Heinrich-Heine-Universität Düsseldorf
Institut für Informatik
Dr. Jens Bendisposto, Dr. Markus Brenneis

26. September 2022

Klausur

Programmierpraktikum 1
SS 2022

Hinweise:

- Diese Klausur enthält 7 nummerierte Klausurseiten. Prüfen Sie bitte zuerst, ob die Klausur alle Seiten enthält.
- Täuschungsversuche führen zum sofortigen Ausschluss von der Klausur. Die Klausur wird dann als nicht bestanden gewertet.
- Schreiben Sie **nicht** mit radierbaren Stiften und auch **nicht** mit rot!

Nachname: _____ Vorname: _____

Matrikelnummer: _____

Hörsaal: _____ Sitzplatznummer: _____

Unterschrift: _____

☐ Ich möchte im WS 22/23 **nicht** an der Veranstaltung *Programmierpraktikum 2* teilnehmen.

Viel Erfolg!

Diesen Teil bitte nicht ausfüllen:

Aufgabe	1	2	3	4	5	6	Σ
Punktzahl	6	4	4	5	4	5	28
Erreicht							

Aufgabe 1 [6 Punkte]

Wir wollen eine Software für die Qualitätskontrolle in einer Fabrik schreiben, die Würfel für Casinos herstellt. Die Würfel sollen für jede Augenzahl die gleiche Wahrscheinlichkeit haben. In der Fabrik werden die Würfel testweise geworfen und die Daten zu jedem einzelnen Wurf in einer Liste festgehalten. Ein einzelner Wurf wird in folgendem Record gespeichert, in welchem neben dem Ergebnis auch der Zeitpunkt des Wurfs und der Name der Person, die gewürfelt hat, festgehalten wird.

```
public record Wurf (LocalDateTime zeitpunkt,
                    String name,
                    int ergebnis) {}
```

Für die Qualitätssicherung benötigen wir eine Methode, mit der wir die Häufigkeitsverteilung für einen Würfel bestimmen können. Vervollständigen Sie die Methode `histogramm`, sodass sie ein Histogramm aus den Testwurf-Daten erzeugt. Ein Histogramm ist eine Map, in der für jede Augenzahl die Anzahl der Würfe, bei der diese Augenzahl aufgetreten ist, gespeichert ist.

Beispiel: `{1=10, 2=9, 3=11, 4=12, 5=9, 6=10}`

Ihre Lösung **muss genau einen** Stream verwenden und darf keine anderen Kontrollstrukturen verwenden.

```
public static Map<Integer, Long> histogramm(List<Wurf> wuerfe) {









}
```

Aufgabe 2 [4 Punkte]

Wir wollen eine Multimap implementieren, die es im Gegensatz zu einer normalen Map erlaubt, zu einem Schlüssel mehr als einen Wert zu speichern. Der Einfachheit halber erlauben wir nur Strings als Schlüssel und Werte. Werte können mehrfach, auch unter demselben Schlüssel, vorkommen. Implementieren Sie die folgenden Methoden und alle notwendigen Attribute der Klasse.

- eine Methode `add`, welche einen Schlüssel und einen Wert übergeben bekommt und den Wert unter dem Schlüssel abspeichert
- eine Methode `get`, welche einen Schlüssel übergeben bekommt und die gespeicherten Werte zu diesem Schlüssel zurückgibt

Die Multimap könnte folgendermaßen verwendet werden:

```
MultiMap multiMap = new MultiMap();
multiMap.add("3A", "Alex");
multiMap.add("3A", "Sascha");
multiMap.add("3A", "Alex");
multiMap.add("5D", "Alex");
// Resultat von multiMap.get("3A") enthält zweimal Alex und einmal Sascha
// Resultat von multiMap.get("5D") enthält nur Alex
```

Verwenden Sie ausschließlich Datenstrukturen, die in Java bereits vorhanden sind. Ihre Implementierung muss ähnlich effizient wie eine normale `Map` sein und darf keine Exceptions werfen.

```
public class MultiMap {









}
```

Aufgabe 3 [4 Punkte]

Wir wollen nun für die Klasse `MultiMap` aus Aufgabe 2 einen Builder entwickeln. Der Builder soll keine Validierungen durchführen, sondern nur eine `MultiMap` konstruieren, welche möglicherweise schon Elemente enthält.

