Lösungsversuch SQS-Praxistest 2017W

Angabe und Lösungsversuch des SQS-Praxistest aus dem Semester 2017W. Keine Garantie auf Richtigkeit der Lösung.

1. Ein Online Lieferdienst für Lebensmittel führt ein Bonuspunktesystem ein. Bestellungen sind als registrierter sowie als anonymer Kunde möglich. Registrierte Kunden erhalten ab einer Einkaufssumme von ≥ 10 € nach dem Einkauf zwei Bonuspunkte gutgeschrieben. Ab zehn Bonuspunkten werden diese automatisch eingelöst und die Einkaufssumme wird um 2 € reduziert. Definieren Sie Anhand der gegebenen Informationen alle Äquivalenzklassen. Und geben Sie zwei wichtige Testfälle (nicht JUnit-Tests) auf Basis von Grenzwerten an.

Klassen:

A1: anonymer Kunde
A2: registrierter Kunde
B1: Einkaufssumme ≥ 10 €
B2: Einkaufssumme < 10 €
C1: Bonuspunkte ≥ 10
C2: Bonuspunkt < 10

2. Kann für die Klasse `MoneyTransferService` ein Integrationstest mit Hilfe von Mocking durchgeführt werden? Begründen Sie Ihre Antwort ausführlich.

```
1  public class MoneyTransferService {
2
3      private BankService bs;
4
5      public void setBankServicee(BankService bs) { this.bs = bs; }
6
7      public MoneyTransferResult transferMoney(String ibanFrom, String ibanTo, int amount) {
8          this.bs = new SimpleBankService();
9          if(!isValid(ibanFrom))
10             throw new InvalidIbanException("Invalid from IBAN");
11         if(!isValid(ibanTo))
12             throw new InvalidIbanException("Invalid to IBAN");
13         if(amount <= 0)
14             throw new InvalidAmountException("Amount must be positive");
15         return bs.transfer(ibanFrom, ibanTo, amount);
16     }
17
18 }
```

Nein, nicht mockbar, da in Zeile 8 der Mock immer wieder überschrieben werden würde.
Ebenfalls widerspricht ein Mock dem Integrationstest, da dieser nur bei Unit-Tests bzw. Komponententests angewendet werden soll.
(Ein Integrationstest testet das Zusammenspiel zwischen den beiden Komponenten, daher macht ein Mock hier auch gar keinen Sinn.)

```
1  public class SetTest {
2
3      private static Set<Integer> integersSet;
4
5      @BeforeClass
6      public static void setUp() {
7          integersSet = new HashSet<>();
8          integersSet.add(1);
9          integersSet.add(2);
10         integersSet.add(3);
11     }
12
13     @Test
14     public void testSetDoesSomethingShouldFail() {
15         assert(!integersSet.remove(3));
16     }
17
18     @Test
19     public void setRemoveShouldReturnTrueIfElementIsRemoved() {
20         for(int i = 1; i <= 3; i++) {
21             if(!integersSet.remove(i))
22                 throw new IllegalStateException("Failed to remove value from set");
23         }
24     }
25
26     @Test(expected = IllegalStateException.class)
27     public void addingExistingElementToSetShouldEnsureElementExistsOnlyOnce() {
28         integersSet.add(1);
29         integersSet.remove(1);
30         if(!integersSet.remove(1))
31             throw new IllegalStateException("Failed to remove value from set");
32     }
33 }
```

Finden Sie fünf Fehler in der Implementierung der Testfälle und erklären Sie die gefundenen Probleme und geben Sie Vorschläge zur Behebung.
Achten Sie im Speziellen auf Testing Bad Practices!
Hinweis: Sollte ein Fehler mehrfach vorkommen, zählt dieser nur als ein Fehler.

1) Zeile [3,5,6] Hier sollte @Before verwendet werden und nicht-statische Methoden und Variablen, da die Tests sonst voneinander abhängen.

2) Zeile [15] Hier wird das Java-Keyword assert() verwendet, diese hat nichts mit JUnit-Tests zu tun.

3) Zeile [13] Sehr schlechte Benennung des Tests.

4) Zeile [21,22] if-throw sollte ~~da~~ durch ein assertXX() ersetzt werden. "If" in Tests generell vermeiden.

