Softwaretechnische Grundlagen
Dr. J. Bendisposto

16. Juni 2016
Abgabe bis 15. Juli 2016, 11:45 Uhr (UTC+1)

Gruppenprojekt

1 Lernziel

Das Ziel der Aufgabe ist es, eine komplexere Anwendung mit einer graphischen Oberfläche zu entwerfen und zu programmieren. Außerdem müssen Sie sich in eine neue Programmschnittstelle einarbeiten.

2 Aufgabe

In der Aufgabe bekommen Sie von uns eine Bibliothek, die Sie verwenden sollen, um eine Anwendung mit graphischer Schnittstelle (in JavaFX) zu schreiben.
Zusätzlich zu der Implementierung müssen Sie einen Bericht und ein Nutzerhandbuch einreichen.

- In dem Bericht soll erklärt werden, wie Ihre Anwendung funktioniert: Zum Beispiel welche Klassen es gibt und wie diese miteinander interagieren. Der Bericht soll nicht länger als 8 Seiten sein und soll den Korrektoren helfen Ihre Gedankengänge nachzuvollziehen.
- Das Nutzerhandbuch soll geeignet sein Teilnehmenden der Informatik 1 Vorlesung die Verwendung Ihres Werkzeugs zu erklären.

3 Anwendungsbeschreibung: TDDT

Die Anwendung TDDT (TDD Trainer) soll Studierenden in den ersten Semestern helfen, die Technik der testgetriebenen Entwicklung (test driven development, TDD) zu üben. Bei der testgetriebenen Entwicklung handelt es sich um eine Technik zur Softwareentwicklung, bei der ein Test vor dem zu testenden Code geschrieben wird. Sollten Ihre Kenntnisse über TDD etwas eingerostet sein, ist der Text von Frank Westphal (`http://www.frankwestphal.de/TestgetriebeneEntwicklung.html`) sehr zu empfehlen. TDDT soll den Prozess abbilden und den Benutzer führen.
Das Szenario soll in etwa so ablaufen:

1. Der Benutzer wählt aus einem Katalog von Übungsaufgaben eine Aufgabe aus.
2. **RED** Das Werkzeug erlaubt es dem Nutzer nur den Test zu editieren, bis es genau einen fehlschlagenden Test gibt oder der Code nicht compiliert[1]. Der Nutzer soll dem Programm mitteilen, dass er bereit ist für den nächsten Schritt. Der Wechsel zum nächsten Schritt darf aber nur erfolgen, wenn die Bedingung erfüllt ist.
3. **GREEN** Das Programm erlaubt es dem Nutzer nun ausschließlich den Code zu modifizieren, bis alle Tests laufen. Alternativ kann der Benutzer zurück zu **RED** wechseln[2]. Beim

[1] Wenn man den ersten Test für eine Methode schreibt, existiert die Methode noch nicht. Folglich kann der Test nicht kompiliert werden

[2] Wenn man einen fehlerhaften Test geschrieben hat der nicht sinnvoll erfüllt werden kann, muss man noch einmal zurück zum Test wechseln.

