Seminar 2

Themen aus dem Vorlesung:

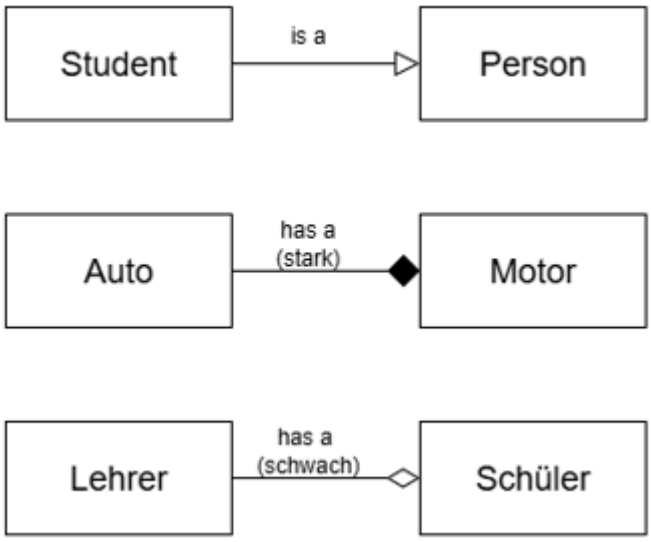
- Klassen, Objekte,
- Beziehungen zwischen Objekten
  - Vererbung
  - Assoziation: Aggregation, Komposition
- Überschreiben (overwriting), Überladen (overloading)
- Polymorphismus
- Subtyping
- Parameterübergabe
- Statische Mitglieder, statische Klassen
- Schnittstellen
- Abstrakte Klassen
- Pakete

Lernziele

- Verstehen, wie Klassen miteinander in Beziehung stehen (Vererbung, Komposition, Aggregation)
- Den Unterschied zwischen Interface und abstrakter Klasse erkennen
- Wissen, wann man was einsetzt (is-a / has-a / can-do)

Themenüberblick

1. Wiederholung: Warum Beziehungen wichtig sind
   - Beziehungen helfen, Realität zu modellieren
     - Student ist eine Person → Vererbung
     - Auto hat einen Motor → Komposition
     - Lehrer hat Schüler → Aggregation



2. Vererbung (is-a)
    - Gemeinsamkeiten und Spezialisierungen
    - Code-Wiederverwendung
    - Polymorphismus durch Überschreiben

```
class Device {
    void start();
}

class Smartphone extends Device {
    @Override
    void start() { System.out.println("Smartphone booting..."); }
}

class Laptop extends Device {
    @Override
    void start() { System.out.println("Laptop booting..."); }
}
```

3. Komposition (has-a, **stark**)
    - Ein Objekt besitzt ein anderes → Lebenszyklus abhängig

```
class Engine {}
class Car {
    private Engine engine = new Engine();
}
```

4. Aggreagtion (has-a, **schwach**)
    - Objekte existieren unabhängig voneinander

```
class Student {}
class Teacher {
    private List<Student> students;
}
```

5. interfaces vs abstrakte Klasses

Interface “what something can do”

- Purpose:
    - Describes what a class can do (a contract of behavior)
    - Use it When classes share behavior but are otherwise unrelated

- Methods:
  - All methods are abstract by default
  - default methods -> can provide default implementation
- Fields:
  - Only **constants** (public static final by default).
- Inheritance:
  - A class can implement multiple interfaces
- Constructore:
  - None
- Example:

```
interface Flyable {
    void fly();
}

class Bird implements Flyable {
    public void fly() { System.out.println("Bird flying."); }
}

class Airplane implements Flyable {
    public void fly() { System.out.println("Airplane flying."); }
}
```

Abstract classes “what something is”

- Purpose:
  - Can define both **a partial implementation** and **a contract**.
  - Describes what a class is (a partial implementation)
  - When classes share structure and logic
- Methods:
  - Can have both abstract and concrete methods
  - default methods -> Can provide concrete implementations normally
- Fields:
  - Can have instance variables, constants, and static variables
- Inheritance:
  - A class can extend only one abstract class (single inheritance)
- Constructore:
  - Can have a constructor (used when subclassing)
- Example:

```
abstract class Animal {
    String name;

    public Animal(String name) { this.name = name; }

    abstract void makeSound();  // abstract method

    void eat() { System.out.println(name + " is eating."); } //
concrete method
}

class Dog extends Animal {
    public Dog(String name) { super(name); }
    @Override void makeSound() { System.out.println("Woof!"); }
}
```

=> Use interface for capabilities and multiple inheritance.

