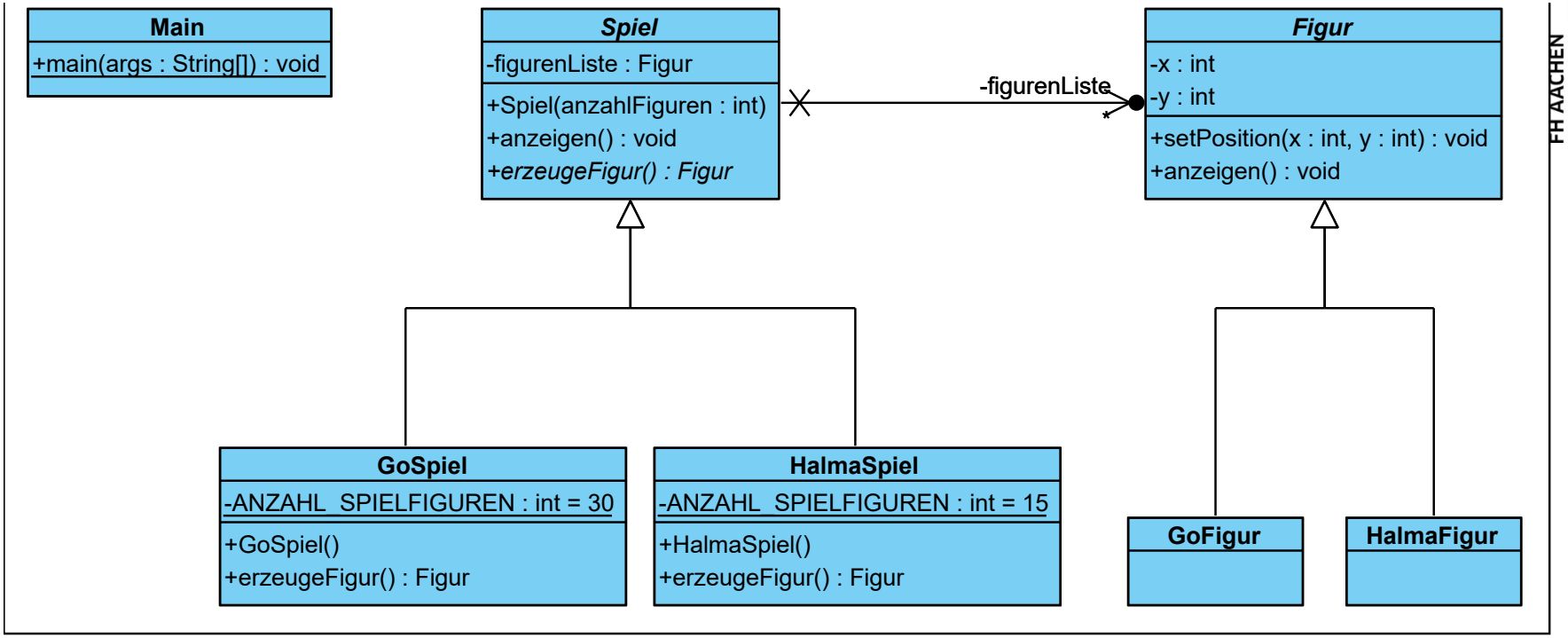
**Identifiziere jene Aspekte, die sich ändern und
trenne sie von jenen, die konstant bleiben.**

→ Keine Kapselung von dem was sich ändert:

Die Erzeugung der Figuren ist unterschiedlich, das Speichern der Figuren in einer Liste aber immer gleich…

Lösung: Kapsele die Erzeugung der Figuren

Fabrikmethode zur Kapselung der Erzeugung
Entwurfsmuster der Kategorie Erzeugung

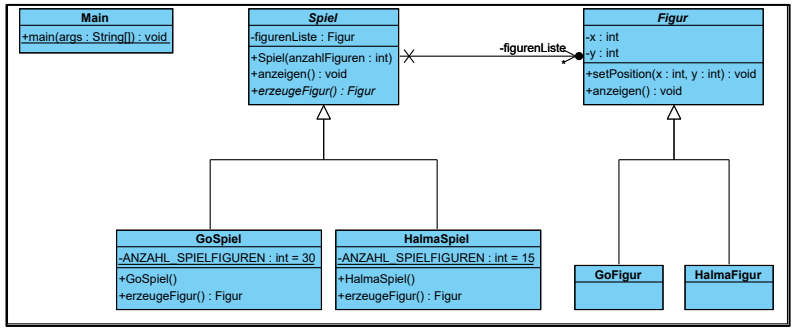


- **Abstrakte Klasse *`Spiel`***
  - Methode ***`erzeugeFigur()`*** sichert einheitlichen Zugriff
  - Speicherung der Spielfiguren generisch
- Objekterzeugung erfolgt über konkrete Unterklassen

Codesicht bei Factory-Methode
Entwurfsmuster der Kategorie Erzeugung

```
public abstract class Spiel {
   private List<Figur> figurenListe;

   protected Spiel(int anzahlFiguren) {
      figurenListe = new ArrayList<Figur>();

      for (int i = 0; i < anzahlFiguren; i++) {
          Figur spielFigur = erzeugeFigur();
      …
```

```
public class GoSpiel extends Spiel {
   private static final int ANZAHL_FIGUREN = 30;

   public GoSpiel() {
      super(ANZAHL_FIGUREN);
   }

   @Override
   public Figur erzeugeFigur() {
     return new GoFigur();
   } …
```



Erweiterung um neue Spiele

- Zwei neue Unterklassen (für Spiel und Figur)
- Verwendung in der nutzenden Klasse

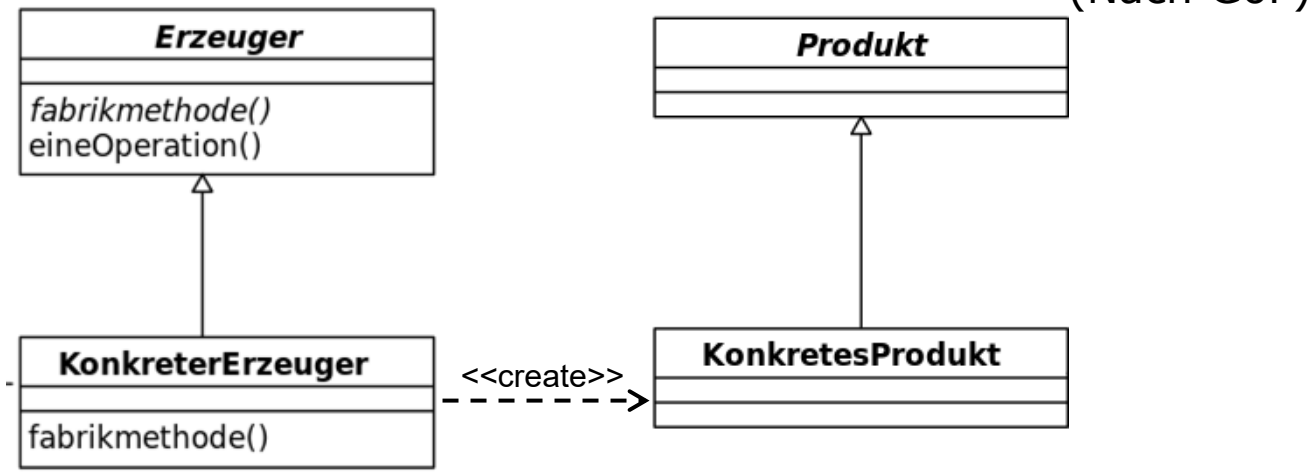
```
public class Main {
   public static void main(String[] args) {
      Spiel goSpiel = new GoSpiel();
      …
…
```

Muster: Factory-Methode nach GoF
Entwurfsmuster der Kategorie Erzeugung

Fabrikmethode

"Definiere eine Schnittstelle zur Erstellung eines Objekts, aber lasse die Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren."

