UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Software Engineering im Wintersemester 2021/2022
Prof. Dr. Martin Leucker, Malte Schmitz, Stefan Benox, Julian Schulz, Benedikt Stepanek, Friederike Weilbeer, Tom Wetterich

Übungszettel 10 (Lösungsvorschlag)

03.01.2022

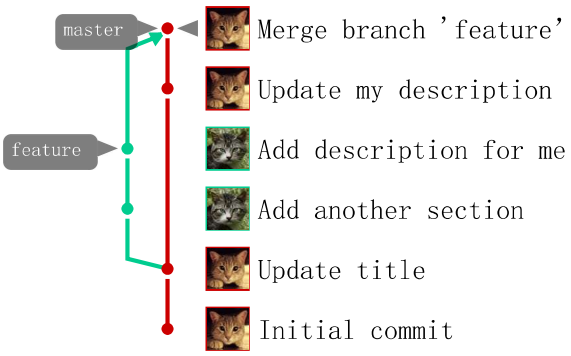
Abgabe bis Donnerstag, 20. Januar um 23:59 Uhr online im Moodle.

Aufgabe 10.1: Git

4 Punkte, mittel

In dieser Aufgabe wollen wir die Arbeit mit Git und Gitlab an einem praktischen Beispiel üben.

Die folgende Aufgabe müssen Sie nun zu zweit im Abgabeteam bearbeiten. Ein Mitglied Ihres Teams legt im Gitlab dieser Veranstaltung ein neues Projekt an und gibt dies für das andere Teammitglied frei. In diesem Projekt legen Sie nun eine Datei `README.md` an, in der Sie sich beide kurz beschreiben. Beide Teammitglieder schreiben dazu in die Datei eine Überschrift mit dem eigenen Namen und darunter einige beschreibende Sätze über sich. Tun Sie dies gemeinsam in der richtigen Reihenfolge, sodass sich folgender Git-Graph ergibt:



Sie sehen den Graph im Gitlab unter dem Menüpunkt *Graph* im Menü *Repository*.

Geben Sie in Ihrer Abgabe den Link zum Projekt im Gitlab an und laden Sie Ihre*n Tutor*in zu diesem Projekt ein.

▼ Lösungsvorschlag

Ich gehe in diesem Lösungsvorschlag davon aus, dass Lena und Felix die Abgabe gemeinsam machen. Lena beginnt und legt im Gitlab das Projekt *about* an und lädt Felix dazu ein. Lena klont nun das Repository:

```
git clone git@projects.isp.uni-luebeck.de:lena/about.git
```

Nun legt Lena eine Datei `README.md` mit dem Inhalt

```
# Über uns

Hier beschreiben wir uns
```

an und fügt diese dem Git hinzu:

```
git add README.md
git commit -m "Initial commit"
```

Nun passt Lena die Überschrift der Datei an, da Sie nachgeschaut hat, dass die Überschrift ihr Name sein soll:

```
# Lena

Hier beschreiben wir uns
```

Lena macht einen weiteren Commit mit dieser Änderungen und pusht ihr lokales Repository zu Gitlab:

```
git add README.md
git commit -m "Update title"
git push -u origin HEAD
```

Jetzt ist Felix dran. Er klont das Repository ebenfalls:

```
git clone git@projects.isp.uni-luebeck.de:lena/about.git
```

Felix legt direkt einen neuen Branch an:

```
git checkout -b feature
```

Nun editiert Felix die Datei und ergänzt einen weiteren Abschnitt für sich:

```
# Lena

Hier beschreiben wir uns

# Felix
```

Felix macht einen Commit mit dieser Änderung:

```
git add README.md
git commit -m "Add another section"
```

Nun ergänzt Felix eine Beschreibung über sich in der Datei:

```
# Lena

Hier beschreiben wir uns

# Felix

Ich bin Felix!
```

Felix macht einen Commit mit dieser Änderung und pusht seinen Branch in das Repository bei Gitlab:

```
git add README.md
git commit -m "Add description for me"
git push -u origin HEAD
```

Jetzt ist Lena wieder dran. Lena aktualisiert Ihre Beschreibung ohne die Änderungen von Felix zu beachten:

```
# Lena

Ich bin Lena!
```

Lena macht einen Commit mit dieser Änderung:

```
git add README.md
git commit -m "Update my description"
```

Jetzt pullt Lena die Änderungen von Felix:

```
git pull
```

Lena hat jetzt lokal den Branch `feature` von Felix und kann diesen mergen:

```
git merge feature
```

