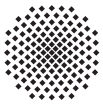


Universität Stuttgart

SQA
Software Quality and Architecture Group

Übungsblatt 09

Programmierung und Softwareentwicklung (WS 2024/2025)

Abgabe: Fr. 20.12.2024, 23:59 Uhr — Besprechung: KW 51

- 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), Ihre Matrikelnummern und Ihre 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: PSE09-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: Dieses Blatt beschäftigt sich mit den Konzepten der objektorientierten Programmierung und mit Datenstrukturen. Darüber hinaus lernen Sie im Heimarbeitsteil ein Entwurfsmuster kennen.

Punkte: Dieses Übungsblatt enthält zwei Teile. In Teil B können Sie bis zu 35 Punkte und 5 Bonuspunkte erzielen. Zum Bestehen des Blatts benötigen Sie mindestens 17.5 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-09`
- Stellen Sie sicher, dass Sie Übungsblatt 08 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 Overriding vs. Overloading

Ziel dieser Aufgabe ist es, die Unterschiede zwischen Überschreiben und Überladen zu vertiefen.

(a) Öffnen Sie die Klasse `OverrideUsageApp` aus dem Paket `presence.overrideOverload` und betrachten Sie die `main`-Operation. Welche Ausgabe erwarten Sie auf der Konsole, wenn Sie das Programm ausführen würden? Schreiben Sie Ihre Ergebnisse auf. Führen Sie das Programm anschließend aus.

(b) Öffnen Sie die Klasse `OverloadedUsageApp` aus dem Paket `presence.overrideOverload` und betrachten Sie die `main`-Operation. Welche Ausgabe erwarten Sie auf der Konsole, wenn Sie das Programm ausführen würden? Schreiben Sie Ihre Ergebnisse auf. Führen Sie das Programm anschließend aus.

Hinweis: Das Schlüsselwort `static` wurde im Foliensatz "*12 Java Klassen Erstellen*" eingeführt.

(c) Nutzen Sie die Fachbegriffe *Polymorphie*, *polymorphe Bindung*, bzw. *polymorphe Zuweisung*[1]., *statischer Typ*, *dynamischer Typ* und *dynamisches Binden*, um das Verhalten der Teilaufgaben (a) und (b) zu erklären. Schreiben Sie Ihre Ergebnisse auf.

Hinweis: Sie haben die genannten Fachbegriffe bereits auf Übungsblatt 08, Teil A, Aufgabe 1 in eigenen Worten beschrieben.

Aufgabe 2 Polymorphie und Collections

In dieser Aufgabe vertiefen Sie den Umgang mit verschiedenen Implementierungen des `List` Interfaces und beschäftigen sich nebenher nochmal mit Polymorphie.

Über die folgenden Teilaufgaben hinweg, werden Sie nach und nach ausprobieren, wie lange es dauert verschiedene Anzahlen an Elementen, an verschiedenen Stellen, in verschiedenen Typen von Listen einzufügen oder aus diesen zu holen. Dabei werden Sie für jede Kombination messen, wie lange die Ausführung dauert, die Werte aufschreiben und zuletzt vergleichen. Halten Sie die gemessen Zeiten in der folgenden Tabelle (oder einer ähnlichen) fest.

	`add(E e)`		`add(int index, E e)`	`get(int index)`	
	1.000	10.000	100	1.000	10.000
`LinkedList`					
`ArrayList`					
Thread-safe Liste					

Tabelle 1: Tabelle für die Messergebnisse aus Teilaufgabe (c) und (f).

Hinweis: Das Ausführen der Operation in den folgenden Teilaufgabe kann unter Umständen einige Zeit dauern. Falls das Ausführen zu lange dauert, verringern Sie die Anzahl der Elemente, die eingefügt bzw. geholt werden.

(a) Machen Sie sich mit der Klasse `Timer` aus dem Paket `presence.polymorphismCollections` vertraut (*nicht* der Timer aus der Base Class Library). Verwenden Sie Instanzen dieser Klasse in den folgenden Teilaufgaben zum Messen der Ausführungszeiten.