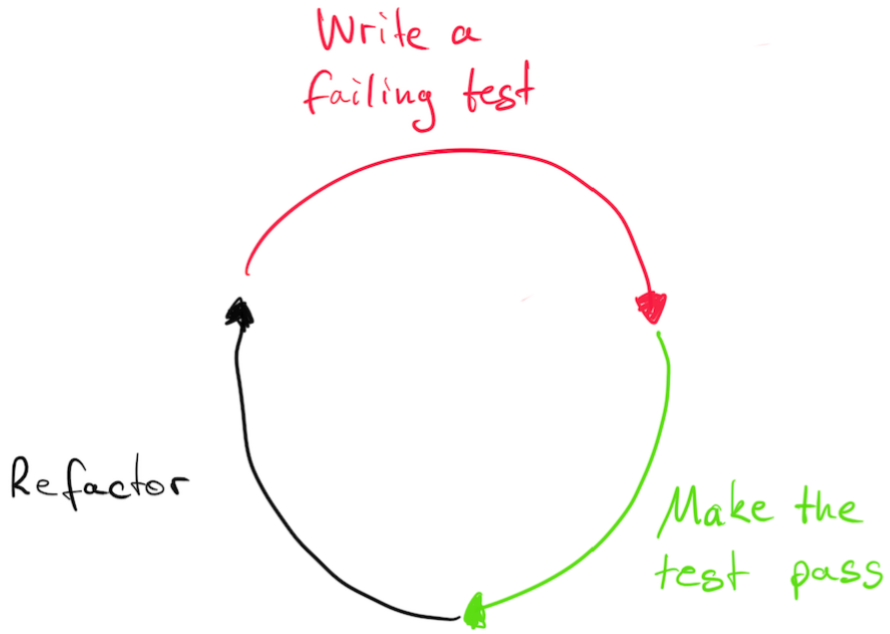

Abbildung 1: Entwickungsprozess: TDD

Zurückwechseln zu den Tests wird der neue Code gelöscht (d.h. der Code befindet sich dann in dem selben Zustand in dem er war, als der Benutzer in den Code-Editier-Modus eingetreten ist).

4. **REFACTOR** Nachdem alle Tests laufen, darf der Nutzer den Code verbessern (refactoring). Der Wechsel von **GREEN** nach **REFACTOR** muss der Benutzer dem Programm explizit mitteilen. Es ist nur erlaubt zu wechseln, wenn die Bedingungen (alles kompiliert und alle Tests laufen durch) erfüllt sind. Sie können auch einen zweiten Refactor Zustand einfügen: Nachdem der Code verbessert wurde, können die Tests ebenfalls verbessert werden. Wichtig ist aber das Refactoring immer bedeutet, dass alle Tests immer vorher und nachher laufen.

5. Wenn der Nutzer die Verbesserungsarbeiten beendet hat, beginnt der Prozess wieder bei Schritt 2 (**RED**).

Eine Reihe von weiteren Anforderungen sind wie folgt:

- Es gibt einen Katalog von möglichen Übungen. Der Katalog darf nicht fest in Ihrem Werkzeug eingebaut sein, sondern muss in irgendeiner Form nachgeladen werden.
- Der Katalog kann mit einem Texteditor einfach geändert und erweitert werden (es ist nicht nötig, dass der Katalog in Ihrem Werkzeug änderbar ist!)
- Eine Übung besteht in der Regel aus Vorlagen für die Klasse und Tests sowie der Konfiguration für Ihr Werkzeug.
- Im Anhang ist ein Beispiel, wie ein Katalog aufgebaut sein könnte. Sie sind hier aber vollkommen frei!
- Der TDD Prozess wird in irgendeiner Form visuell dargestellt. Der Nutzer soll zu jedem Zeitpunkt wissen, was zu tun ist, d.h., es muss klar sein, ob ein Test oder eine Klasse geändert werden soll und wie der Gesamtprozess aussieht.

4 Erweiterungen

Sie müssen zusätzlich zur Grundfunktionalität mindestens zwei der folgenden Erweiterungen in Ihrem Werkzeug implementieren.

4.1 Babysteps

Wenn für eine Übung Babysteps eingeschaltet sind, wird die Zeit, die der Nutzer in den Phasen **RED** und **GREEN** hat, limitiert. Ist die Zeit abgelaufen, wird der neue Test/Code gelöscht und es wird in die vorangegangene Phase zurückgewechselt. Babysteps haben den Sinn dem Nutzer die Entwicklung in kleineren Schritten nahezulegen/anzutrainieren. Die Zeit soll konfigurierbar sein. Ein üblicher Wert sind 2 -3 Minuten. Die Zeit für **REFACTOR** ist nicht limitiert.

4.2 Tracking

Die Tracking Funktion zeichnet auf, was der Benuzter wann geändert hat. Tracking soll erlauben herauszufinden für welche Aktivitäten der Nutzer viel Zeit benötigt und welche Fehler auftreten. Die Daten können analysiert werden um Probleme zu erkennen in die die Nutzer häufig laufen. Sie müssen zusätzlich zum reinen Tracking mindestens eine Analyse schreiben, die in Form eines Charts anzeigt, wie lange der Nutzer in den einzelnen Phasen verbracht hat.

4.3 ATDD

Bei ATDD gibt es zusätzlich zu den Unit-Tests auch noch Akzeptanztests. Ein Akzeptanztest ist ein Test, der grün wird, wenn ein Feature vollständig implementiert wurde. Der Arbeitsfluss ändert sich dadurch folgendermaßen:

- Als erstes wird ein Akzeptanztest geschrieben. Dieser Test bleibt solange rot, bis ein Feature vollständig implementiert wurde.
- Dann startet der normale TDD Zyklus. Das Feature wird mit Unit-Tests schrittweise entwickelt, bis alle Tests (also auch der Akzeptanztest) grün werden.
- Dann wird der nächste Akzeptanztest geschrieben.