Dabei tritt ein Merge-Konflikt auf, da Lena und Felix die gleiche Stelle in der gleichen Datei editiert haben. Lena löst diesen Merge-Konflikt, in dem Lena die Datei editiert

```
# Lena

Ich bin Lena!

# Felix

Ich bin Felix!
```

Lena macht anschließend einen Commit und pusht ihr Repository zu Gitlab:

```
git add README.md
git commit
git push
```

Aufgabe 10.2: Testen mit JUnit

4 Punkte, leicht

Wir definieren in dieser Aufgabe Testfälle für das von Ihnen in Aufgabe 9.1 implementierte Java-Programm und implementieren diese Testfälle mit JUnit 5.

Zur Erinnerung: Das Programm nimmt drei Integer-Zahlen entgegen. Jede Zahl soll dabei als Seitenlänge eines Dreiecks verstanden werden. Das Programm gibt aus, ob das Dreieck mit diesen drei Seitenlängen gleichschenklig, gleichseitig oder ungleichseitig ist.

1. Schreiben Sie alle Eingaben (eine Eingabe sind jeweils drei Integerwerte) für diese Funktion auf, die Sie benötigen, um diese Funktion ausreichend zu testen.

Schreiben Sie jeweils kurz zu jeder Eingabe dazu, was diese Eingabe testet.

Berücksichtigen Sie dabei sowohl verschiedene Variationen gültiger Eingaben, als auch verschiedene Variationen ungültiger Eingaben, wie zum Beispiel negative Zahlen oder 0 sowie Eingaben, die kein Dreieck beschreiben.

▼ **Lösungsvorschlag**

- Ein ungleichseitiges Dreieck, z.B. (2,3,4).
- Drei gültige gleichschenklige Dreiecke, z.B. (2,2,3), (2,3,2) und (3,2,2).
- Ein gültiges gleichseitiges Dreieck, z.B. (2,2,2).
- Eine Eingabe, bei der eine Seite 0 ist, z.B. (0,2,3).
- Eine Eingabe, bei der eine Seite negativ ist, z.B. (-1,2,3).
- Drei Eingaben, die Geraden und keine Dreiecke beschreiben, z.B. (1,2,3), (1,3,2) und (3,2,1).
- Drei Eingaben, bei denen alle drei Zahlen größer als 0 sind, aber trotzdem kein Dreieck bilden können, z.B. (1,2,4), (1,4,2) und (4,2,1).
- Eine Eingabe, bei der alle Zahlen 0 sind, also (0,0,0).

2. Implementieren Sie die oben definierten Testfälle in Ihrem Java-Programm. Verwenden Sie JUnit 5 in Maven, sodass die Tests über `mvn test` ausgeführt werden können. Ergänzen Sie ggf. das Programm so, dass alle Tests erfüllt werden.

   Reichen Sie Ihren Code als Zip-Archiv im Moodle ein. Achten Sie dabei darauf, tatsächlich nur den Quellcode und keine gebauten Artefakte mit abzugeben.

   ▼ **Lösungsvorschlag**

   Siehe `triangle-tests.zip`.

Aufgabe 10.3: LTL-Semantik

4 Punkte, schwer

Betrachten Sie folgende minimale LTL-Syntax:

> $$\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \mathcal{X}\,\varphi \mid \varphi\,\mathcal{U}\,\varphi$$
>
> Dabei steht p für eine beliebige atomare Proposition.

Die zugehörige Semantik sei wie folgt:

> Für ein ω-Wort $w = w_0 w_1 \ldots \in \Sigma^\omega$ mit $w_i \in \Sigma$ ist die Modell-Relation $\models$ induktiv wie folgt definiert:
>
> $$\begin{array}{ll} w \models p & \text{falls } p \in w_0 \\ w \models \varphi_1 \vee \varphi_2 & \text{falls } w \models \varphi_1 \text{ oder } w \models \varphi_2 \\ w \models \neg\varphi & \text{falls } w \not\models \varphi \\ w \models \mathcal{X}\,\varphi & \text{falls } w^{(1)} \models \varphi \\ w \models \varphi_1\,\mathcal{U}\,\varphi_2 & \text{falls } \exists i\colon w^{(i)} \models \varphi_2 \text{ und } \forall j, 0 \leq j < i\colon w^{(j)} \models \varphi_1 \end{array}$$

1. Erläutern Sie, wieso diese minimale LTL-Variante in Ihrer Ausdrucksmächtigkeit nicht eingeschränkt ist gegenüber dem aus der Vorlesung bekannten LTL mit den zusätzlichen Operatoren $\mathbf{true}$, $\mathbf{false}$, $\wedge$, $\rightarrow$, $\leftrightarrow$, $\mathcal{F}$, $\mathcal{G}$ und $\mathcal{R}$.