(b) Öffnen Sie die Klasse `ListTester` und fügen Sie der Klasse zwei neue Operationen hinzu, die jeweils ein Listen-Objekt entgegen nehmen – also einen Parameter von Typ `List<String>` haben. Die eine Operation soll 1.000 Strings am Ende der übergebenen Liste einfügen, und die andere Operation soll 10.000 am Ende der übergebenen Liste einfügen. In beiden Operationen sollen außerdem gemessen werden wie lange das Einfügen der Strings dauert. Die Messung soll dann auf der Konsole ausgeben werden. Verwenden Sie Instanzen der Klasse `Timer`, um die Ausführungszeiten zu messen.

[1] Zur Erinnerung: Während der Laufzeit werden Objekte and Entitäten (Variablen) gebunden. Die Zuweisung (`=`) ist das in der Sprache integrierte Kommando, das eine solche Bindung herstellt, ändert oder löscht.

(c) Rufen Sie die neuen Operationen aus Teilaufgabe (b) in der `main` Operation je dreimal auf. Übergeben Sie jeweils ein Listen-Objekt vom Typ `LinkedList`, `ArrayList` und ein Thread-safes[2] Listen-Objekt.

Stellen Sie eine Hypothese auf, welche der drei Listen die schnellste ist. Führen Sie das Programm aus und schreiben Sie die Ausführungszeiten auf.

Hinweis: Sie müssen die oben genannten Listen-Objekte selbst erstellen.

Hinweis: Verwenden Sie die Operation `synchronizedList(...)` aus der `Collections` Klasse um eine Thread-safe Liste zu erstellen. Listing 1 zeigt ein Beispiel. Weitere Informationen finden Sie in der Dokumentation der JBCL.

```
1  List<String> syncedList = Collections.synchronizedList(new ArrayList<>());
```

Listing 1: Beispiel zur Erstellung einer synchronisierten Liste.

(d) Fügen Sie der Klasse `ListTester` analog zu Teilaufgabe (b) eine Operation hinzu, die 100 Strings am Anfang eines übergebenen Listen-Objekts einfügt und die Ausführungszeit misst.

(e) Fügen Sie der Klasse `ListTester` analog zu Teilaufgabe (b) Operationen hinzu, die 1.000 bzw. 10.000 Elemente über ihren Index aus einem übergebenen Listen-Objekt holt und die Ausführungszeit misst.

(f) Fügen Sie der `main` Operation weitere Aufrufe hinzu. Rufen Sie die neuen Operationen aus Teilaufgabe (d)und (e) je dreimal auf. Übergeben Sie jeweils ein Listen-Objekt vom Typ `LinkedList`, `ArrayList` und ein Thread-safes Listen-Objekt.

Stellen Sie jeweils eine Hypothese auf, welche der drei Listen die schnellste ist. Führen Sie das Programm aus und schreiben Sie die Ausführungszeiten auf.

(g) Diskutieren Sie mit Ihrem Partner die vorherigen Ergebnisse, und vergleichen Sie diese mit Ihren Annahmen vor der Ausführung. Falls hierbei Abweichungen auftreten, überlegen Sie sich, was der Grund hierfür gewesen sein könnte.

(h) Diskutieren Sie mit Ihrem Partner wo Sie in der Klasse `ListTester` polymorphe Bindungen, beziehungsweise polymorphe Zuweisung genutzt haben.

[2] Ob eine Liste Thread-safe ist oder nicht, wird erst bei nebenläufigen Programmen relevant. Nebeläufigkeit ist kein Thema der PSE. Wir verwenden an dieser Stelle die Thread-Safe Liste nur, um die Auswirkungen auf die Laufzeit zu beobachten.

2 Teil B - Heimarbeit

Aufgabe 1 Die Hamster-Olympics (`.java`, PDF)

Bald sind wir im neuen Jahr angekommen. Für die Hamstergemeinschaft bedeutet das, dass die lang erwarteten Hamster-Olympics anstehen. In dieser Aufgabe geht es darum, verschiedene Möglichkeiten kennen zu lernen, um unterschiedliches Verhalten zu modellieren und zu implementieren, sowie deren Vor- und Nachteile kennen zu lernen. Öffnen Sie dazu im Projekt des Aufgabenblatts das Paket `homework.olympics`. Dort finden Sie Klassen, mit denen ein Hamster-Marathonrennen durchgeführt werden kann. Die Klasse `RunnerHamster` stellt einen Teilnehmer eines solchen Rennens dar. Sie sollen im Folgenden die dort bereits implementierten Operationen nutzen, um die Hamster durch ein Rennen zu navigieren. Machen Sie sich also gut mit diesen Operationen vertraut.