Implementieren Sie händisch einen Builder für die Multimap. Wählen Sie Methodennamen, die den Zweck der entsprechenden Methoden klar kommunizieren. Sie können davon ausgehen, dass Aufgabe 2 gelöst wurde und eine `MultiMap`-Klasse mit den Methoden `add` und `get` vorhanden ist.

```
public class MultiMapBuilder {









}
```

Aufgabe 4 [5 Punkte]

Wir wollen nun die Klasse `MultiMap` testen.

(a) [3 Punkte] Schreiben Sie eine vollständige Testmethode, die überprüft, dass wir unter dem Schlüssel `3A` tatsächlich die drei Strings wie im Aufrufbeispiel von Aufgabe 2 speichern und wieder abrufen können. Sie können davon ausgehen, dass JUnit, Mockito und AssertJ bereits importiert sind.

```
public class MultiMapTest {









}
```

(b) [2 Punkte] Welche Teile in Ihrem Code machen den Act-Schritt aus? Markieren Sie die Stellen im Code und erklären Sie in 1–2 Sätzen, warum der Act-Schritt aus diesen Stellen besteht.

Aufgabe 5 [4 Punkte]

Die 327,6 Millionen Dollar teure Mars Climate Orbiter Mission der NASA ist 1999 an einem Software-Bug gescheitert. Der Grund war die Verwendung unterschiedlicher Maßeinheiten in zwei von unterschiedlichen Teams entwickelten Software-Komponenten.

Eines der Teams hat das metrische System verwendet und so den Impuls in newton-seconds angegeben. Das andere Team hat imperiale Einheiten und so den Impuls in pound-force-seconds angegeben. Beide Komponenten haben fehlerfrei funktioniert und waren getestet, aber Zahlen, die von einem in das andere System übergeben wurden, waren um einen Faktor von ca. 4,45 falsch.

Wir stellen uns nun vor, wir hätten einen (vollkommen frei erfundenen) Code-Schnippsel einer der beiden Komponenten in Java vorliegen. Die andere Komponente, welche für die Messung des Impulses zuständig ist, heißt in diesem Code `sensor`.

```
double impulse = sensor.getImpulse();
if (checkError(impulse) > TOLERANCE) {
  applyCorrection(impulse);
}
```

(a) [2 Punkte] Welcher Code-Smell ist in dem Code erkennbar? Beschreiben Sie ihn in 1–2 Sätzen und begründen Sie, warum in diesem Code der Smell vorliegt.

(b) [2 Punkte] Wie hätte die NASA diesen Fehler **durch Änderung dieses Codes** (und des Codes in `sensor`) verhindern können? Schreiben Sie keinen Code, sondern erklären Sie in 1–2 Sätzen, wie eine Lösung für das Problem aussieht.

Aufgabe 6 [5 Punkte]

Wir haben einen Test für eine Klasse, der folgendermaßen aussieht:

```
@Test
@DisplayName("Die Mail mit der höchsten Priorität wird zuerst verschickt")
void test_35() {
  MailSender sender = mock(MailSender.class);
  SendQueue queue = new SendQueue(sender);
  Mail lowPrio = new Mail("test1@example.com",2);
  queue.addMail(lowPrio);
  Mail highPrio = new Mail("test1@example.com",10000);
  queue.addMail(highPrio);
  queue.send("subject", "body");
  assertThat(queue.content()).containsExactly(lowPrio);
  verify(sender.send(highPrio.getMail(), "subject", "body"));
}
```

(a) [3 Punkte] Beurteilen Sie die **getestete** Klasse im Hinblick auf die SOLID-Prinzipien: Kreuzen Sie in der Tabelle für jedes Prinzip an, ob es verletzt ist oder nicht. Begründen Sie für alle **verletzten** Prinzipien, warum eine Verletzung vorliegt. Verwenden Sie nur das Wissen, das Sie durch den Test über die getestete Klasse ableiten können.

Prinzip	Verletzt? (ankreuzen)		Begründung (nur wenn „Ja“ angekreuzt)
	Ja	Nein/keine Aussage möglich	
S			
O			
L			
I			
D			

(b) [2 Punkte] Der Assert-Schritt des Tests ist auch nicht optimal. Was ist das Problem und wie können wir es verbessern?