   ▼ **Lösungsvorschlag**

   Diese zusätzlichen Operatoren sind lediglich syntaktischer Zucker. Sie können über folgende Äquivalenzen unter Verwendung der Operatoren aus der minimalen Syntax definiert werden:

   $$\begin{aligned} \mathbf{true} &:\equiv a \vee \neg a \\ \mathbf{false} &:\equiv \neg\mathbf{true} \\ \varphi_1 \wedge \varphi_2 &:\equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) \\ \varphi_1 \rightarrow \varphi_2 &:\equiv \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 &:\equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\ \mathcal{F}\,\varphi &:\equiv \mathbf{true}\,\mathcal{U}\,\varphi \\ \mathcal{G}\,\varphi &:\equiv \neg\,\mathcal{F}\,\neg\varphi \\ \varphi_1\,\mathcal{R}\,\varphi_2 &:\equiv \neg(\neg\varphi_1\,\mathcal{U}\,\neg\varphi_2) \end{aligned}$$

2. Betrachten Sie folgende alternative LTL-Synax:

   > $$\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi\mathcal{U}^{+}\varphi$$

> Dabei steht p für eine beliebige atomare Proposition.

Die zugehörige Semantik sei wie folgt:

> Für ein ω-Wort $w = w_0 w_1 \ldots \in \Sigma^\omega$ mit $w_i \in \Sigma$ ist die Modell-Relation $\models$ induktiv wie folgt definiert:
>
> $$
> \begin{array}{ll}
> w \models p & \text{falls } p \in w_0 \\
> w \models \varphi_1 \vee \varphi_2 & \text{falls } w \models \varphi_1 \text{ oder } w \models \varphi_2 \\
> w \models \neg\varphi & \text{falls } w \not\models \varphi \\
> w \models \varphi_1 \mathcal{U}^+ \varphi_2 & \text{falls } \exists i \geq 1 : w^{(i)} \models \varphi_2 \text{ und } \forall j, 1 \leq j < i : w^{(j)} \models \varphi_1
> \end{array}
> $$

Diskutieren Sie, wieso diese LTL-Variante die gleiche Ausdrucksmächtigkeit wie das aus der Vorlesung bekannte LTL besitzt.

▼ **Lösungsvorschlag**

Um die Gleichheit der Ausdrucksmächtigkeit zu zeigen, müssen wir nachweisen, dass jeder beliebige Ausdruck zwischen den beiden LTL-Varianten konvertiert werden kann.

Die Formel $\varphi_1 \mathcal{U}^+ \varphi_2$ kann im klassischen LTL ausgedrückt werden als $\mathcal{X}(\varphi_1 \mathcal{U} \varphi_2)$.

Umgekehrt kann $\varphi_1 \mathcal{U} \varphi_2$ in der neuen LTL-Variante dargestellt werden, in dem wir die Äquivalenz $\varphi_1 \mathcal{U} \varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \mathcal{X}(\varphi_1 \mathcal{U} \varphi_2))$ verwenden. Entsprechend lässt sich $\varphi_1 \mathcal{U} \varphi_2$ darstellen als $\varphi_2 \vee (\varphi_1 \wedge (\varphi_1 \mathcal{U}^+ \varphi_2))$. Die Formel $\mathcal{X}\varphi$ lässt sich darstellen als $\mathbf{false} \mathcal{U}^+ \varphi$, denn es gilt $\mathbf{false} \mathcal{U} \varphi \equiv \varphi$.

Formal könnte man diese Äquivalenzen beweisen durch Einsetzen der entsprechenden Definitionen.

3. Der Operator $\mathcal{U}^+$ wird als irreflexives Until bezeichnet, da er sich im Gegensatz zum klassischen Until nicht auf die aktuelle Position bezieht. Diskutieren Sie, wie ein irreflexives Finally $\mathcal{F}^+$ und ein irreflexives Globally $\mathcal{G}^+$ definiert werden könnten.

