Übungsblatt 4

Ausgabe: 28.11.2022 Besprechung: 12.-15.12.2022 10,5 Punkte

Bitte laden Sie die Übungsbeispiele rechtzeitig bis Sonntag vor der Übungswoche (11.12.2022) bis spätestens 23:55 als Zip Datei (`<Matrikelnummer>_<Name>_SE1_ÜB<Nummer>`) in Moodle hoch und tragen Sie Ihre Kreuze in ZEUS ein. Die Zip Datei muss alle gelösten Beispiele (auch den Source Code) und evtl. deren Präsentationen beinhalten. Da es in ZEUS keine halben Punkte gibt, wurde die Anzahl der Punkte für ZEUS verdoppelt → laut ZEUS können Sie für diese Übung 32 Punkte bekommen. Für die Übung selbst werden aber die am Übungsblatt angegebenen Punkte gezählt → es können maximal 16 Punkte erreicht werden (= Anzahl der Punkte in ZEUS / 2). Sie müssen in der Lage sein, die Lösung und den Lösungsweg Ihren KollegenInnen und der LV-Leitung in der Übungsstunde via Beamer erklären können. Sollten Sie Fragen zur Übung haben oder Unklarheiten auftreten, nutzen Sie bitte das Moodle-Forum.

Anmerkung zum Kreuzen: Da die Beispiele in diesem Übungsblatt grob-granular sind, gibt es auf die Teilaufgaben Punkte, um zu verhindern, dass Sie alles oder nichts kreuzen müssen. Das heißt, Sie kreuzen in der Kreuzerlliste die **Teilpunkte** die Sie gelöst haben und erklären können. Sie finden bei jeder Teilaufgabe, für die es Punkte gibt, am Ende in Klammer stehend wie viele Punkte Sie für diese Aufgabe bekommen und eine "Kreuzerlnummer", um Ihnen das Kreuzen zu erleichtern.

Generelle Anmerkung: Benutzen Sie für diese Übung das im Moodle bereitgestellte Projekt. Vermerken Sie bitte Ihren **Namen** und Ihre **Matrikelnummer** als Kommentar in all Ihren Klassen. Weiters werden in dieser Einheit die Tools JUnit[1] zum automatisierten Testen, JaCoCo[2] zur Messung der Code Coverage, und PIT[3] zur Durchführung der Mutations Analyse verwendet. Es wird empfohlen sich vor Beginn der Übung mit den Tools auseinander zu setzen. Eine Einführung zu den Themen finden Sie unter anderem auf Google. Einige Tutorials sind auch hier angeführt: JUnit `http://www.vogella.com/tutorials/JUnit/article.html`. Die Webseite des Buches "xUnit Test Patterns"[4] bietet zudem einen Überblick über Best Practises zum Thema Testing (z.B. Wie sehen gute Tests aus?).

Abgabedateien

Die Zip Datei `<Matrikelnummer>_<Name>_SE1-ÜB<Nummer>` sollte folgende Dateien beinhalten:

- `4.1_reports.zip`
- `4.2_Analyse.pdf`
- `4.2_Schritt_<N>_reports.*`
- `project.zip` (Mit allen notwendigen Dateien!)

[1] `http://junit.org/junit4/`
[2] `http://www.eclemma.org/jacoco/`
[3] `http://pitest.org`
[4] `http://xunitpatterns.com`

Testing

4.1 Unit-Testen Einführung (3 Punkte, K1)

Gegeben ist die Klasse `at.aau.serg.exercises.shapetesting.RectangularShapeFactory`. Diese Klasse kann dazu verwendet werden, `Rectangle`- und `Square`-Objekte mit unterschiedlichen Seitenlängen zu erstellen. Ziels dieses Beispiels ist es, dass Sie sich mit den grundlegenden Funktionen von JUnit 5 vertraut machen. Hierzu testen Sie die korrekte Funktionsweise der Factory selbst sowie der Objekte, die von der Factory erstellten werden. Fügen Sie alle Tests, die Sie erstellen, zur bereitgestellten Testklasse `at.aau.serg.exercises.shapetesting.RectangularShapeFactoryTest` hinzu.

- Implementieren Sie Tests die mithilfe der JUnit 5 Methoden `assertTrue` bzw. `assertFalse` überprüfen, dass die mittels create erstellten `Square`s equilateral und die erstellten `Rectangle`s nicht equilateral sind.
- Implementieren Sie Tests die mithilfe der JUnit 5 Methoden `assertEquals` bzw. `assertNotEquals` überprüfen, dass `Square`s bzw. `Rectangle`s mit gleichen Seitenlängen äquivalent sind, bzw. solche mit ungleichen Seitenlängen nicht äquivalent sind.
- Implementieren Sie Tests die mithilfe der JUnit 5 Methode `assertThrows` überprüfen, dass Exceptions geworfen werden, wenn die `create`-Methoden mit ungültigen Werten aufgerufen werden.
- Implementieren Sie Tests die überprüfen, dass die `getSides`-Methoden von `Square`s bzw. `Rectangle`s jeweils 4 Side-Objekte mit den richtigen Längen zurückgeben.
- Implementieren Sie eine `setUp`-Methode, die ein Factory-Objekt erstellt. Passen Sie alle bestehenden Tests so an, dass nur mehr auf dieses eine Factory-Objekt zugegriffen wird. Annotieren Sie die `setUp`-Methode dafür entweder mit der Annotation `@BeforeEach` oder `@BeforeAll`. Begründen Sie die Wahl, die Sie bzgl. der Annotation getroffen haben.
- Wandeln Sie einen der Tests, die Sie erstellt haben, in einen parametrisierten Test um. Verwenden Sie hierfür die JUnit 5 Annotation `@ParameterizedTest` und etwaige weitere Annotationen, die für parametrisierte Tests benötigt werden. Verwenden Sie zumindest 3 verschiedene Parameterbelegungen.
- Erstellen Sie am Ende einen Coverage Report mit JaCoCo und einen Mutations Testing Report mit PIT und geben Sie diese auch ab. Wie können die Reports interpretiert werden?

