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

31. Juli 2023

Klausur

Programmierpraktikum 1
SS 2023

Hinweise:

- Diese Klausur enthält 11 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: _____

Viel Erfolg!

Diesen Teil bitte nicht ausfüllen:

Aufgabe	1	2	3	4	5	6	Σ
Punktzahl	5	3	4	4	4	6	26
Erreicht							

Aufgabe 1 [5 Punkte]

Wir schreiben Code für die Registrierung von neuen Nutzerkonten in unserer Anwendung.

```
public interface UserDatabase {
    boolean containsUser(String name);
}

public class UserRegistration {
    private final UserDatabase db;

    public UserRegistration(UserDatabase db) {
        this.db = db;
    }

    public boolean isValidNewUserName(String name) {
        if(db.containsUser(name)) {
            return false;
        }
        if(name.startsWith(" ")) {
            return false;
        }
        return true;
    }
}
```

(a) [4 Punkte] Schreiben Sie einen Test für die Methode `isValidNewUserName` für den Fall, dass sich eine Person mit dem Username `"foobar123"` anmelden will und `containsUser` `false` zurückgibt.

```
@Test
@DisplayName("siehe (b)")
void test_1() {
```

Lösung:

```
UserDatabase db = name -> false; // Mockito auch ok
UserRegistration res = new UserRegistartion(db);
boolean reslt = res.isValidNewUserName("fooar123");
assertThat(reslt).isTrue();
```

```
}
```

(b) [1 Punkt] Geben Sie einen Displaynamen für den Test an, der genau angibt, welchen Fall der Test abdeckt.

Lösung:
Ein Benutzername, der bisher nicht in der Datenbank steht und nicht mit einem Leerzeichen beginnt, ist valide.

Aufgabe 2 [3 Punkte]

Wir betreiben einen kleinen Handwerksbetrieb. In unserer Betriebssoftware gibt es unter anderem folgende Funktionalitäten:

- Für die Erstellung von Gehaltsabrechnungen wird berechnet, wie viel Prozent des Brutto-Arbeitslohns an die Sozial-Versicherungen gezahlt werden müssen. Unter anderem gehen aktuell 2 % an die Arbeitslosenversicherung.
- Unsere Angestellten können sich in einer Simulation ausgeben lassen, wie viel von einem Brutto-Lohn an die Sozial-Versicherungen bezahlt wird.
- Außerdem können wir Rechnungen für unsere Kundschaft ausstellen. Allen, die innerhalb von 7 Tagen bezahlen, gewähren wir aktuell einen Rabatt von 2 %.

Die Implementierung dieser Funktionalitäten sieht so aus:

```
class Gehaltssimulation {
    // ...

    static int arbeitslosenversicherung(int bruttoLohn) {
        return 2 * bruttoLohn / 100;
    }
}

record Angestellter(int lohn) {
    // ...

    Map<String, Integer> sozialabgaben() {
        Map<String, Integer> abgaben = new HashMap<>();
        abgaben.put("Arbeitslosenversicherung", (int)(0.02 * lohn));
        // ...
        return abgaben;
    }
}

record Rechnung(int betrag) {
    private final static int RABATT = 2;

    // ...

    public double rabattBeiZeitnaherZahlung() {
        return betrag * RABATT / 100;
    }
}
```

(a) [2 Punkte] Die Berechnung $\frac{2}{100} \cdot x$ wird an drei Stellen durchgeführt. Durch welche dieser Stellen wird das DRY-Prinzip (nicht) verletzt? Begründen Sie Ihre Antwort; gehen Sie dabei auf alle drei Stellen ein.

Lösung:
erste und zweite, weil gleicher Änderungsgrund
drittes nicht, weil anderer Sachverhalt

(b) [1 Punkt] Was müsste in Zukunft passieren, damit sich die identifizierten DRY-Probleme negativ auf die Wartbarkeit auswirken?

Lösung:
Änderung des Sozialversicherungssatzes

Aufgabe 3 [4 Punkte]

Wir entwickeln einen Online-Shop mithilfe von Spring Boot. Je nach Land, in dem der Shop betrieben wird, soll entweder die Zahlungsart Kreditkarte oder Girocard verwendet werden. Die Zahlungsart wird in der `application.properties` z. B. mit `country=DE` festgelegt.

