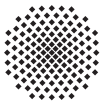

Universität Stuttgart

SQA
Software Quality and Architecture Group

Übungsblatt 08

Programmierung und Softwareentwicklung (WS 2024/2025)

Abgabe: Fr. 13.12.2024, 23:59 Uhr — Besprechung: KW 50

- Bitte lösen Sie die Übungsaufgabe in **Gruppen von 2 Studierenden**.
- Dieses Übungsblatt besteht aus zwei Teilen (A, B). Teil A ist in der Präsenzübung zu lösen. Teil B ist in Heimarbeit (Gruppe von 2 Studierenden) zu lösen und rechtzeitig abzugeben. Die Abgabe erfolgt über ILIAS.
- Geben Sie `.java`-Dateien nur im UTF-8 Encoding ab. Ändern Sie das Textdateiencoding auf UTF-8 ab, bevor Sie die Unterlagen herunterladen. Abhängig von Ihrem Betriebssystem müssen Sie möglicherweise auch nichts tun.
- Geben Sie zu Beginn der Dateien Ihre Namen (Vor- und Nachname), die Matrikelnummern und die E-Mail-Adressen an. Nutzen Sie bei Java-Dateien die korrekte JavaDoc-Syntax.
- Benennen Sie die Dateien nach dem folgenden Schema:
  1. **PSE**[**ÜB-Nr**]-[**Nachnamen der Teammitglieder**]-[**Nachname des Tutors**]**.pdf**
     Beispiel: PSE08-StießSpethKrieger-Becker.pdf
  2. [**Klassenname**]**.java**: Alle von Ihnen bearbeiteten Java-Dateien, die Lösungen für die Aufgaben enthalten.
- Missachtung der formalen Kriterien kann dazu führen, dass einzelne Aufgaben oder die gesamte Abgabe mit 0 Punkten bewertet werden.

Lernziel: Auf diesem Übungsblatt werden Sie Ihr Wissen über die Konzepte der Objektorientierten Programmierung vertiefen. Des Weiteren werden Sie den Umgang mit sämtlichen Konzepten der vergangenen Wochen weiter üben, insbesondere den Umgang mit defensiver und offensiver Programmierung und eigenen Klassen und Operationen.

Punkte: Dieses Übungsblatt enthält zwei Teile. In Teil B können Sie bis zu 50 Punkte und 5 Bonuspunkte erzielen. Zum Bestehen des Blatts benötigen Sie mindestens 25.0 Punkte.

Style: Bitte halten Sie die in der Vorlesung vorgestellten Style-Regeln ein. Dazu gehören auch JavaDoc sowie Vor- und Nachbedingungen. Der Style Ihrer Implementierung wird mit bis zu 50% bewertet.

Vorbereitung: Bitte erledigen Sie die folgenden Schritte **vor** der Präsenzübung.

- Importieren Sie das zu diesem Übungsblatt gehörende Maven Projekt in Ihre IDE. Sie finden das Maven Projekt in unserem git-Repository: `https://github.com/SQA-PSE-WS-2024-2025/exercise-sheet-08`
- Stellen Sie sicher, dass Sie Übungsblatt 07 absolviert haben, sowie alle Software installiert und funktionsfähig ist (IDE (Eclipse, IntelliJ, VSCode,...) und Java 21).

Unterlagen:

- Git-Repositories: `https://github.com/SQA-PSE-WS-2024-2025/`
- Dokumentation des Hamstersimulators: `https://tinyurl.com/5yx654w8`

Scheinkriterien: Durch die Teilnahme am Übungsbetrieb können Sie sich für die Teilnahme an der Klausur qualifizieren:

- Bestehen von min. 80% aller Übungsblätter.
- Ein Übungsblatt gilt als bestanden, wenn 50% der Punkte des abgegebenen Heimarbeitsteils erreicht wurden.
- Aktive Teilnahme an min. 80% der Übungen.

Viel Erfolg!

1 Teil A - Präsenzaufgaben

Aufgabe 1 Begriffe, Begriffe, Begriffe.

In dieser Aufgabe festigen Sie Ihren Umgang mit einigen Fachbegriffen der Objektorientierten Programmierung.

(a) Beschreiben Sie in eigenen Worten die Begriffe *Polymorphie*, *polymorphe Bindung*, *statischer Typ*, *dynamischer Typ* und *dynamisches Binden*. Schreiben Sie Ihre Ergebnisse auf.

Aufgabe 2 Vererbung und Polymorphie

Diese Aufgabe befasst sich mit den Grundlagen der Vererbung und Polymorphie.