Für diese Erweiterung müssen Sie Akzeptanztests und Unit-Tests unterscheiden und es gibt zusätzliche Phasen außer **RED**, **GREEN** und **REFACTOR**

5 Testen der Anwendung

Sie müssen Ihr Programm nicht testgetrieben entwickeln, es soll aber ausreichend mit Unit-Tests abgedeckt werden. Es ist schwierig graphische Oberflächen (graphical user interface, GUI) zu testen, achten Sie daher auf eine gute Trennung zwischen der GUI und der Logik des Programms. Es empfiehlt sich um eine bessere Testbarkeit zu erzielen den Code der GUI sehr simpel zu halten, d.h., zerlegen Sie Ihre Anwendung in Schichten und halten Sie die GUI-Schicht sehr dünn.

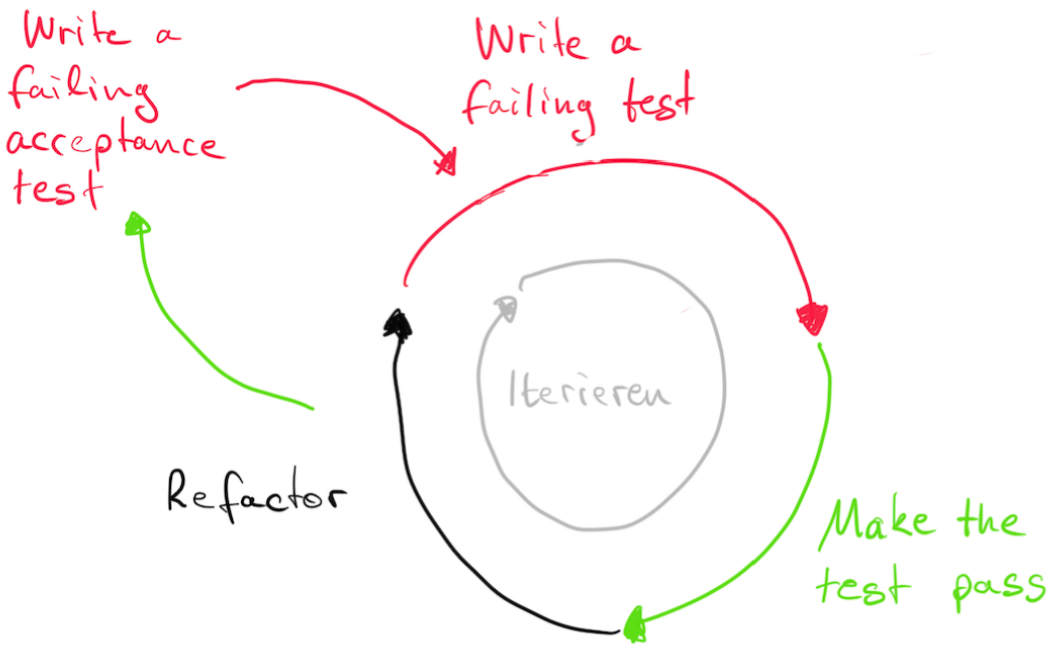

Abbildung 2: Entwickungsprozess: ATDD

6 Bauen der Anwendung

Wir möchten das Bauen der Anwendung einheitlich haben, daher müssen Sie den Quellcode Ihrer Anwendung in Github verwalten. Sie sollten nach Möglichkeit häufig pushen, da die Korrektoren in regelmäßigen Abständen Ihren Fortschritt anschauen werden. Die Anwendung muss mit Hilfe eines Gradle-Skriptes gebaut werden. Wir empfehlen die Verrwendung des Gradle Application Plugins. Es muss eine Datei README.md eingecheckt werden, in der beschrieben ist, welcher Gradle Task zum Bauen verwendet wird und wie das Programm ausgeführt wird. Ihr Programm muss mit Hilfe von Travis CI automatisch bei jedem Push nach Github gebaut und getestet werden.

7 Lizenz

Sie müssen Ihre Anwendung mit einer Softwarelizenz versehen. Sie haben die Auswahl zwischen der Eclipse Public License 1.0, Apache License 2.0 und der MIT-Lizenz.