Die Klasse `Race` führt ein Rennen durch. Jeder teilnehmende Läufer darf dort reihum eine Bewegungsaktion auswählen, die durchgeführt werden soll. Der Laufkurs des Rennens führt vom Startpunkt, oben links, zum Zielbereich im unteren Bereich (vgl. Abbildung 1).

Abbildung 1: Die Rennstrecke der Rennen.

Hinweis: Wir werden Sie in dieser Aufgabe nicht mehr dazu auffordern, das Hamsterspiel auszuführen. Führen Sie das Hamsterspiel an beliebigen Stellen eigenständig aus, um zu überprüfen, ob Ihre Implementierungen der einzelnen Teilaufgaben das erwartete Verhalten zeigen.

(a) *(2 Punkte)* Das Teilnehmerfeld eines Rennens besteht aus verschiedenen Läufern, von denen jeder mit einer etwas anderen Taktik laufen möchte. Sie haben in der Vorlesung bereits kennen gelernt, dass sich das durch Vererbung umsetzen lässt. Skizzieren Sie in einem Schaubild, wie eine Vererbungshierarchie aussehen könnte, wenn es drei unterschiedliche Arten von rennenden Hamstern gibt:

- *SprintingHamster*: läuft immer so schnell, wie es seine aktuellem Energiereserven zulassen. Wenn die Energiereserven zu gering sind, um ganz schnell zulaufen, läuft er mit mittlerer Geschwindigkeit weiter. Wenn die Energiereserven dann zu gering sind, um mit mittlerer Geschwindigkeit zu laufen, läuft er mit langsamer Geschwindigkeit weiter.
- *SteadyRunnerHamster*: läuft, solange seine Energiereserven es zulassen, mit mittlerer Geschwindigkeit. Wenn seine Energiereserven aufgebraucht sind, läuft er mit langsamer Geschwindigkeit weiter.
- *SlowlyRunnerHamster*: läuft dauerhaft mit langsamer Geschwindigkeit.

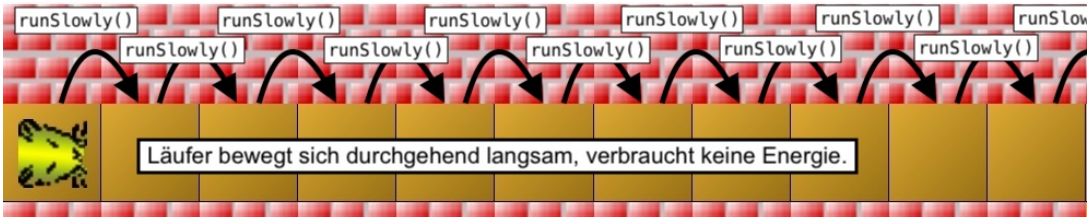
(b) *(5 Punkte)* Vererbung ist nicht immer die beste Wahl, um solches Verhalten in Software zu modellieren. Recherchieren Sie das „Strategy Design Pattern“. Beschreiben Sie dieses Pattern in eigenen Worten. Geben Sie anschließend zwei Gründe an, warum das Pattern in diesem Fall eine bessere Wahl sein könnte, um die verschiedenen Typen von Läufern umzusetzen.

(c) *(5 Punkte)* Sie sollen dieses Pattern verwenden, um die unterschiedlichen Läufertypen zu implementieren. Dazu existieren die Klassen `RunSlowlyRacePlan`, `RunSteadilyRacePlan` und `SprinterRacePlan`, die alle das Interface `RacePlan` implementieren. Die Klasse `RunSlowlyRacePlan` wurde bereits vollständig implementiert, die anderen beiden Klasse sind noch unvollständig. Vervollständigen Sie die Klassen `RunSteadilyRacePlan` und `SprinterRacePlan`.