(a) Machen Sie sich mit den gegebenen Klassen im `presence.transportation`-Paket und deren Hierarchie vertraut. Zeichnen Sie ein Diagramm der Vererbungshierarchie der Klassen.

Hinweis: Die Struktur bzw. das Aussehen des Diagramms ist Ihnen hier frei überlassen. Ignorieren Sie die Klassen und Enums im Paket `presence.transportation.energyefficiency`.

(b) In der Hierarchie finden Sie das Interface `Transportation` und die abstrakte Klasse `PublicTransportation`. Diskutieren Sie anhand dieser Klassen die Gemeinsamkeiten und Unterschiede von Interfaces und abstrakten Klassen. Schreiben Sie Ihre Ergebnisse auf.

(c) In den gegebenen Klassen gibt es mehrere Stellen, an denen Abstraktionen, wie weitere Vererbungen, Extraktion von Operationen in die Elternklassen und ähnliches, sinnvoll wären, aber nicht umgesetzt wurden. Beispielsweise enthält jede Klasse eine Operation `getSpeed()` welche in `Transportation` deklariert werden könnte.

Finden Sie mindestens eine weitere solche Stelle und ändern Sie den entsprechenden Code sowie das Diagramm aus Teil (a), sodass diese Abstraktion genutzt wird.

(d) Bestimmen Sie für die in Listing 1 abgebildeten Codezeilen, ob diese vom Compiler korrekt übersetzt werden können. Geben Sie für jeden gefundenen Fehler einen Grund an.

Hinweis: Hier kann es hilfreich sein, das Diagramm aus Teilaufgabe (a) zu Rate zu ziehen.

```
1  Plane plane = new Plane();
2  Train trainA = new Train("train", EnergyType.ELECTRIC);
3  Hyperloop hyperloop = new Hyperloop();
4  Ferry ferry = new PublicTransportation("Ferry");
5  Transportation transportation;
6
7  PublicTransportation publicTransportation;
8  Train trainB;
9
10 publicTransportation = plane;
11 transportation = new Transportation();
12 transportation = plane;
13 plane = publicTransportation;
14 trainB = plane;
15 trainB = hyperloop;
16 publicTransportation = trainB;
17 publicTransportation = new Plane();
18 publicTransportation.getEfficiencyCategory(true);
19 trainB.getEfficiencyCategory();
```

Listing 1: Fehlerhafter Code

Aufgabe 3 Bindung, Typen, Überladen und Überschreiben

In dieser Aufgabe geht es um dynamische Bindung, statische und dynamische Typen sowie Überladen und Überschreiben. Diese Aufgabe bezieht sich auf die bereits in Aufgabe 2 verwendete Klassenhierarchie.

(a) Betrachten Sie den Code in Listing 2. Bestimmen Sie für die Variablen `transportation` und `train` jeweils den statischen Typ und alle dynamischen Typen, die sie während der Laufzeit des Codes haben könnten.

1. Welche dieser dynamischen Typen können die Variablen in den Zeilen 27-28 haben?
2. Welchen statischen Typen haben die Variablen in den Zeilen 27-28?

Gehen Sie davon aus, dass `passengerIsRich` und `fearOfFlying` zwei durch Benutzerabfrage initialisierte Variablen vom Typ `boolean` sind.

```java
Plane plane = new Plane();
Hyperloop hyperloop = new Hyperloop();
Ferry ferry = new Ferry();
SteamTrain steamTrain = new SteamTrain();
Yacht yacht = new Yacht();

Transportation transportation;
Train train;

transportation = steamTrain;
train = steamTrain;

if (passengerIsRich) {
    train = hyperloop;
} else {
    train = steamTrain;
}

if (!fearOfFlying) {
    transportation = plane;
} else if (passengerIsRich) {
    transportation = yacht;
} else {
    transportation = ferry;
}

train.getEfficiencyCategory(true);
transportation.getEfficiencyCategory();
```

Listing 2: Code mit statischen und dynamischen Typen

(b) Erklären Sie den Unterschied zwischen *Überladen* und *Überschreiben* und finden Sie je ein Beispiel in den Klassen des `presence.transportation`-Pakets.