8 Die Bibliothek

Die Bibliothek gibt Ihnen einfachen Zugriff auf den Java Compiler und den JUnit Testrunner. Sie können mit Hilfe der Bibliothek Java Code, den Sie in Form von Strings vorliegen haben, kompilieren und Testfälle laufen lassen. Der Sourcecode ist auf Github (`https://github.com/bendisposto/virtual-kata-lib`) zu finden.
Die Schnittstelle ist im Sourcecode dokumentiert und es gibt Tests.
Die Javadoc können Sie unter `http://kata.bendisposto.de` online lesen. Die Tests sind im Paket `vk.core.api` im Ordner `src/test/java`, der Sourcecode ist im Paket `vk.core.api` im Ordner `src/main/java`. Der Einstiegspunkt ist die Klasse `CompilerFactory`. Die Implementierung ist im Paket `vk.core.internal` im Ordner `src/main/java`, Sie sollten aber eigentlich nicht dort hineinsehen müssen, um die Funktionsweise zu verstehen.

Sie müssen die Bibliothek als Build-Dependency in Ihrem Projekt verwenden. Die Group-ID ist `de.hhu.stups`, die Artifact-ID ist `virtual-kata-lib`. Als Versionsnummer verwenden Sie bitte die neuste Version, die Sie in Maven finden können.

9 Offene Punkte

Die Problembeschreibung ist möglicherweise unvollständig, ungenau oder widersprüchlich. Sie können Fragen, die nicht aus der Beschreibung heraus beantwortet sind, selber beantworten (d.h., Sie dürfen eine Antwort erfinden). Dokumentieren Sie aber in Ihrem Bericht welche Frage Sie hatten und wie diese beantwortet wurde, damit wir die Entscheidung nachvollziehen können.

10 Teambildung

Das Projekt muss in einer Gruppe von 4 ($\pm$ 1) Studierenden durchgeführt werden. Die Entwicklung findet in Github statt, die Team-Anmeldung erfolgt wie in den Übungen in Github Classroom. Der Link, unter dem Sie ihre Teams registrieren ist `https://classroom.github.com/group-assignment-invitations/115ea4e320d618894c24fe61a2e58053`. Bitte treten Sie einem Team nur bei, wenn Sie das mit den anderen Mitgliedern abgesprochen haben.
Wenn Sie noch keine Partner gefunden haben, können Sie im Ilias Organisationsforum nach weiteren Gruppenmitgliedern suchen. Sie müssen sich mindestens einmal pro Woche als Gruppe treffen (wann und wo bleibt Ihnen überlassen). Bei diesem Treffen sollen Sie gemeinsam die kommende Woche planen (Ziele setzen für jedes Mitglied, etc.) und die letzte Woche reviewen (Wurden die Ziele erreicht? Welche Ziele wurden verfehlt? Warum gab es Probleme? Wie können die Probleme beim nächsten Mal vermieden werden? ...). Die Ergebnisse dieser Treffen sind in einem Protokoll festzuhalten. Die Protokolle müssen wie der Code in Github gepusht werden.

11 Abgabeformat, Modus und Bewertung

Für diese Aufgabe haben Sie vier Wochen Zeit. Wir werden die Version Ihrer Software werten, die am 15.7.2016 um 11:45 Uhr in Ihrem Github Projekt gepusht ist. Es wird für diese Aufgabe kein Review durch Ihre Kommilitonen geben.

A Beispielkonfiguration

So könnte eine Katalogkonfiguration aussehen für zwei Übungen (der Unterschied ist die Babystep Konfiguration). Sie können sich aber vollkommen frei für ein anderes Format entscheiden, XML ist nicht verpflichtend.

```
<excercises>
  <exercise name="Römische Zahlen">
    <description>Konvertiert arabische in römische Zahlen.</description>
    <classes>
      <class name="RomanNumberConverter">
public class RomanNumberConverter {

}
      </class>
    </classes>
    <tests>
       <test name="RomanNumberConverterTest">
import static org.junit.Assert.*;
import org.junit.Test;

public class RomanNumbersTest {

   @Test
   public void testSomething() {

   }
}
       </test>
    </tests>
    <config>
      <babysteps value="False" />
      <timetracking value="True" />
    </config>
  </exercise>
  <exercise name="Römische Zahlen">
    <description>Konvertiert arabische in römische Zahlen.</description>
    <classes>
      <class name="RomanNumberConverter">
public class RomanNumberConverter {

}
      </class>
    </classes>
    <tests>
       <test name="RomanNumberConverterTest">
import static org.junit.Assert.*;
import org.junit.Test;

public class RomanNumbersTest {
```

```
   @Test
   public void testSomething() {

   }
}
      </test>
    </tests>
    <config>
      <babysteps value="True" time="2:00" />
      <timetracking value="True" />
    </config>
  </exercise>
</excercises>
```