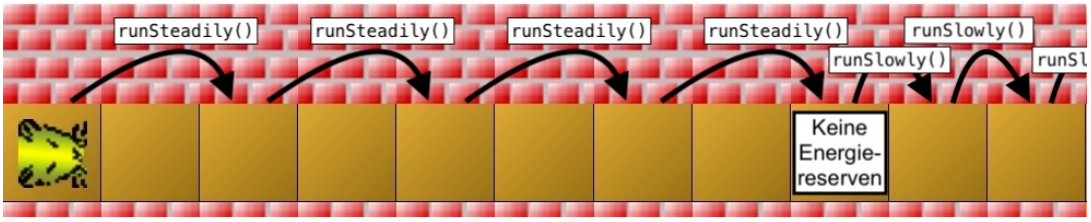
Die API der Klasse `RunnerHamster` bietet verschiedene Operationen, die einen Läufer, falls er noch ausreichend Energiereserven hat, genau ein Mal mit langsamer, mittlerer, oder schneller Geschwindigkeit bewegen. Kombinieren Sie diese Operationen, um den Hamster entsprechend den in Teilaufgabe (a) beschrieben Verhalten zu bewegen.

Hinweis: Lesen Sie die Dokumentation der Klasse `RunnerHamster` sorgfältig durch. Sie finden dort weiter Informationen, unter welchen Bedingungen welche Bewegungsaktion verwendet werden kann.

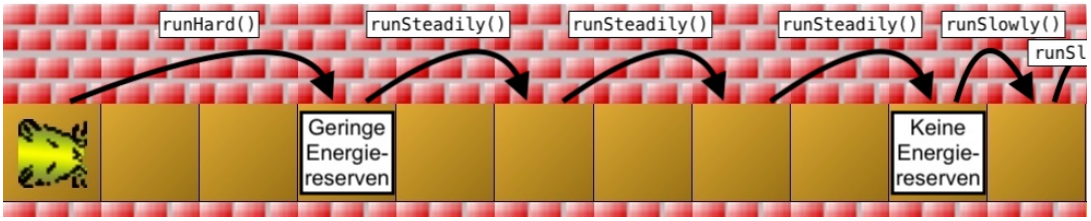
Hinweis: Abbildung 2 zeigt mögliche Bewegungsaktionen der verschiedenen Taktiken.



(a) Mögliche Bewegungsaktionen eines Hamster, der die `RunSlowlyRacePlan` Taktik nutzt.



(b) Mögliche Bewegungsaktionen eines Hamster, der die `RunSteadilyRacePlan` Taktik nutzt.



(c) Mögliche Bewegungsaktionen eines Hamsters, der die `SprinterRacePlan` Taktik nutzt.

Abbildung 2: Mögliche Bewegungsaktionen der verschiedenen Taktiken.

(d) *(6 Punkte)* Einige Läufer haben sich beim olympischen Hamsterverband beschwert, dass die Laufstrecke zu lang ist, um sie ohne zusätzliche Verpflegung zu schaffen. Dieser lenkt ein und führt nun Verpflegungsstationen auf der Strecke ein, an denen die Läufer neue Energie tanken können. Dazu müssen sie an einer Verpflegungsstation stehen (markiert durch Körner, siehe Abbildung 1) und dürfen dann pro Zug nur **entweder** einmal Verpflegung aufnehmen **oder** sich fortbewegen. Auch hier gibt es wieder unterschiedliche Strategien unter den Läufern. Einige Hamster möchten nie Energie aufnehmen, andere möchten an jeder Station einmal Energie aufnehmen, andere an jeder Station zweimal.

Dazu existieren die Klassen `FeedNeverStrategy`, `FeedOnceStrategy` und `FeedTwiceStrategy`, die alle das Interface `FeedingStrategy` implementieren. Die Klasse `FeedNeverStrategy` wur-

de bereits vollständig implementiert, die anderen beiden Klasse sind noch unvollständig. Vervollständigen Sie die Klassen `FeedOnceStrategy` und `FeedTwiceStrategy` entsprechend dem eben beschrieben Verhalten.