(Nach GoF)



Weitere Anwendungsfelder

- UI (*Erzeuger*: Dialog mit WebDialog *Produkt*: Button mit HTMLButton)
- KfzWerk (*Erzeuger*: KfzWerk mit TeslaWerk *Produkt*: Kfz mit Model5)

Kategorien von Entwurfsmustern

Erzeugung	Verhalten	Struktur
Fabrikmethode	Schablone	Proxy
Abstrakte Fabrik	**Beobachter**	**Kompositum**
Erbauer	Iterator	Adapter
Prototyp	Besucher	**Fassade**
Singleton	**Strategie**	**Dekorierer**
	Zustand Befehl	

Die Facade – Kapselung von Subsystemen

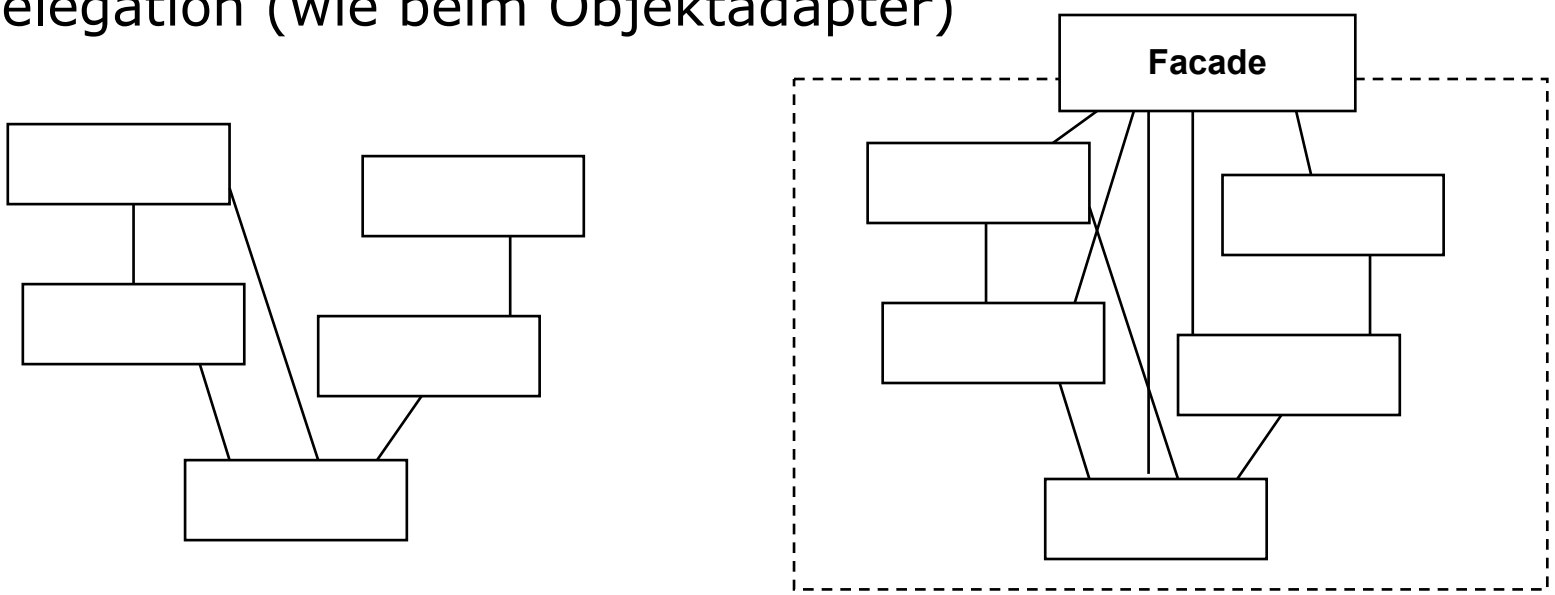
Entwurfsmuster der Kategorie Struktur

Problem:

Ein komplexes Subsystem bietet eine Vielzahl von Schnittstellen an, die vereinheitlicht werden sollen.

Lösung:

Definition einer einheitlichen abstrakten Schnittstelle und Abbildung auf die bestehenden Schnittstellen durch Delegation (wie beim Objektadapter)



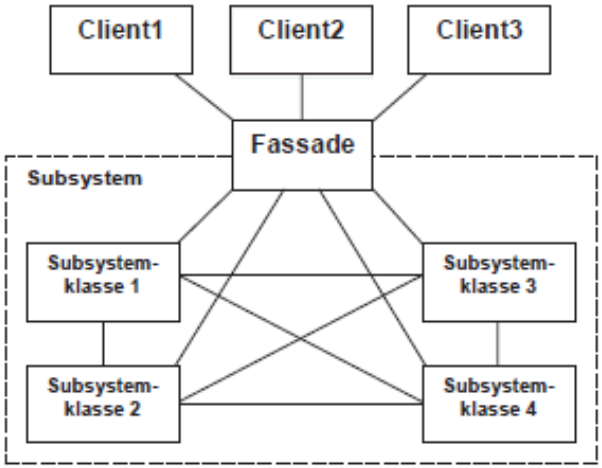
Facade Design Pattern
Entwurfsmuster der Kategorie Struktur

Gang of Four-Definition:

"Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht."

