Hilfe:

- Dash (Mac) oder Zeal (Windows,Linux) nutzen.
- bei der schriftlichen Matura gibt es auch kein Internet.

Abgabe:

Repository: `sew4_sem1j` – Module `UE01_Junit` – Package `firsttry`

A Einführung

A.1 Aufgabe: Bezahlung

A.1.1 Idee

Um eine gerechte Entlohnung zu erreichen, werden Berichte in einer Redaktion nach Wörtern bezahlt. Zur automatischen Abrechnung soll die Anzahl der geschriebenen Wörter automatisch bestimmt werden. Ihre Aufgabe ist, das Unterprogramm zum Zählen der Wörter in einem Text zu schreiben.

```
count("eins")             // ==> 1
count("eins ")            // ==> 1
count("ein erster Text") // ==> 3
```

A.1.2 Oops

Einige Kollegen versuchen, mit fiesen HTML Code die Zählung auszutricksen, und mittels HTML das große Geld zu machen. Daher muss das Wörterzählen angepasst werden.

```
count(" eins  <html> ")  // ==> 1
```

A.1.3 Todo

Schreibe ein Unterprogramm zum Zählen der Wörter in einem Text. Wörter bestehen aus Buchstaben (A-Z, a-z)[1], alle anderen Zeichen sind kein Bestandteil eines Wortes.

In der komplizierteren Fassung enthält der Text HTML Formatierungen. Die Wörter innerhalb von Tags sollen dabei nicht gezählt werden.

Die Testfälle (siehe beigefügte Datei) sind vorgegeben und müssen in der vorgegebenen Reihenfolge als JUnit-Tests implementiert und ausgeführt werden.

- die Testfälle *einzeln nacheinander* einfügen und immer *alle* bereits eingefügten Testfälle testen
- ein `git commit` **spätestens** nach jedem "Block" = Leerzeile

A.1.4 assert

`assert  expr`

- Macht nichts, wenn die Bedingung erfüllt ist.
- Sonst wird eine Exception geworfen.

```
assert a > b; // wirft eine Exception, wenn a <= b ist
assert a > b : "ein Problem"; // mit eigener Fehlermeldung
```

nicht vergessen: VM-Options `-ea` setzen, siehe

- IntelliJ
  – Run / Edit Configurations / VM Options

[1] oder auch `Character.isLetter()`

A.1.5 Theorie

Konzept: TDD = Test Driven Development

- zuerst neuen, zusätzlichen Test schreiben – funktioniert (meist) nicht ;-(FAIL
- dann zum Laufen bringen – OK
- eventuell verschönern = refactoring, alle Test bleiben OK

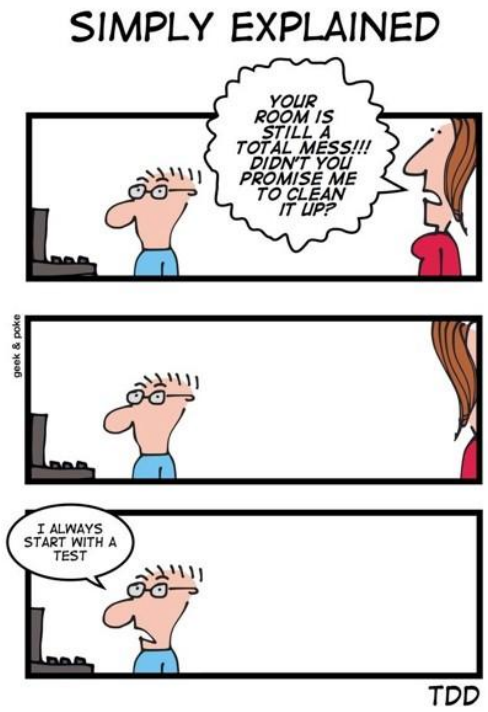

Abbildung 1: Bild von http://www.datamation.com/news/tech-comics-dont-worry-its-just-a-geek-2.html