(c) Was wird in den Zeilen 6-9 aus Listing 3 ausgegeben? Begründen Sie Ihre Antwort unter Verwendung der in Aufgabe 1 und Teilaufgabe (b) beschriebenen Fachbegriffe.

```java
Ferry ferry = new Ferry();
PublicTransportation transportation = new SteamTrain();
Plane plane = new Plane();
Hyperloop hyperloop = new Hyperloop();

ferry.printTransportationInfo();
transportation.printTransportationInfo();
plane.printTransportationInfo();
hyperloop.printTransportationInfo();
```

Listing 3: Code mit Ausgabe

Aufgabe 4 Erweiterung der Funktionalität der Hamster-Klasse

Die `Hamster`-Klasse aus dem Hamstersimulator bietet die Funktionalität an, dass sich ein Hamster nach links, jedoch nicht nach rechts, drehen kann. Auf dem zweiten Übungsblatt sollten Sie deshalb in der Klasse `ExerciseHamsterGame` die Operation `turnRight` implementieren, die es Paule ermöglicht, sich nach rechts zu drehen.

(a) Machen Sie sich hier (noch einmal) bewusst, warum diese Operation in `ExerciseHamsterGame` nicht an der richtigen Stelle implementiert ist. Geben Sie zwei Gründe dafür an.

(b) Den Hamstersimulator müssen Sie als gegeben annehmen, d.h. Sie können die Klasse `Hamster` nicht verändern. Wie können Sie dennoch einen Hamster realisieren, der zusätzlich zur Funktionalität der `Hamster`-Klasse weitere sinnvolle Operationen anbietet (wie z.B. `turnRight`)?

(c) Betrachten Sie die Klasse `MyGreatHamster` im `presence.hamster`-Paket und wenden Sie Ihre Idee aus Teilaufgabe (b) an. Wie unterscheiden sich die Attribute der Klasse `Hamster` und der Klasse `MyGreatHamster`? Implementieren Sie den gegebenen Konstruktor für die Klasse `MyGreatHamster`, sodass dieser alle Attribute dieser Klasse initialisiert. Wenn Sie nun den Code ausführen, sollte der neue `MyGreatHamster greatPaule` (erzeugt in der Operation `run()` der Klasse `MyGreatHamsterGame`) in der Kachel unter dem Default-Hamster Paule erscheinen.

Hinweis: Überlegen Sie sich, wie Sie hierfür den Konstruktor der `Hamster`-Klasse nutzen können.

(d) Ermöglichen Sie Objekten der Klasse `MyGreatHamster`, sich nach rechts zu drehen, indem Sie die Operation `turnRight()` implementieren.

(e) Ergänzen Sie darüber hinaus die Operationen `multiMove(final int stepCounter)`, durch die der Hamster `stepCounter` Schritte machen soll, und `pickAllGrains()`, durch die der Hamster alle auf dem aktuellen Feld verfügbaren Körner aufsammelt.

2 Teil B - Heimarbeit

Aufgabe 1 Mutable und Immutable (PDF, `.java`)

In dieser Aufgabe geht es um die Veränderlichkeit von Objekten.

(a) *(4 Punkte)* Betrachten Sie die Klasse in Listing 4. Ist die Klasse semantisch korrekt und sind die Objekte der Klasse unveränderlich? Falls nein, wie kann man sie korrigieren? Begründen Sie Ihre Antwort.

```
public final class Vehicle {

    private final int numberOfWheels;
    private final int numberOfSeats;

    public Vehicle(final int numberOfWheels) {
        this.numberOfWheels = numberOfWheels;
    }

    public int getNumberOfSeats() {
        return numberOfSeats;
    }

    public int getNumberOfWheels() {
        return numberOfWheels;
    }

}
```

Listing 4: Unveränderliche(?) Klasse

(b) Betrachten Sie die Klasse `Person` im `homework.immutable`-Paket. Die Objekte der Klasse sind veränderlich.

i. *(6 Punkte)* Nennen Sie alle Stellen der Klasse (mit Zeilennummern), welche die Instanzen von `Person` veränderlich machen. Erklären Sie jeweils, wie die Stelle ausgenutzt werden kann, um eine Instanz der Klasse `Person` nach ihrer Erstellung zu verändern.
Hinweis: Denken Sie nicht nur an die direkten Instanzen von `Person`, sondern auch an die indirekten Instanzen.

ii. *(6 Punkte)* Ändern Sie den Code der Klasse `Person` so, dass ihre Instanzen unveränderlich sind. Es muss weiterhin möglich sein, die Attributwerte der Klasse von außen abzufragen.

Aufgabe 2 Hamsterhäuser (`.java`, PDF)

In dieser Aufgabe bauen Sie Häuser für Ihre Hamster. Dabei werden Sie noch mal den Umgang mit eigenen Klassen und offensivem und defensivem Programmieren üben, und bereits bekanntes Wissen zu Verträgen, dem Umgang mit logischen Ausdrücken und Kontrollflussstrukturen wiederverwenden.