(a) [3 Punkte] Vervollständigen Sie die Annotationen, sodass die Dependencies automatisch injectet werden.

```
public interface PaymentService {
  void processPayment(int amount);
}
public class CreditCardPaymentService implements PaymentService {
  public void processPayment(int amount) { /* ... */ }
}
public class GiroCardPaymentService implements PaymentService {
  public void processPayment(int amount) { /* ... */ }
}

@____________
public class ShoppingCart {
  private final PaymentService paymentService;

  public ShoppingCart(PaymentService paymentService) {
    this.paymentService = paymentService;
  }

  public void checkout(int amount) {
    paymentService.processPayment(amount);
  }
}

@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }

  @____________
  CommandLineRunner commandLineRunner(ShoppingCart shoppingCart) {
    return args -> shoppingCart.checkout(998);
  }

  @____________
  PaymentService serviceSelect(@____________("${country}") String country) {
    switch(country) {
      case "DE":
        return new GiroCardPaymentService();
      default:
        return new CreditCardPaymentService();
    }
  }
}
```

Lösung:
Component/Service, Bean, Bean, Value

(b) [1 Punkt] `CreditCardPaymentService` und `GiroCardPaymentService` haben keine Spring-Annotationen. Ist das korrekt oder wurde das hier nur vergessen? Begründen Sie Ihre Antwort.

Lösung:
richtig, wenn @Configuartion an beidem stehen würde, gäbe es drei Möglichkeiten, wie ein PaymentService ausgewählt werden könnte

oder anders begründet: die @Bean-Methode reicht, damit Spring die Abhängigkeiten findet

Aufgabe 4 [4 Punkte]

Wir entwickeln die Verwaltungssoftware für eine Bibliothek. In der Bibliothek können Bücher ausgeliehen werden. Für den Betrieb der Bibliothek ist wichtig, dass nie mehr als 50 % der Bücher ausgeliehen werden.

Wir haben zwei Klassen geschrieben, die ein Aggregat bilden sollen. Den Code finden Sie **auf der nächsten Seite**.

Das Aggregat verstößt gegen eine der Regeln, die wir über den Aufbau von Aggregaten kennengelernt haben.

(a) Was besagt diese Regel?

(b) Woran erkennen Sie, dass das Aggregat gegen diese Regel verstößt? Geben Sie die konkrete Stelle im Code an.

(c) Wie kann dieser Regelverstoß in diesem konkreten Kontext zu einem Problem werden?

(d) Machen Sie einen Vorschlag, wie das Problem, das sie im vorherigen Aufgabenteil beschrieben haben, durch Änderung des Aggregats verhindert werden kann.

Lösung:

1. veränderlicher Zustand darf Aggregat nicht verlassen
2. allBooks
3. ein herausgegebenes Buch kann auf ausgeliehen gesetzt werden, ohne dass ausgeleiheneBuecher aktualisiert wird
4. String-Liste der Titel rausgeben statt der Buch-Objekt (oder Kopie der Objekte erzeugen) (ausgeleiheneBuecher immer frisch berechnen ginge hier jetzt auch)

```
public class Bibliothek { // aggregate root
    private final Set<Book> books = new HashSet<>();
    private long ausgelieheneBuecher = 0;

    public boolean ausleihen(String titel) {
        // sucht Buch mit dem angegebenen Titel
        // sofern verfügbar und ausgelieheneBuecher+1 <= books.size()/2.0,
        //  wird das Buch als ausgeliehen markiert, ausgelieheneBuecher++
        //  und true zurückgegeben
        // ansonsten Rückgabewert false
    }

    public Set<Book> allBooks() {
        return new HashSet<>(books);
    }

    public void addBook(String titel) {
        books.add(new Book(UUID.randomUUID(), titel));
    }
}
```

```
class Book {
    private final UUID id;
    private final String titel;
    private boolean ausgeliehen = false;

    public Book(UUID id, String titel) {
        this.id = id;
        this.titel = titel;
    }

    public boolean ausleihen() {
        if(!ausgeliehen) {
            ausgeliehen = true;
            return true;
        }
        return false;
    }

    public void zurueckgeben() { ausgeliehen = false; }

    public boolean ausleihbar() { return !ausgeliehen; }

    public boolean hatTitel(String titel) { return this.titel.equals(titel); }

    /* equals und hashcode von IDE erzeugt,
       sodass jeweils nur die id verwendet wird */

}
```