=> Use abstract class when you have shared code and want to define a base class.

Example with both abstract and interface:

```
interface Payable {
    void pay();
}

abstract class Employee implements Payable {
    String name;
    public Employee(String name) { this.name = name; }
    public void showInfo() { System.out.println(name); }
}

class Manager extends Employee {
    public Manager(String name) { super(name); }
    @Override public void pay() { System.out.println("Manager salary paid."); }
}
```

Praktischer Teil

Übung 1 - Tierwelt (Vererbung & Polymorphismus)

Erstelle folgende Klassenhierarchie:

```
abstract class Animal {
    protected String name;
    public Animal(String name) { this.name = name; }
    public abstract void makeSound();
}
```

```
class Dog extends Animal {
    public Dog(String name) { super(name); }
    public void makeSound() { System.out.println(name + " bellt."); }
}

class Cat extends Animal {
    public Cat(String name) { super(name); }
    public void makeSound() { System.out.println(name + " miaut."); }
}
```

Aufgaben:

1. Erzeuge eine ArrayList mit Hunden und Katzen.
2. Iteriere und rufe makeSound() auf → Polymorphismus.
3. Füge eine neue Klasse Bird hinzu, ohne bestehende Klassen zu ändern.

Übung 2 - Auto und Motor (Komposition)

```
class Engine {
    public void start() { System.out.println("Motor gestartet."); }
}

class Car {
    private Engine engine;
    public Car() { this.engine = new Engine(); }
    public void drive() { engine.start(); System.out.println("Auto fährt."); }
}
```

Aufgaben:

1. Erstelle ein Car-Objekt und rufe drive() auf.
2. Erkläre, warum das Auto den Motor „besitzt“.
3. Was passiert, wenn das Auto gelöscht wird?

Übung 3 - Auto und Motor (Komposition)

```
class Student {
    private String name;
    public Student(String name) { this.name = name; }
    public String getName() { return name; }
}

class Teacher {
    private List<Student> students;
    public Teacher(List<Student> students) { this.students = students; }
    public void printStudents() {
        for (Student s : students)
            System.out.println("Schüler: " + s.getName());
```

```
    }
}
```

Aufgaben:

1. Erzeuge einige Student-Objekte und einen Teacher, der sie bekommt.
2. Rufe printStudents() auf.
3. Was passiert, wenn du den Lehrer löschst, aber die Liste behältst?

Übung 4 — Interface und Abstraktion verbinden

```
interface Driveable {
    void drive();
}

abstract class Vehicle implements Driveable {
    protected String name;
    public Vehicle(String name) { this.name = name; }
}

class Truck extends Vehicle {
    public Truck(String name) { super(name); }
    public void drive() { System.out.println(name + " transportiert Ware."); }
}
```

Aufgaben:

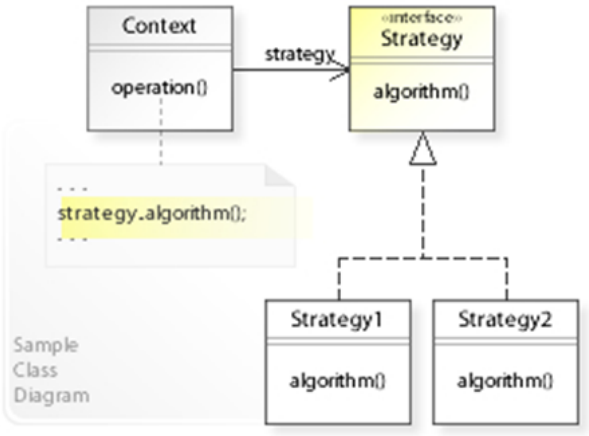
1. Erstelle ein List und füge Truck-Objekte hinzu.
2. Füge später eine neue Klasse Bicycle hinzu (ebenfalls Driveable).
3. Rufe drive() für alle Elemente auf.