A.1.6 Tests

Tipp: gute Editoren können “replace regular expression”

```
<!-- ##Tests## -->
  // leicht
  count("") == 0
  count(" ") == 0
  count("  ") == 0

  // normal
  count("one") == 1
  count(" one") == 1
  count("one ") == 1
  count(" one ") == 1
  count(" one  ") == 1
  count("  one ") == 1
  count("  one  ") == 1

  count("one:") == 1
  count(":one") == 1
  count(":one:") == 1
  count(" one  ") == 1
  count(" one : ") == 1
  count(": one :") == 1
  count("ein erster Text") == 3
  count(" ein  erster   Text      ") == 3
  count("ein:erster.Text") == 3

  // vielleicht falsch
  count("a") == 1
  count(" a") == 1
  count("a ") == 1
  count(" a ") == 1

  // mit html
  count(" one  <html> ") == 1
  count(" one  < html> ") == 1
  count(" one  <html > ") == 1
  count(" one  < html > ") == 1
  count(" one <html> two<html>three <html> four") == 4

  count(" one <html> two ") == 2
  count(" one <html>two ") == 2
  count(" one<html> two ") == 2
  count(" one<html>two ") == 2
  count(" one<img alt=\"xxx\" > two") == 2
  count(" one<img alt=\"xxx yyy\" > two") == 2
```

```
  count(" one \"two\" ") == 2
  count(" one\"two\" ") == 2
  count(" one \"two\"") == 2
  count(" one \"two\"three") == 3
  count(" one \"two\" three") == 3

  // html - trickreich
  // Achtung: das ist teilweise nicht ganz legales HTML
  count(" one<html") == 1 // kein >

  count(" one<img alt=\"<bild>\" > two") == 2 // <> innerhalb ""
  count(" one<img alt=\"bild>\" > two") == 2 // <> innerhalb ""
  count(" one<img alt=\"<bild>\" keinwort> two") == 2
  count(" one<img alt=\"<bild>\" src=\"bild.png\" >two") == 2
  count(" one<img alt=\"<bild\" keinwort>two") == 2

  count(" one<img alt=\"<bild\" keinwort") == 1
  count(" one<img alt=\"<bild\" keinwort> two") == 2
  count(" one<img alt=\"<bild keinwort> keinwort") == 1
  count(" one<img alt=\"<bild keinwort keinwort\">two") == 2
  count(" one<img alt=\"<bild keinwort< keinwort\">two") == 2

  // ganz ganz fies -- \ entwertet das nächste Zeichen
  count(" one<img alt=\"<bild \\\" keinwort> keinwort\" keinwort>two") == 2
  count(" one<img alt=\"<bild \\\" keinwort<keinwort\" keinwort>two") == 2
  count(" one<img alt=\"<bild \\\" keinwort keinwort\" keinwort>two") == 2

  count(" \\\"null\\\" one<img alt=\"<bild \\\" keinwort keinwort\" keinwort>two \"three\"") == 4
<!-- ##ENDE## -->
```

B Einfache Tests -– Test Driven Development (TDD)

B.1 Framework JUnit

Idee: eigene Datei mit den Tests, *bessere* Methoden zum Test, Framework enthält *Runner* (`main()`).

Umbauen auf :

```
    @Test
    public void testEmptyString() {
        assertEquals(0, WordCount.count("")); // Reihenfolge: expected, actual
        // oder mit besserer Fehlermeldung
        assertEquals("leerer String", 0, WordCount.count(""));
    }
```

IntelliJ (https://www.jetbrains.com/help/idea/creating-tests.html):

- Alt-Insert bzw RK-Generate / Test…
- Alt-Enter am Wort `@Test` sollte für das Einbinden von `JUnit5` reichen.
- sonst mit `import static org.junit.api.Assert.*;` bzw. `@org.junit.api.Test`.
- oder mit der zu testenden Klasse beginnen und
  - Alt-Enter bzw.
  - Ctrl-Shift-T

- Run As: JUnitTest
  - `Fail` – roter Balken

→ Testframework ist OK!

Damit es *compilierbar* und fehlerfrei ("grünes Häkchen in IntelliJ") wird: IntelliJ *hilft*

- Klasse erzeugen – kann IDE
- Methode erzeugen – kann IDE
- sollte `int` liefern, vorerst 0

→ OK, grüner Balken: ein Testfall + ein Interface festgelegt

Jetzt ist ein guter Moment für

- `git commit` – immer wenn etwas funktioniert (im Bild bei *yes*): sichern
- Refactoring: Code "schöner" machen – bei Bedarf
- `git commit`
- Nächster Testfall…