Ein Hamsterhaus (`House`) besteht aus mindestens einer Hauswand (`HouseWall`). Jede Wand kann optional eine Tür (`Door`) enthalten. Eine Hauswand wird über die Position ihres ersten und letzten Feldes beschrieben, siehe Abbildung 1. Dabei muss das erste Feld (`start`) immer näher am Ursprung des Territoriums liegen, als das letzte (`end`). Außerdem ist eine Hauswand immer mindestens zwei Felder lang, d.h. `start` und `end` der Hauswand dürfen nicht auf dem gleichen Feld liegen. Eine Hauswand kann zudem eine optionale Tür enthalten. Wenn eine Hauswand eine Tür enthält, dann muss die Tür zwischen dem ersten und dem letzten Feld der Wand liegen, aber nicht direkt auf diesen, siehe ebenfalls Abbildung 1.

Um ein Haus zu bauen, das heißt, um all Hauswände des Hauses auf dem Territorium zu platzieren, gibt es in dieser Aufgabe die Klasse `HouseBuilder`. Im Laufe dieser Aufgabe werden Sie sich zuerst mit den Bestandteilen eines Hauses auseinandersetzen, und dann die Klasse `HouseBuilder` vervollständigen.

Hinweis: Denken Sie in **allen** Teilaufgaben daran, die gegebenen Vorbedingungen zu überprüfen. Wir werden Sie nicht mehr jedes mal explizit dazu auffordern!

Hinweis: Bereits implementierte Operationen dürfen und sollen Sie in den folgenden Teilaufgaben wiederverwenden. Bei Bedarf dürfen und sollen Sie nach eigenem Ermessen neue Operationen anlegen.

Hinweis: In dieser Aufgabe haben Sie es mit zwei unterschiedlichen Arten von Wänden zu tun: den Wänden auf dem Territorium (Klasse `Wall` aus dem Hamstersimulator), und den Hauswänden der

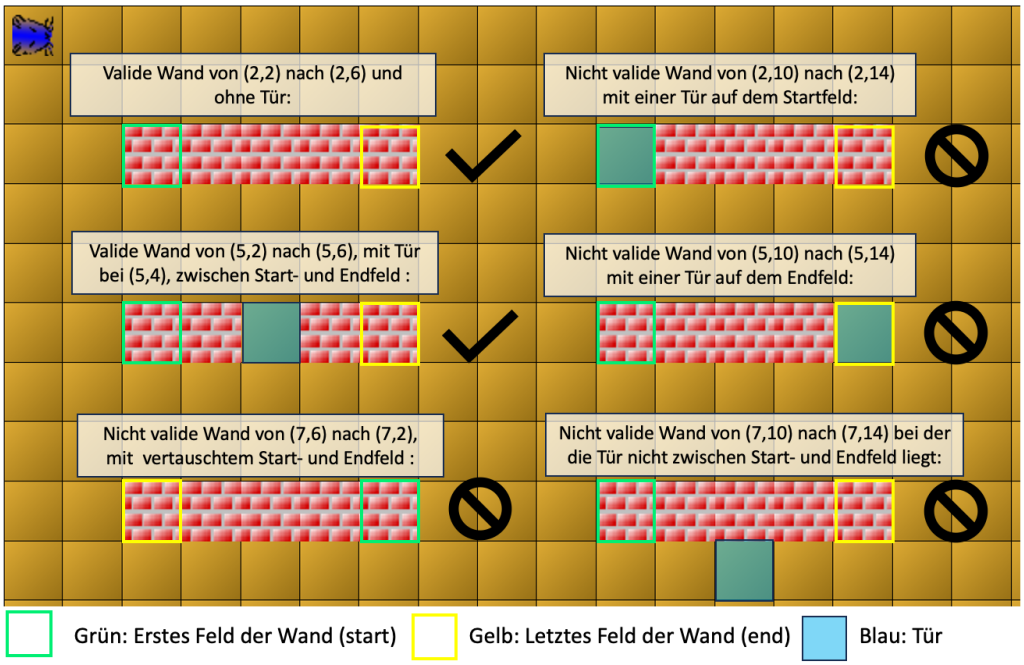

Abbildung 1: Beispiele verschiedener Wände mit und ohne Türen. Die beiden Wände links oben sind erlaubt, alle anderen entsprechen nicht den gegebenen Invarianten.

Häuser (Klasse `HouseWall` aus dem Paket `...habitat.house`). Eine Hauswand wird durch mehrere Wände auf dem Territorium dargestellt. Bringen Sie die Wände nicht durcheinander.