5) Zeile [29] Erstes remove sollte mit einem assertTrue() geprüft werden, um sicher zu gehen, dass nicht schon beim ersten Element ein Fehler auftritt.

4. Bei der folgenden Klasse `AlarmService` handelt es sich um ein Service das zwischen einer definierten Startzeit und Endzeit eine Alarmanlage aktiviert. Testen Sie die Methode `isAlarmActive(int startHour, int endHour, boolean daylightSaving)` auf Ihre korrekte Funktionsweise, indem Sie folgende Testfälle in JUnit-Syntax mit mocking implementieren. (Verwenden Sie kein Mocking Framework!).

a) `startHour = 20, endHour = 6, daylightSaving = false; result: true`

b) `startHour = 8, endHour = 16, daylightSaving = true, result: false`

```
1  interface Clock {
2      // throws IllegalArgumentException if timeZoneOffset is not in range [+-24]
3      Integer getDayTimeInMillis(Integer timeZoneOffset) throws IllegalArgumentException;
4  }
5  class AlarmService {
6      private Clock clock;
7      public boolean isAlarmActive(int startHour, int endHour, boolean daylightSaving) {
8          Integer dayTime = clock.getDayTimeInMillis(daylightSaving ? 1 : 0) / 1000 / 60 / 60;
9          if(startHour <= endHour) return (dayTime >= startHour) && (dayTime <= endHour);
10         else return (dayTime <= startHour) && (dayTime >= endHour);
11     }
12     public void setClock(Clock clock) { this.clock = clock; }
13 }
```

```
public class AlarmTest {
    public ClockMock mock;
    public AlarmService alarm;

    @Before
    public void bef() {
        mock = new ClockMock();
        alarm = new AlarmService();
        alarm.setClock(mock);
    }

    @Test
    public void testIfAlarmIsActive() {
        mock.setTime(0);
        assertTrue(alarm.isAlarmActive(20,6,false));
    }

    @Test
    public void testIfAlarmIsInactive() {
        mock.setTime(0);
        assertFalse(alarm.isAlarmActive(8,16,true));
    }
}

public class ClockMock {
    private int time;
    public void setTime(int t) {
        time = t;
    }

    public Integer(Integer offset) throws Illeg...
    {
        if (Math.abs(offset) > 24) {
            throw new IllegalArgumentException();
        }
        return time;
    }
}
```

5. Implementieren Sie folgende vorgegebenen Tests in JUnit

a) `removeCorrectValueForGivenKey()`:
Überprüft für den richtigen Value der richtige Key zurückgegeben und aus der Map entfernt wird.

b) `removeShouldFailWhenKeyDoesNotExist()`:
Überprüft ob eine `NoSuchElementException` beim Aufruf von `remove` geworfen wird, wenn der Key nicht existiert.

c) `removeAllShouldRemoveAllElements()`:
Überprüft ob alle Elemente aus der Map entfernt wurden.

Ihre Aufgabe ist die Überprüfung der korrekten Funktionsweise der Methoden `remove(<key>)` und `removeAll()`. Die Methode `remove(<key>)` liefert den Wert am angegebenen Key und entfernt ihn aus der Map. Sollte die Map an der Stelle leer sein, wird eine `NoSuchElementException` geworfen. Die Methode `removeAll()` löscht alle Elemente. Zusätzlich steht Ihnen die sicher richtig implementierte Methode `size` zur Verfügung, die die Anzahl der Elemente in der Map zurückliefert.

```
1  public class MapTest {
2      private MyMap<String, Integer> map;
3      private static final String NAME_1 = "Jane";
4      private static final String NAME_2 = "John";
5      private static final String AGE_1 = 25;
6      private static final String AGE_2 = 26;
7      @Before
8      public void setUp() {
9          map = MyMap<>();
10         map.put(NAME_1, AGE_1);
11         map.put(NAME_2, AGE_2);
12     }
```

```
@Test
public void removeCorrectValueForGivenKey() {
    assertEquals(map.remove(NAME_1), AGE_1);
    assertEquals(map.remove(NAME_2), AGE_2);
    assertEquals(map.size(), 0);
}

@Test(expected = NoSuchElementException.class)
public void removeShouldFailIfKeyDoesNotExist() {
    map.remove("test");
}

@Test
public removeAllShouldRemoveAllElements() {
    map.removeAll();
    assertEquals(map.size(), 0);
}
```