▼ **Lösungsvorschlag**

Wir wenden die aus der Definition von Finally und Globally bekannten Äquivalenzen auf das irreflexive Until an:

$$\mathcal{F}^+\varphi :\equiv \mathbf{true}\mathcal{U}^+\varphi$$
$$\mathcal{G}^+\varphi :\equiv \neg\mathcal{F}^+\neg\varphi$$

Damit erhalten wir folgende Zusammenhänge:

$$\mathcal{F}^+\varphi \equiv \mathcal{X}\,\mathcal{F}\,\varphi$$
$$\mathcal{G}^+\varphi \equiv \mathcal{X}\,\mathcal{G}\,\varphi$$

Formal könnte man diese Äquivalenzen beweisen durch Einsetzen der entsprechenden Definitionen.

Hinweise zur Verwendung von Git und Gitlab

Installieren Sie zunächst Git auf Ihrem Computer. Sie können dabei zum Beispiel dem Tutorial von Atlassian folgen.

Stellen Sie sicher, dass Git auf ihrem Computer korrekt funktioniert und machen Sie sich mit dem Tool in der Kommandozeile vertraut. Tutorials zur Verwendung von Git finden Sie zum Beispiel bei GitHub.

Wir nutzen Git in Kombination mit Gitlab. Registrieren Sie sich dazu einen Account an unserem Gitlab: Nutzen Sie dazu Ihre offizielle E-Mail-Adresse an der Uni-Lübeck als E-Mail-Adresse und den lokalen Teil dieser E-Mail-Adresse als Benutzername. Wenn Sie also die E-Mail-Adresse `erika.mustermann@student.uni-luebeck.de` haben, registrieren Sie sich mit dem Benutzernamen `erika.mustermann`. Registrierungen, die nicht diesem Schema entsprechen, werden kommentarlos gelöscht. Wir schalten die Registrierung manuell frei, aber wir bemühen uns, diese zeitnah zu tun.

Heben Sie Ihre Zugangsdaten gut auf. Sie werden diese für das Praktikum im nächsten Semester ebenfalls benutzen. Bitte beachten Sie, dass sie dieses Gitlab nur für diese Vorlesung und das anschließende Praktikum verwenden können. Wir werden das Gitlab am Ende des Praktikums löschen. Für andere Projekte können Sie sich zum Beispiel einen kostenlosen Account bei Gitlab.com registrieren.

Um das Gitlab optimal nutzen zu können, sollten Sie einen SSH-Key im Gitlab hinterlegen. Sie können zum Generieren des SSH-Key der Anleitung von Atlassian oder der Anleitung von Gitlab folgen. Die Anleitung von Gitlab erläutert, wie der SSH-Key im Gitlab hinzugefügt wird.

Hinweise zur Verwendung von JUnit mit Maven

Zunächst muss JUnit 5 als Dependency zum Projekt hinzugefügt werden. Ergänzen Sie dazu folgenden Abschnitt in der `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.7.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.7.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Weitere Informationen dazu finden Sie im Abschnitt *4.2. Build Support / 4.2.2. Maven* in der offiziellen Dokumentation.

Maven führt die Tests mit dem Surefire-Plugin aus. Wenn dieses Plugin in der `pom.xml` nicht explizit angegeben ist, wird je nach Umgebung eine deutlich ältere Version des Surefire-Plugins verwendet. Das hat zur Folge, dass Tests nur erkannt werden, wenn der Methodenname mit `test` beginnt. Um die Methoden nur durch die Annotation `@Test` zu markieren und einen beliebige Methodennamen wählen zu können, muss eine neuere Version des Surefire-Plugins explizit verlangt werden. Dazu muss folgende Definition in der `pom.xml` innerhalb von `<build><plugins>` ergänzt werden:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
</plugin>
```

Tests werden in der Maven-Ordnerstruktur in einem separaten Ordner abgelegt. Während sich der Code in `src/main` befindet, liegen die Tests in `src/test`. Tests werden üblicherweise im gleichen Paket wie die zu testenden Klassen abgelegt. Legen Sie also folgenden Ordner an: `src/test/java/triangle`.

In diesem Ordner legen Sie nun eine Klasse `TriangleTest` an:

```
package triangle;
```

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class TriangleTest {
    // TODO ...
}
```

Öffentliche Methoden in dieser Klasse, die mit der Annotation `@Test` ausgezeichnet werden, werden von JUnit als Tests ausgeführt.

Die Tests können mit Maven ausgeführt werden über

```
mvn test
```