Aufgabe 5 [4 Punkte]

Für eine Software zur Zugangskontrolle möchte Markus Tests schreiben. Innerhalb der Tests benötigt er oft eine Klasse `Zugangspunkt`, die konfiguriert werden muss. Jeder Zugangspunkt speichert eine Liste berechtigter Personen und einen Scanner. Der Konstruktor ist `Zugangspunkt(List<String> berechtigtePersonen, Scanner scanner)`.

Um die Lesbarkeit seiner Tests zu erhöhen, möchte er einen `ZugangspunktBuilder` mit Fluent-API haben, um Zugangspunkte zu konfigurieren.

Der Builder soll wie folgt verwendet werden können:

```
Scanner scanner = mock(Scanner.class);
Zugangspunkt zugangspunkt = new ZugangspunktBuilder()
    .berechtigt("Marlin")
    .berechtigt("Kim")
    .withScanner(scanner)
    .build();
```

Falls kein Scanner über den Builder konfiguriert wird, soll der Zugangspunkt einen Mockito-Dummy (mit Standardkonfiguration) verwenden. Mehr als die im Verwendungsbeispiel aufgeführten Methoden muss der Builder nicht besitzen.

```
public class ZugangspunktBuilder {
```

Lösung:

```
  private List<String> names = new ArrayList<>();
  private Scanner scanner = mock(Scanner.class);

  public ZugangspunktBuilder berechtigt(String name) {
    names.add(name);
    return this;
  }

  public ZugangspunktBuilder withScanner() {
    this.scanner = scanner;
    return this;
  }

  public Zugangspunkt build() {
   return new Zugangspunkt(names, scanner);
  }
```

```
}
```

Aufgabe 6 [6 Punkte]

Sie können Aufgabenteil b) bearbeiten, auch wenn sie a) nicht gelöst haben.

Für ein Restaurant soll eine Anwendung entwickelt werden, die den Wert der Produkte im Lager verwaltet. Für jedes Produkt ist gespeichert, welchen Wert es pro Stück hat (`preis`), wie viel Stück im Lager sind (`anzahl`) und ob das Produkt schnell verderblich ist (`verderblich`).

```
public record Produkt(String name, int preis, int anzahl, boolean verderblich) {}
```

(a) [5 Punkte] Implementieren Sie die Methode `gesamtwert`, sodass sie den Gesamtwert der übergebenen Produkte, aufgeteilt in verderblich und nicht verderblich, zurückgibt.

Beispiel:

```
gesamtwert(List.of(
    new Produkt("Milch", 95, 2, true),
    new Produkt("Nudeln", 85, 1, false),
    new Produkt("Bohnen", 55, 2, false)));

// => {true: 190,   // 2×95
       false: 195}  // 1×85 + 2×55)
```

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

Tipp: `Collector summingInt(ToIntFunction mapper)`

```
static Map<Boolean, Integer> gesamtwert(List<Produkt> produkte) {
```

Lösung:

```
public static Map<Boolean, Integer> gesamtwert(List<Produkt> prodkute) {
    return prodkute.stream()
        .collect(Collectors.groupingBy( // oder partitioningBy
            Produkt::verderblich,
            Collectors.summingInt(p -> p.preis() * p.anzahl())));
}
```

```
}
```

(b) [1 Punkt] Ist die Methode `gesamtwert` gut testbar? **Begründen Sie Ihre Antwort** und schlagen Sie ggf. vor, wie die Aufgabenstellung (a) angepasst werden müsste, um eine einfacher testbare Methode zu erhalten.

Lösung:

gut testbar weil pure