(a) *(4 Punkte)* Wozu gibt es sowohl offensives als auch defensives Programmieren in Java? Wann wird welche Programmierweise verwendet und welche Schlüsselworte sind dabei relevant?

(b) *(6 Punkte)* Öffnen Sie die Klasse `HouseWall` aus dem Paket `.habitat.house`. Lesen Sie die Dokumentation der Klasse sorgfältig durch. Im JavaDoc der Klasse werden unter anderem in natürlicher Sprache die Invarianten der Klasse beschrieben.

Vervollständigen Sie den Konstruktor der Klasse `HouseWall` und stellen Sie sicher, dass die neue Wand die beschriebenen Invarianten erfüllt. Ergänzen Sie, wenn sinnvoll, weiter Hilfsoperationen.

(c) *(6 Punkte)* Öffnen Sie die Klasse `HouseWall` aus dem Paket `.habitat.house`. Vervollständigen Sie die Operation `addDoor`. Ergänzen Sie, wenn sinnvoll, weiter Hilfsoperationen.

(d) *(7 Punkte)* Öffnen Sie die Klasse `HabitatHamsterGame` aus dem Paket `.habitat.game`. Vervollständigen Sie den Konstruktor an der mit einem `TODO` Kommentar markierten Stelle. Erzeugen Sie eine Instanz der Klasse `HouseBuilder`. Bauen Sie das in Abbildung 2 dargestellte Haus nach. Nutzen Sie dazu die Operationen `withWall` und `build`.

Hinweis: Falls Sie das Spiel in diesem Zustand starten, ist das Haus auf dem Territorium noch nicht zu sehen. Paule wird versuchen, einmal um sein Haus herum zu laufen um die Positionen der Wände und Türen zu überprüfen. Da das Haus noch nicht zu sehen ist, wird er sich über fehlende Wände beschweren.

Hinweis: Achten Sie darauf, dass sich die Hauswände, insbesondere an den Ecken des Hauses, nicht überlappen dürfen. Achten Sie außerdem darauf, dass das Haus genau zwei Türen haben soll.

(e) *(7 Punkte)* Öffnen Sie die Klasse `HouseBuilder` aus dem Paket `.habitat.house`. Die Klasse enthält unter anderem die Operation `buildWall`, die eine einzelne Hauswand auf dem Territorium platzieren soll. Implementieren Sie die Operation `buildWall`. Nutzen Sie das bereits vordefinierte Objekt `territoryBuilder` um die Wandteile, aus denen die Hauswand besteht, auf dem Territorium zu platzieren.

Führen Sie das Spiel aus. Nun sollte das Haus, das Sie in der vorherigen Teilaufgabe gebaut haben, auf dem Territorium zu sehen sein.

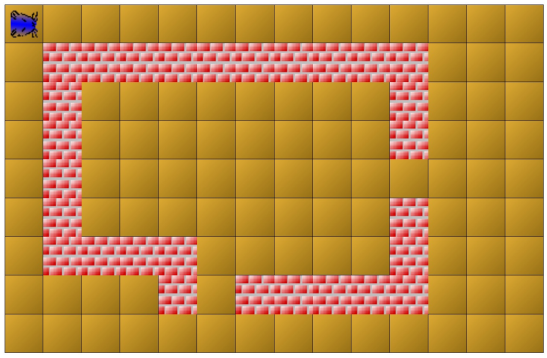

Abbildung 2: Ein Hamsterhaus mit zwei Türen.

Hinweis: Die Klasse `TerritoryBuilder` bietet eine Operation `wallAt` an, die eine einzelne Wand platziert. Weitere Informationen zu der Klasse finden Sie in der Dokumentation des Hamstersimulators.

Hinweis: Achten Sie darauf, dass Sie an den Positionen, an denen sich Türen befinden, keine Wände auf dem Territorium platzieren.

(f) *(4 Punkte)* Die Klasse `House` besitzt eine Operation `getDoors`, die die Menge aller in den Wänden des Hauses angebrachten Türen zurückgeben soll. Im Moment gibt die Operation jedoch nur eine leere Menge zurück.

Beschreiben Sie zwei unterschiedliche Möglichkeiten, die Operation `getDoors` zu implementieren. Nennen Sie Vor- und Nachteile, und Argumentieren Sie, unter welchen Umständen Sie welche Möglichkeit wählen würden.

Hinweis: Denken Sie an das Prinzip des gleichförmigen Zugriffs.

(g) *(5 Bonuspunkte)* Implementieren Sie die Operation `getDoors`.