Verwendung:

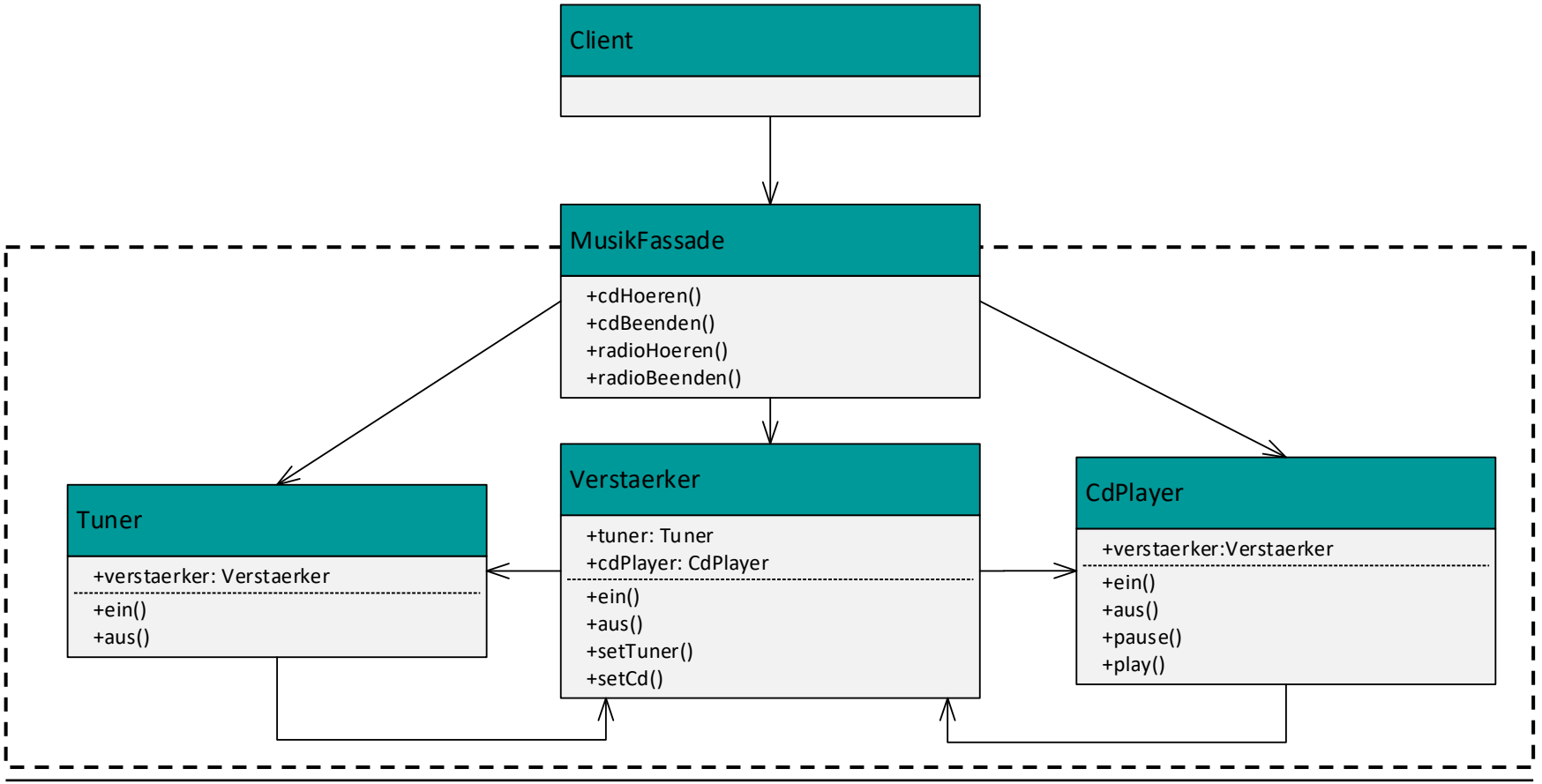
- Zur Verwendung des Systems werden dem Client einfache Schnittstellen angeboten
- Client wird entlastet.
  Er braucht Interna (Klassen und Abhängigkeiten) nicht kennen
- Fassade delegiert die Methodenaufrufe an zuständige interne Klassen/Teil-systeme
- Unterteilung des Systems in einzelne Schichten möglich



Anwendungsbeispiel Facade Design Pattern
Entwurfsmuster der Kategorie Struktur

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Wie kann das in Java umgesetzt werden?



Das Decorator – Pattern
Entwurfsmuster der Kategorie Struktur

Das Problem:

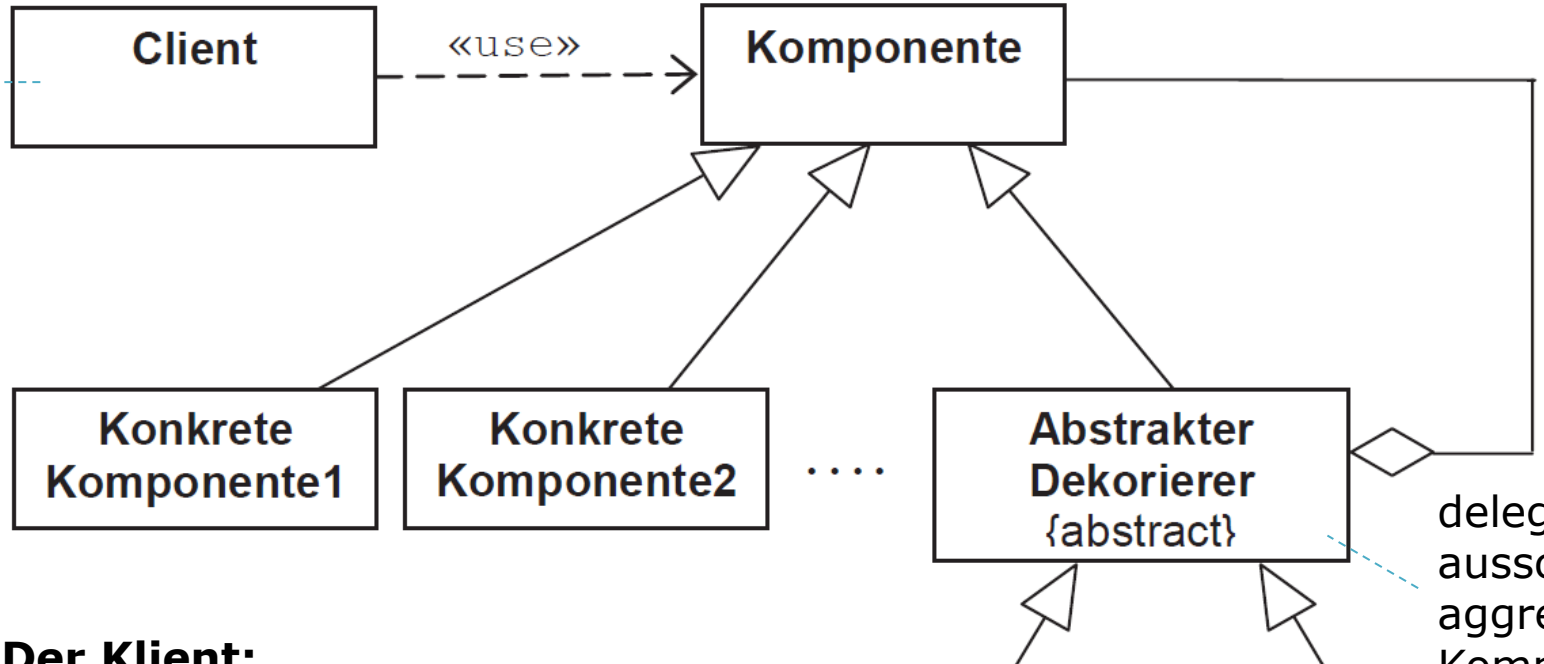
Objekte einer bestehenden Klassenhierarchie sollen dynamisch zur Laufzeit zusätzliche Funktionalität erhalten

Lösungsansatz:

- Gemeinsame Basisklasse der bestehenden Objekte suchen (hier Komponente genannt)
- Ein Dekorator/Dekorierer-Objekt
  - aggregiert jeweils das zu erweiternde Objekt (ugs.: „verziert“)
  - implementiert neue Funktionalität
  - delegiert alles Andere an das bestehende Objekt
- Die Gemeinsame Schnittstelle (Komponente) wird dabei beibehalten

Decorator Entwurfssicht

Entwurfsmuster der Kategorie Struktur



Der Klient:

- ist entkoppelt
- agiert nur auf der gemeinsamen Schnittstelle
- Änderungen in der Implementierung erfordern keine Änderung des Klienten

Decorator Anwendungsbeispiel: Kfz-Konfiguration
Entwurfsmuster der Kategorie Struktur

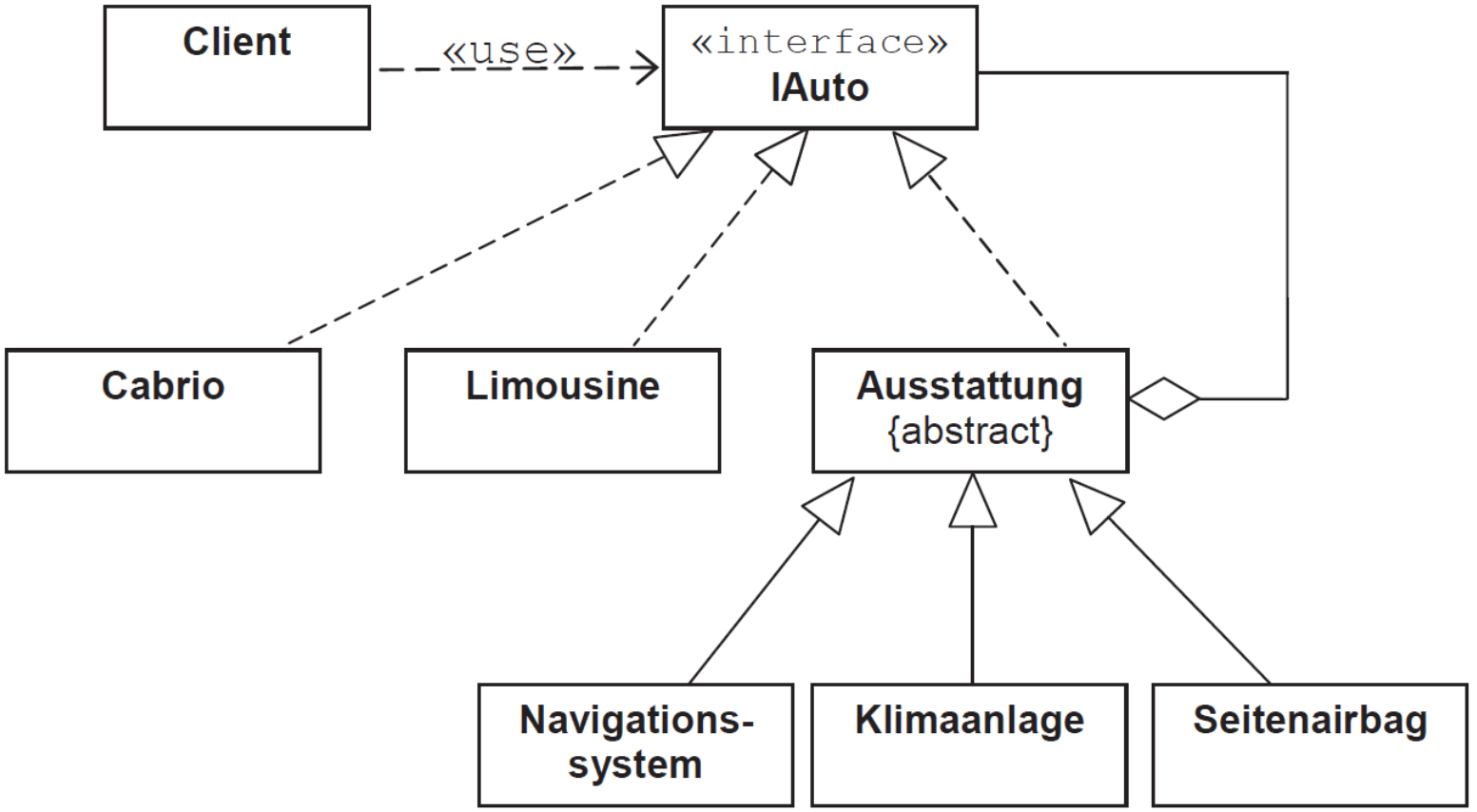


Bild 4-12 Klassendiagramm des Programmbeispiels zum Dekorierer

Beispielimplementierung (1)
Decorator Pattern

```
public interface Auto {

  public void fahren();
}


public class Cabrio implements Auto {

  public void fahren(){
    System.out.println(„Brumm Brumm“);
  }

}

public class Limousine implements Auto {

  public void fahren(){
    System.out.println(„Wroooooooooomm“);
  }

}
```

Beispielimplementierung (2)
Decorator Pattern

```
public abstract class Ausstattung implements Auto {
  protected Auto auto;

  public Ausstattung(Auto auto){
    this.auto = auto;
  }
  public void fahren(){
    auto.fahren();
  }
}

public class Navigationssystem extends Ausstattung {
  public Navigationssystem(Auto auto){
    super(auto);
  }

  @Override
  public void fahren(){
    auto.fahren();
    System.out.println(„ mit Navi“);
  }
}[…]
```

Beispielimplementierung (3)
Decorator Pattern


```
public static void main (String[] args) {
  Auto auto = new Cabrio();
  auto.fahren();

  auto = new Seitenairbag (auto);
  auto.fahren();

  auto = new Navigationssystem(new Klimaanlage(new Cabrio()));
  auto.fahren();
}
```

Ausgabe:

„Brumm Brumm"

„Brumm Brumm mit Seitenairbag"

„Brumm Brumm mit Klimaanlage mit Navi"

Bewertung
Decorator Pattern

Vorteile:

- Dynamisch einzeln erweiterbar: Komponenten **und** Dekorierer
- Redundanzfreie Kombinationsmöglichkeiten von Dekorierern
- Lose Kopplung gegen Client durch gemeinsame Schnittstelle

Nachteile:

- Großer Overhead durch Delegation
- Performance-Einbußen durch Delegation
- Fehlerfindung in kombinierten Dekorierern schwer

Ähnliche Muster:

Kompositum, Proxy, Strategy, Visitor, Pipes&Filter

Kategorien von Entwurfsmustern

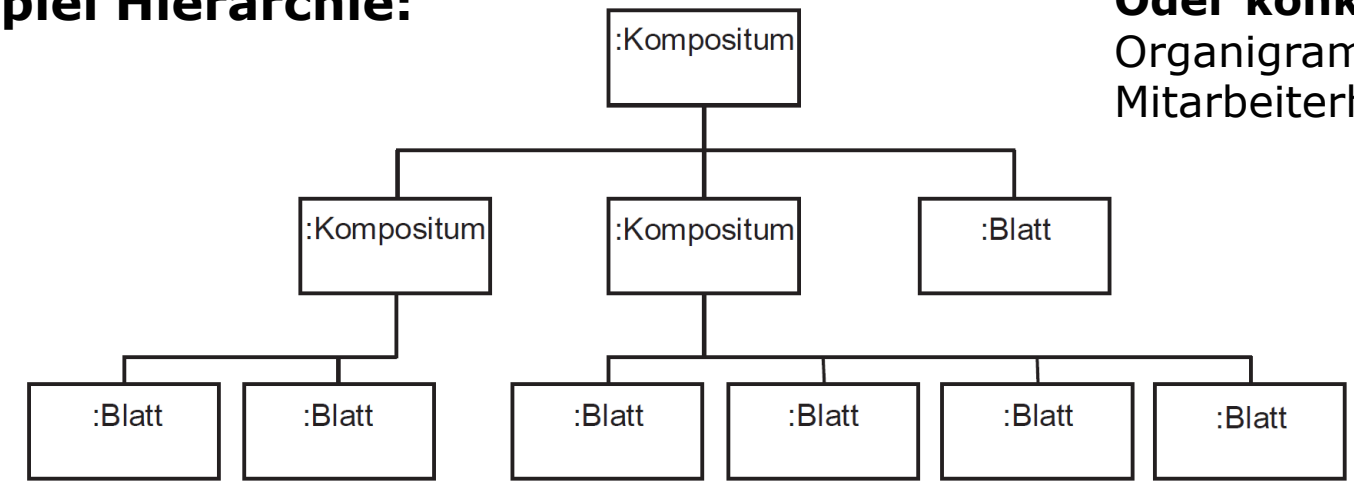
Erzeugung	Verhalten	Struktur
Fabrikmethode	Schablone	Proxy
Abstrakte Fabrik	**Beobachter**	**Kompositum**
Erbauer	Iterator	Adapter
Prototyp	Besucher	**Fassade**
Singleton	**Strategie**	**Dekorierer**
	Zustand Befehl	

Das Composite Pattern
Entwurfsmuster der Kategorie Struktur

Das Problem:

- Abbildung von Objekthierarchien (Teil-Ganzes-Beziehungen)
- Gruppierung in einer baumartigen Struktur
- Einheitliche Behandlung gewünscht, unabhängig ob Blatt oder innerer Knoten

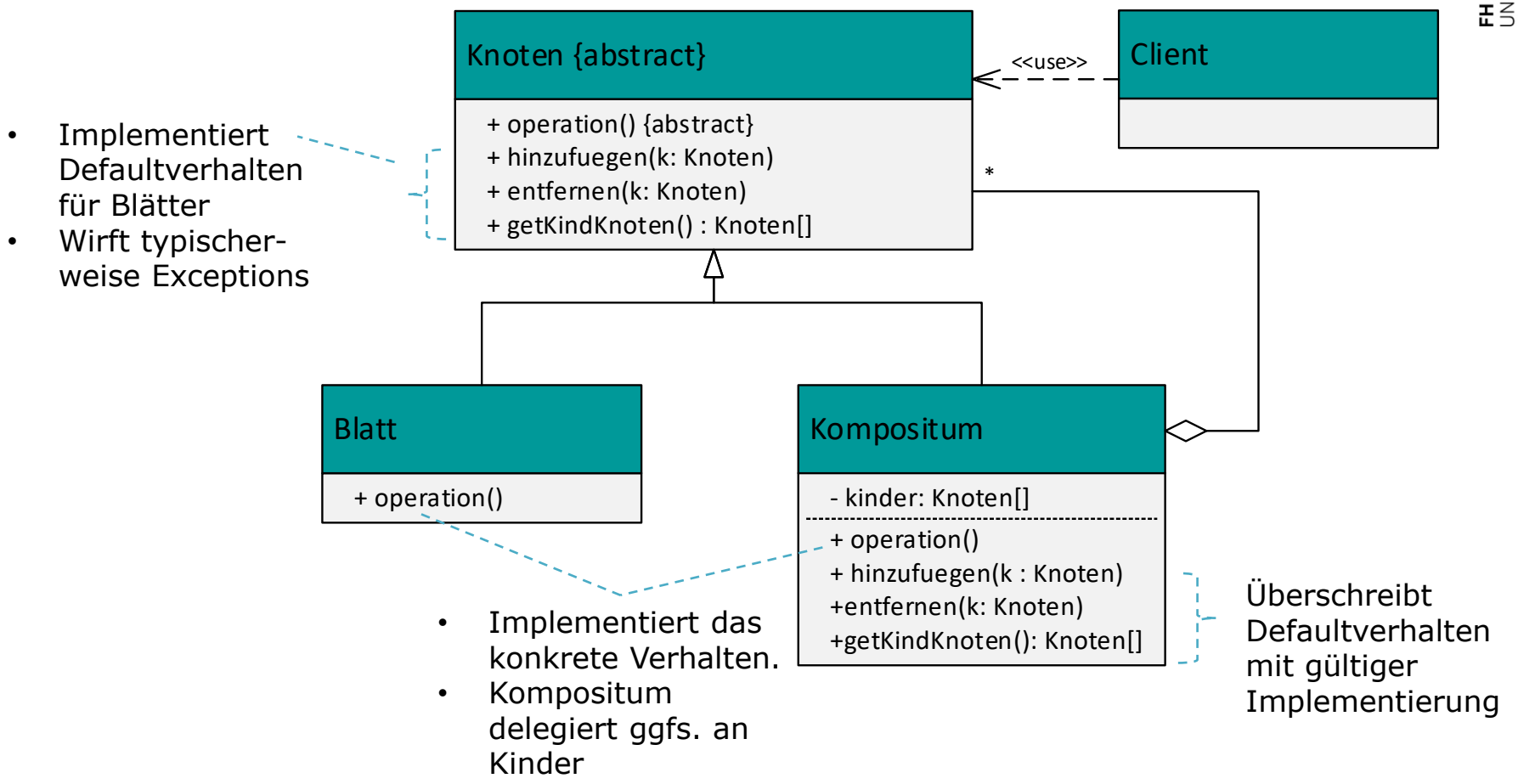
Beispiel Hierarchie:



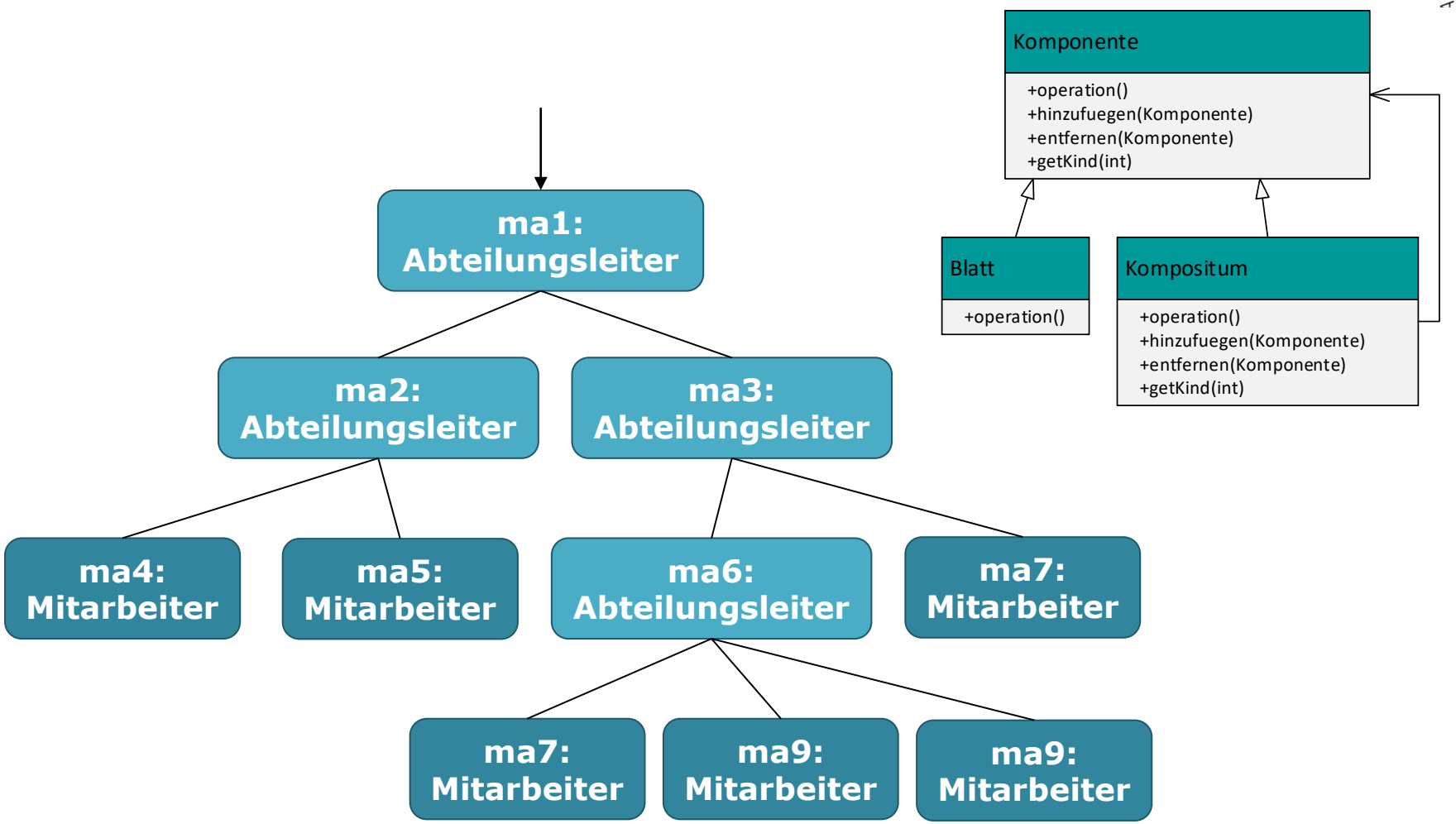
Oder konkret:
Organigramm einer Mitarbeiterhierarchie

Entwurfssicht Composite Pattern
Entwurfsmuster der Kategorie Struktur

Lösungsansatz:



Beispiel zum Composite Pattern
Entwurfsmuster der Kategorie Struktur



Bewertung Composite Pattern
Entwurfsmuster der Kategorie Struktur

Vorteile:

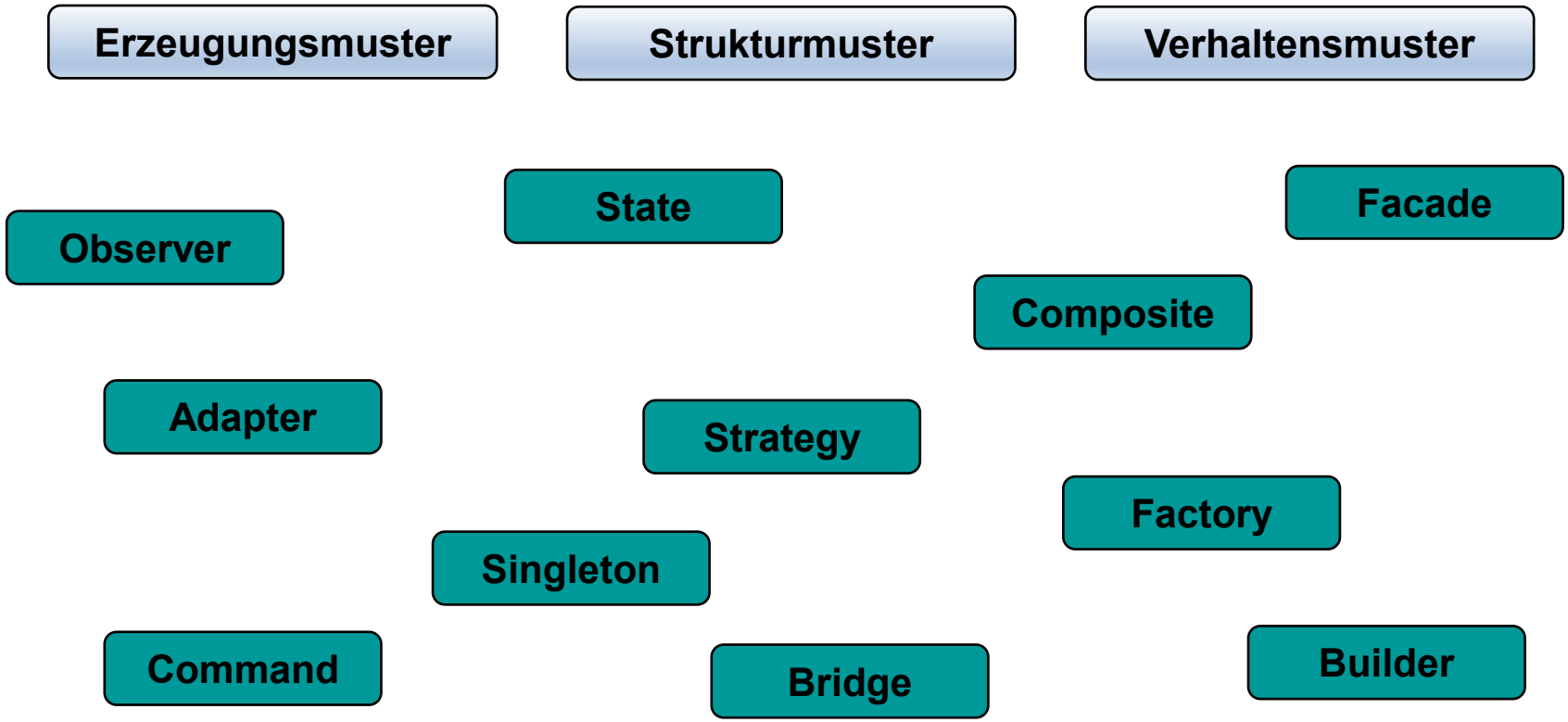
- Client behandelt Blatt und Compositum-Objekte gleich, einfache Verwendung
- Beliebig/mehrfach verschachtelte Strukturen sind möglich
- Erweiterung einfach durch neue Blatt oder Compositum-Klassen

Nachteile:

- Design und Aufbau schnell unübersichtlich, wenn Baumstruktur viele Elemente enthält (→ in der Verwendung)
- Ungeeignet, wenn nicht alle Compositum-Objekte gleich behandelbar (bsp. Zusätzliche Einschränkungen)
- Änderungen an Basisschnittstelle aufwendig umzusetzen

Murmelgruppe
Entwurfsmuster

In welche Kategorie gehören die genannten Entwurfsmuster?