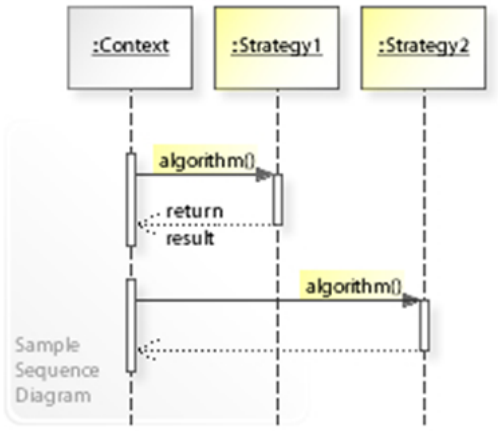
Übung 5 - Kiste mit Sachen

In einer Kiste befinden sich Sägen, Hämmer, Nägel und Scheren. Implementiere eine Klasse Filter mit 2 Methoden:

1. eine Methode, die alle Entitäten ausgibt, deren Gewicht größer als ein Kilogramm ist.
2. eine Methode, die alle Entitäten ausgibt, die schneiden können.

Übung 6 - Strategy Pattern





Die Klasse Strategie definiert nur eine Schnittstelle für alle unterstützten Algorithmen. Die Implementierung der eigentlichen Algorithmen findet sich erst in den Erweiterungen wieder. Der Kontext hält eine Variable der Schnittstelle Strategie, die mit einer Referenz auf das gewünschte Strategieobjekt belegt ist. Auf diese Weise wird der konkrete Algorithmus über die Schnittstelle eingebunden und kann bei Bedarf selbst zur Laufzeit noch dynamisch gegen eine andere Implementierung ausgetauscht werden.

Also:

Der Strategy Pattern hilft dabei, **ein Verhalten** (Algorithmus) **flexibel zu ändern**, ohne bestehende Klassen anzupassen. Man trennt also das „Was" vom „Wie":

- Was soll gemacht werden? → definiert die Schnittstelle (Interface / abstrakte Klasse)
- Wie wird es gemacht? → konkrete Implementierungen der Strategien

Analogie:

Stell dir vor, du hast eine Navigations-App. Die Strategie bestimmt, wie der beste Weg berechnet wird. Es gibt mehrere Strategien: „schnellster Weg", „kürzester Weg", „landschaftlich schönster Weg". Die App (der Kontext) fragt einfach die Strategie, wie der Weg berechnet wird. Du kannst jederzeit eine andere Strategie auswählen, ohne die App selbst umzubauen.

Beispiel:

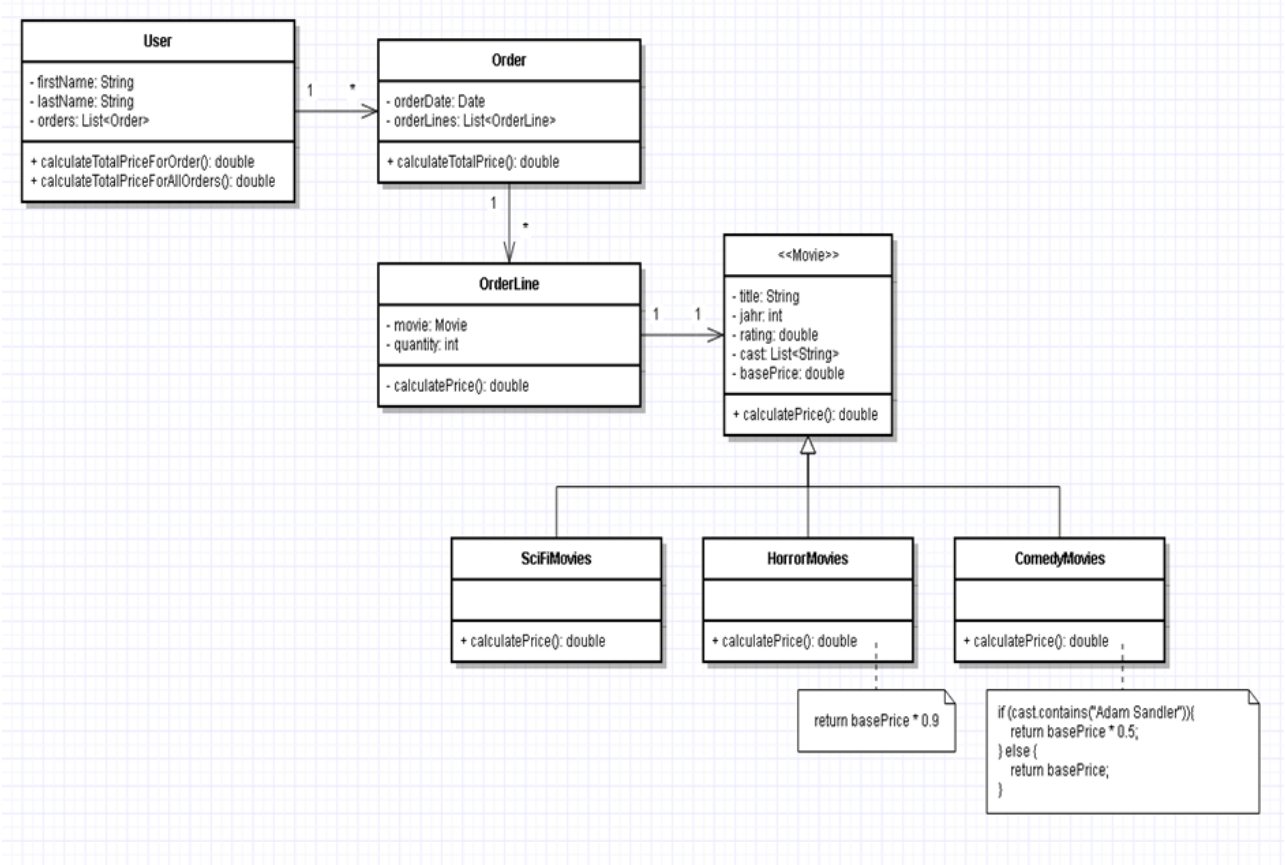
```
// Strategy Interface
interface PriceStrategy {
    double calculate(double basePrice);
}

// Concrete Strategies
class NormalPrice implements PriceStrategy {
    public double calculate(double basePrice) { return basePrice; }
}

class Discount10 implements PriceStrategy {
    public double calculate(double basePrice) { return basePrice * 0.9; }
}
```

```
// Context
abstract class Movie {
    protected double basePrice;
    protected PriceStrategy strategy;

    public Movie(double basePrice, PriceStrategy strategy) {
        this.basePrice = basePrice;
        this.strategy = strategy;
    }

    public double getPrice() {
        return strategy.calculate(basePrice);
    }
}
```

Übung 7 - Watching a movie at home

Ein Benutzer möchte eine kleine Anwendung haben, wo er mehrere Filme bestellen kann. Eine Bestellung kann mehrere Filme beinhalten, bzw. ein Film kann in einer Bestellung mehrmals erscheinen.

- Jeder Film hat einen Titel, ein Erscheinungsjahr, eine Auswertung, eine Liste mit Schauspielern und einen Preis.
- Es gibt drei Arten von Filmen: Sci-Fi-Filme, Horror-Filme und Komödien.
- Der Preis eines Films hängt von seiner Art ab.
    - Sci-Fi-Filme werden mit dem normalen Preis kalkuliert.
    - Horror-Filme haben 10% Rabatt.
    - Komödien, in denen Adam Sandler spielt, haben 50% Rabatt.
- Der Benutzer hat die Möglichkeit den endgültigen Preis einer Bestellung bzw. den Preis von allen bisherigen Bestellungen zu kalkulieren.



Aufgaben:

1. Implementiere mit der Anwendung von Strategy die folgenden Klassen: Movie, SciFiMovies, HorrorMovies, ComedyMovie. Movie wird als eine abstrakte Klasse implementiert.
2. Implementiere die Klasse OrderLine mit der Methode calculatePrice().

Abschlussdiskussion

Leitfragen:

- Wann würdest du Vererbung vermeiden und stattdessen Komposition nutzen?
- Warum ist „Mehrfachvererbung“ über Interfaces erlaubt, aber nicht über Klassen?
- Wie helfen Interfaces und Abstraktion bei der Wartbarkeit?