4.2 Unit-Testen Praxis (2,5 Punkte)

Gegeben ist die Klasse `at.aau.serg.exercises.ringbuffer.RingBuffer`. Verwenden Sie für dieses Beispiel wieder das auf Moodle zur Verfügung gestellte Projekt. Verwenden Sie für Ihre Tests die bekannten Techniken aus der Vorlesung und den Übungen. Mit `mvn clean test` (oder alternativ via IDE) können Sie die Tests laufen lassen und die Reports generieren. Die Berichte befinden sich anschließend im `target`-Folder Ihres Projekts. Geben Sie abschließend die Berichte mit ab. Jeweils nach den Schritten 2,3 und 5 führen Sie folgende Messungen durch:

a. Messen Sie mittels JaCoCo die Coverage-Metriken für die Klassen und erstellen Sie den Coverage-Report.

b. Messen Sie die Qualität Ihrer Tests mittels dem Mutation Testing Tool PIT.[5]

Arbeitsschritte:

1. Überlegen Sie sich eine Strategie um die Klasse zu testen und notieren Sie sich Ihre Vorgehensweise (Wie gehen Sie vor um gut zu testen?, Welche Strategien wenden Sie an?, Welche Verfahren eignen sich hier gut?,...).
2. Implementieren Sie die Tests gemäß Ihrer Strategie mit Hilfe von JUnit in der Testklasse `at.aau.serg.exercises.ringbuffer.BaseTest`. **0,5 Punkte, K2)**
3. Erweitern Sie die Tests um volle Branch-Coverage in der Testklasse `at.aau.serg.exercises.ringbuffer.FullCoverageTest` zu erhalten. **(0,5 Punkte, K3)**
4. Erweitern Sie die Tests um 100% Mutation-Score von Jumble zu erreichen. (`at.aau.serg.exercises.ringbuffer.BaseTest`) **(1 Punkte, K4)**
5. Wie hängen Code-Coverage Metriken und Mutation Score zusammen? **(0,5 Punkte, K5)**

4.3 TDD Feature Implementieren (3 Punkte)

Gegeben ist die Klasse `at.aau.serg.exercises.tdd.MyCollection`. Diese Klasse stellt eine stark vereinfachte Implementierung einer Collection dar. In diesem Beispiel sollen Sie zwei Features (`remove()` und `empty()`) mittels Test Driven Development[6] implementieren. Um die Vorgehensweise zu dokumentieren, verwenden Sie Git.[7] Achten Sie darauf gute Commit-Messages zu verwenden.[8]

1. Erstellen Sie im Projektverzeichnis ein Git Repository (`git init`) und committen Sie das Projekt (`git add .` und `git commit`)
2. Folgen Sie nun der TDD Vorgehensweise und erstellen Sie Tests die die Spezifikation laut JavaDoc des ersten Features testen. **(0,5 Punkte, K6)**
3. Lassen Sie die Tests mittels `mvn test` laufen (die Tests müssen fehlschlagen) und committen Sie die Tests (`git commit`). **(0,5 Punkte, K7)**
4. Implementieren Sie die Funktionialität des Features. Die Tests müssen anschließend erfolgreich sein. **(0,5 Punkte, K8)**
5. Wiederholen Sie Schritte 2. bis 4. für das zweite Feature. **(1 Punkte, K9)**
6. Lassen Sie Coverage und Mutation Analyse laufen, verbessern Ihre Tests falls notwendig und committen Sie am Ende erneut. **(0,5 Punkte, K10)**

[5] `http://pitest.org`
[6] `https://www.it-agile.de/wissen/agiles-engineering/testgetriebene-entwicklung-tdd/`
[7] `https://git-scm.com/docs/gittutorial`
[8] `https://chris.beams.io/posts/git-commit/`

4.4 Basispfade nach McCabe (2 Punkte, K10)

Gegeben ist Listing 1.

Listing 1: Codebeispiel

```
1  public static int ggt(int zahl1, int zahl2) {
2      while (zahl2 != 0) {
3          if (zahl1 > zahl2) {
4              zahl1 = zahl1 - zahl2;
5          } else {
6              zahl2 = zahl2 - zahl1;
7          }
8      }
9      return zahl1;
10 }
```

1. Bestimmen Sie die Basis-Pfade nach McCabe.
2. Implementieren Sie konkrete Testfälle in der Klasse `at.aau.ue4.bsp3.McCabeTest`, sodass alle Basis-Pfade zumindest 1x durchlaufen werden und erstellen Sie einen Coverage Report und eine Mutationsanalyse. Was können Sie beobachten?
3. Bestimmen und interpretieren Sie die zyklomatische Komplexität nach McCabe.