Architekturmuster – Entwurfsmuster – Idiome
Kategorien von Mustern

Muster bei der Software-Architektur
- Wie wird das System in Subsysteme/Module unterteilt?

Muster im Software-Design
- Welche Klassen/Objekte werden erstellt?
- Welche Schnittstellen werden definiert?
- Wie kommunizieren die Objekte?

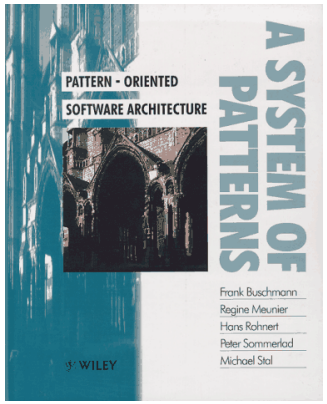
Muster bei der Implementierung
- Programmiersprachen-spezifisch
- Namenskonventionen für Klassen, Attribute, Methoden
- Formatierung von Quellcode zur besseren Les- und Wartbarkeit

Ziele von Architekturmustern
Architekturmuster

Das Hauptziel: Hilfsmittel für den Entwurf im Großen

- Wie kann man effiziente und flexible Architekturen unabhängig von einer bestimmten technischen Basis entwerfen?
- Wie findet man Komponenten und Subsysteme für große Systeme?
- Wie kann man die Verantwortlichkeiten und Schnittstellen der Komponenten bestimmen?
- Wie geht man bei der Umsetzung der Architektur vor?

Buch "A System of Patterns" von Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal („das Siemens-Buch"), 1995



Anwendungsbeispiele
Architekturmuster

Struktur
- EVA, Hollywood
- ISO/OSI-Schichtenmodell
- Pipes & Filters
- Komponentenbasierte Arch.
- Objektorientierte Arch.

Interaktion
- Model-View-Controller
- Model-View Presenter
- Model-View-ViewModel

Verteilung
- Client/Server Architektur
- N-Tier
- Serviceorientierte Architektur

Adaptiv
- Plug-In Architektur für Frameworks

➢ Beispiel Webbrowser: Wiedergabe von Multimedia-Inhalten

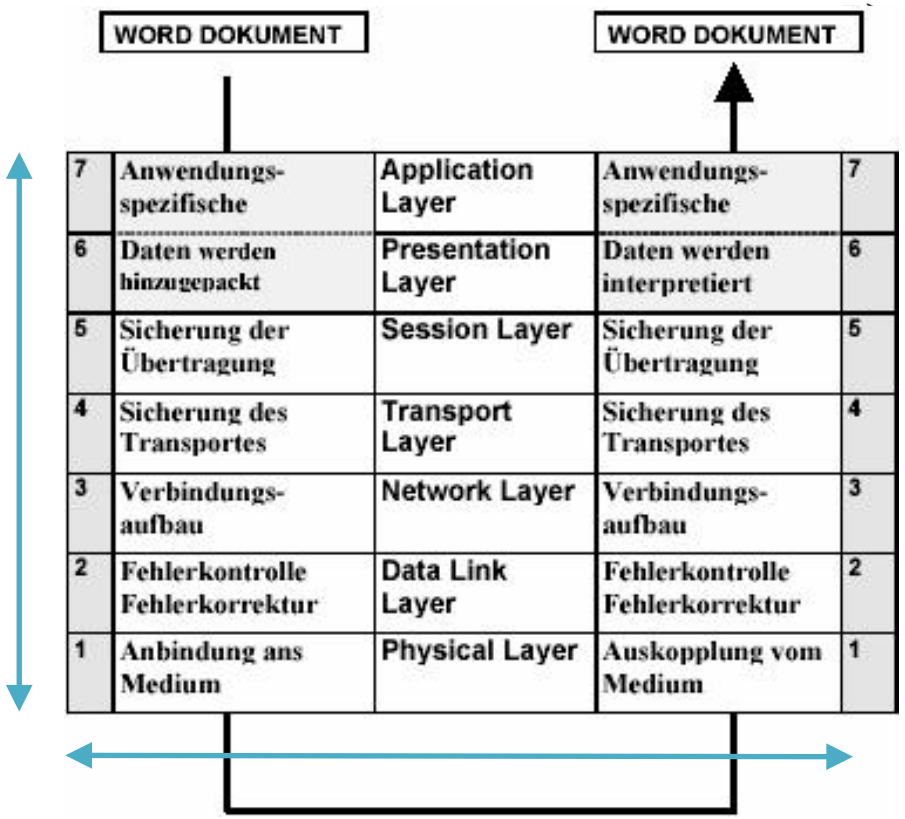
Layers
Architekturmuster - Struktur

Name:

- Layers oder Schichtenmuster

Zweck:

- Strukturiere Anwendungen, deren Funktionalität in Gruppen von Unterfunktionen zerfällt
- Einzelnen Gruppen werden unterschiedliche Abstraktionsniveaus zugeordnet

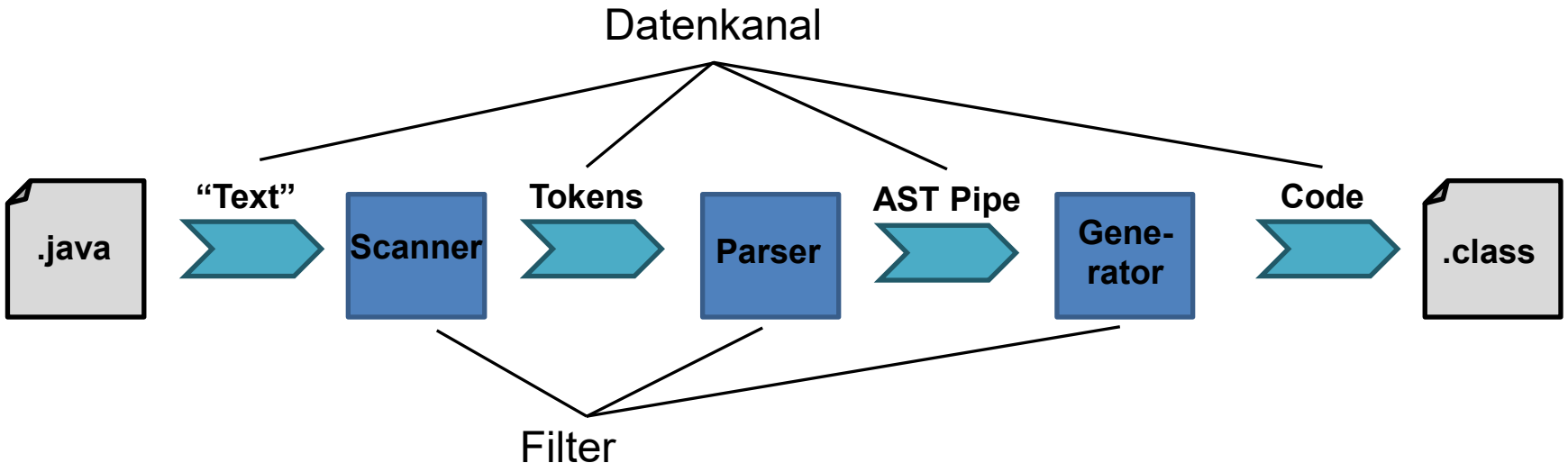


Beispiel:

- Netzwerkprotokolle wie ISO/OSI oder TCP/IP

Pipes and Filters
Architekturmuster - Struktur

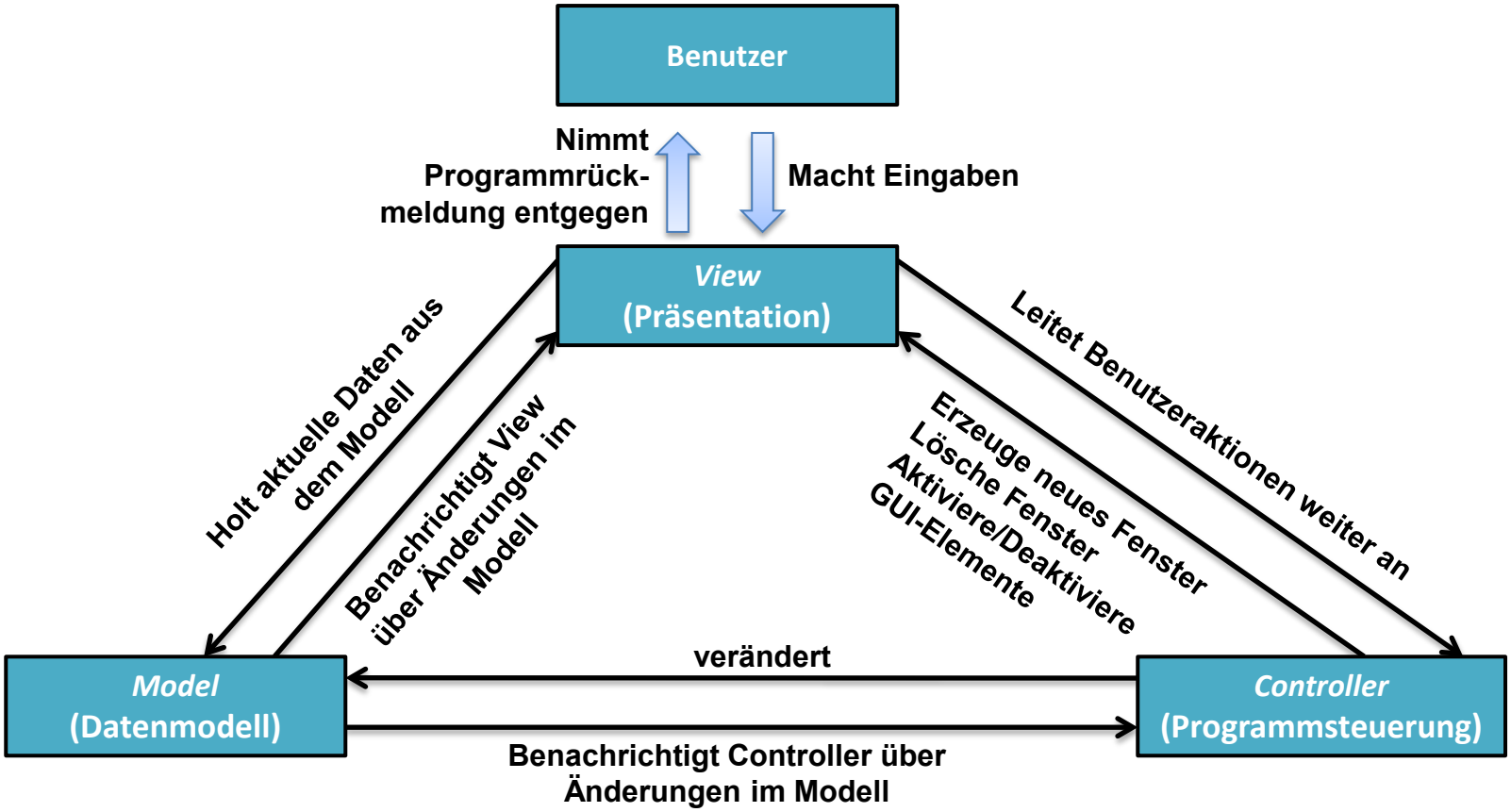
- Verarbeitungseinheiten (Filter) und Datenkanäle (Pipes)
- Filter reichen Daten an andere Filter weiter
- Pipes werden nicht als eigene Komponenten realisiert



Weiteres einfaches Beispiel: Pipes unter UNIX

➢ `ls | grep | sort`

Model-View-Controller
Architekturmuster - Interaktion



Model-View-Controller
Architekturmuster - Interaktion

Modell:

- Kapselt fachliche Objekte
- Kennt die Views und Controller, die sich angemeldet haben
- Benachrichtigt angemeldete Komponenten über Zustandsänderungen
- Unabhängig von konkreten Views und Controllern

Model-View-Controller
Architekturmuster - Interaktion

View:

- Darstellung des Modells für den Benutzer
- Kennt Modell und seinen Controller
- Keine Weiterverarbeitung der übergebenen Daten
- Realisiert einen Update-Callback-Mechanismus (Observer)

Model-View-Controller
Architekturmuster - Interaktion

Controller:

- Verarbeitet Ereignisse aus der View (od. vom Benutzer)
- Übersetzt Ereignisse in Anfragen/Befehle an Modell oder View
- Realisiert einen Update-Callback-Mechanismus
- Kann unterschiedliche Views bedienen.

Model-View-Controller
Architekturmuster - Interaktion

Vorteile:

- Erlaubt mehrere Sichten auf ein Modell
- Automatische Synchronisation der Sichten
- Austauschbarkeit von Darstellung und Kontrollablauf

Nachteile:

- Relative enge Kopplung zwischen View und Controller
- Hohe Anzahl an Aktualisierungen möglicherweise ein Performance-Problem

Client/Server
Architekturmuster - Verteilte Systeme

- Teilt System in zwei Applikationen
  - Client sendet Anfragen an den Server
- Alle Daten auf dem Server gespeichert
  - Höhere Sicherheit
  - Zugriffe/Updates einfacher zu administrieren
- Server kann ausgetauscht werden, ohne das Client dies merkt

Request – Response - Pattern