Hinweis: Bearbeiten Sie ausschließlich die Klassen `FeedOnceStrategy` und `FeedTwiceStrategy`. Überlegen Sie sich ein geeignetes Vorgehen, um sich innerhalb dieser Klassen zu merken, ob, beziehungsweise wie oft, ein Läufer eine Verpflegungsstationen bereits genutzt hat. Deklarieren Sie, falls nötig, zusätzliche Attribute.

(e) *(2 Punkte)* Nehmen Sie nun wieder an, Sie hätten sich dafür entschieden, jede Strategie wie in Teilaufgabe (a) durch Erweiterung der Klasse `RunnerHamster` umzusetzen. Wie sähe ihre Vererbungshierarchie aus, nachdem für jede dort erstellte Laufstrategie nun auch das unterschiedliche Verhalten der Verpflegungstaktiken berücksichtigt werden muss? Skizzieren Sie wieder eine Vererbungshierarchie.

(f) *(5 Bonuspunkte)* Die besten Läufer der Hamsterwelt bleiben auch auf der Laufstrecke immer flexibel. Einer von ihnen möchte nun außerhalb des Wettkampfs einen Rekordversuch starten. In der Klasse `OlympicsHamsterGame` finden Sie die Operation `recordAttempt`. Bringen Sie den dort bereits initialisierten `RunnerHamster` innerhalb von 30 oder weniger Aktionen ins Ziel.

Hinweis: Nutzen Sie dazu nur bereits in den vorherigen Teilaufgaben implementierte Strategien zum Laufen und Verpflegen durch einen Aufruf der Operation `executeNextAction`.

Hinweis: Als Aktion zählt jeder Aufruf der Operation `executeNextAction`.

Aufgabe 2 OOP - Theorie (PDF)

Ziel dieser Aufgabe ist es, die theoretischen Grundlagen der Objektorientierung zu vertiefen.

Betrachten Sie die Listings 2 und 3. Nutzen Sie zum Lösen der folgenden Teilaufgaben die Fachbegriffe aus der Vorlesung.

```
1  public void runGame() {
2      Hamster paule = new Hamster(/*territory, location, ...*/);
3      Hamster hamster = new RotatingHamster(/*territory, location, ...*/);
4
5      if (Math.random() < 0.5) {
6        paule = hamster;
7      }
8
9      paule.move();
10     paule.turnLeft();
11     paule.move(5);
12 }
```

Listing 2: Auszug aus der Operation `runGame`. Der Konstruktoraufruf ist vereinfacht dargestellt.

```
1  class RotatingHamster extends Hamster {
2
3      public RotatingHamster(final Territory territory, final Location location,
4                             final Direction direction, final int grainCount) {
5          super(territory, location, direction, grainCount);
6      }
7
8      @Override
9      public void turnLeft() {
10         super.turnLeft();
11         super.turnLeft();
12         super.turnLeft();
13         super.turnLeft();
14         super.turnLeft();
15     }
16
17     public void move(final int times) {
18         for(int i = 0; i < times; i++) {
19             move();
20         }
21     }
22 }
```

Listing 3: RotatingHamster.java

(a) *(4 Punkte)* Welchen statischen Typ hat `paule` ab Zeile 6 der Ausführung der Operation `runGame` in Listing 2? Welche dynamischen Typen könnte `paule` ab Zeile 6 haben? Begründen Sie Ihre Antworten. Zu welchem Zeitpunk wird entschieden, welchen dynamischen Typ `paule` ab Zeile 6 tatsächlich hat?

(b) *(5 Punkte)* Erläutern Sie, warum die Operation `runGame` fehlerhaft ist und warum der Code so nicht kompilieren kann. Klassifizieren Sie den Fehler als lexikalisch, syntaktisch, statisch semantisch oder dynamisch semantisch. Beschreiben Sie eine mögliche Korrektur, welche die erwünschte Funktionalität umsetzt und den in der Vorlesung vorgestellten Stilregeln entspricht.

(c) *(4 Punkte)* Beschreiben Sie die Begriffe *dynamisches Binden* und *polymorphe Zuweisung*. Nennen Sie je ein Beispiel aus Listing 2.

(d) *(2 Punkte)* Einige Programmiersprachen, wie z.B. C++, erlauben die Mehrfachvererbung. Beschreiben Sie in eigenen Worten, welches Problem durch Mehrfachvererbung entstehen kann.