B.1.1 TDD

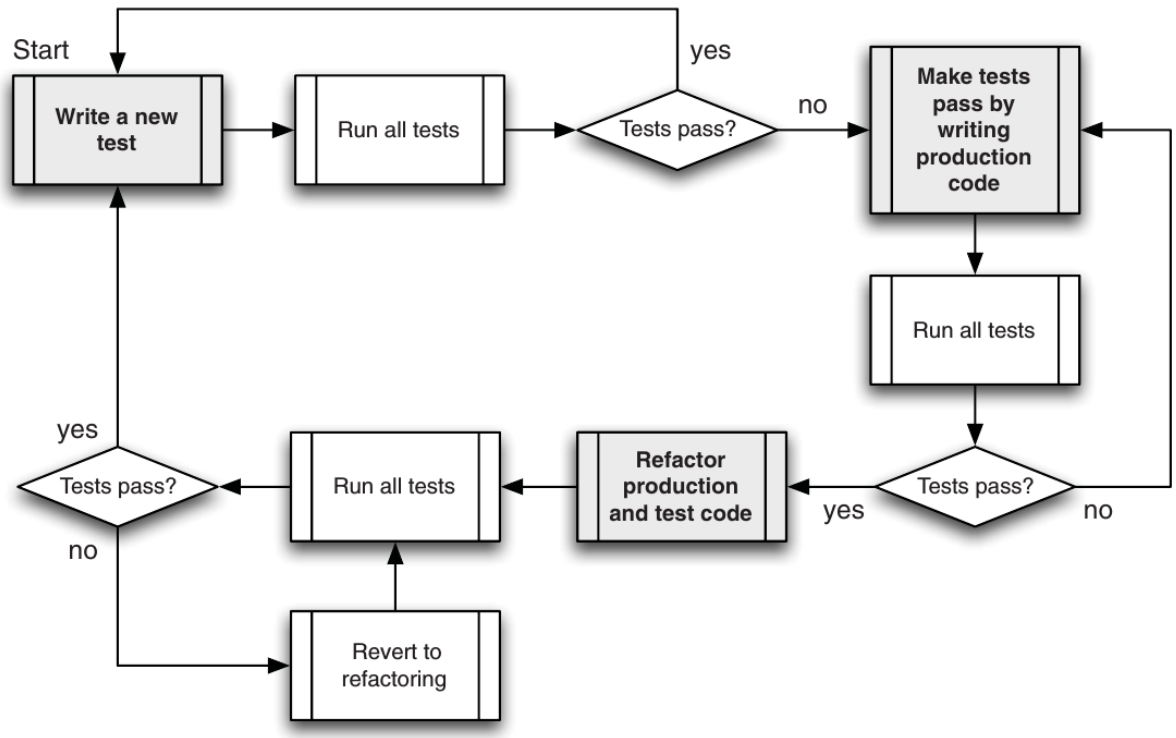

Abbildung 2: TDD, Bild aus: Effective Unit Testing, A guide for Java developers, Lasse Koskela

Ziel ist der grüne Balken, so schnell wie möglich

- keine/wenig *Planung* – KISS (Keep it stupid simple bzw. Keep it simple, stupid!)
- keine *könnte man brauchen* Features – YAGNI (You Aren't Gonna Need It)
- aber DRY (Don't repeat yourself) – Refactoring

B.1.2 Hinweis

Alle Tests haben ein `@Test` (*Annotation*), es gibt viele Varianten von `assert...`:

- erster Paramter: `expected`,
- zweiter Parameter `actual` (wichtig für Fehlermeldung)
- zusätzliche Variante mit einem String `messsage` als ersten Parameter
- auch für Arrays

B.2 Erweiterte Testmöglichkeiten

B.2.1 lesbare Tests: `hamcrest`

```
import org.junit.Ignore;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

@Test
public void testEmptyString() {
    // https://code.google.com/p/hamcrest/wiki/Tutorial
    assertThat(WordCount.count(""), is(0));
    assertThat(WordCount.count(""), is(not(3)));
    assertThat(WordCount.count(""), equalTo(0));
    assertThat(WordCount.count(""), is(equalTo(0)));
}
```

B.2.2 Test auf Exception

```
@Test(expected= IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}

@Test
public void empty2() {
    try {
        new ArrayList<Object>().get(0);
        fail("sollte Exception werfen");
    } catch (IndexOutOfBoundsException e) {
        // Exception genauer untersuchen
        assertThat(e.toString(), containsString("Index: 0"));
    }
}
```

B.2.3 Test mit *Timeout*

```
@Test
@Timeout(value = 42, unit = TimeUnit.MILLISECONDS)
public void testWithTimeout() {
    // aufwändiger Test
}
```

B.2.4 Test kurzfristig ausschalten

```
@Disabled("Test is ignored as a demonstration")
@Test
public void testSane() {
    assertThat(1, is(1));
}
```

B.2.5 Viele ähnliche Tests

Nachteil: eventuell unübersichtliche Anzeige

https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-parameterized-tests/

B.2.6 Tests in mehreren Java-Dateien

Man kann einen Gesamttest definieren: Junit `test suite` anlegen – alle Test zusammenfassen siehe https://howtodoinjava.com/junit5/junit5-test-suites-examples/

B.2.7 Test vor- und Nachbereitung

Weitere `@...` erlauben das Vorbereiten (zB. Objekte anlegen) und das Nachbereiten (zB. Objekte löschen).

Reichenfolge

- `@BeforeClass` : one time setUp
  - `@Before` : setup (for each test)
    - `@Test` : first test
  - `@After` : tear down
  - `@Before` : setup
    - `@Test` : next Test
  - `@After` : tear down
  - …
- `@AfterClass` : one time tear down

B.2.8 Wieviele Tests?

Ziel: Sicherheit = ich vertraue diesem Code

- viele Tests
- schnelle Tests
- Abdeckung (Coverage): möglichst hoch
  - 100 % sind nicht immer erreichbar.
  - Andererseits: Codezeilen die man nie verwendet sind per Definition *falsch*.
- Unterschied
  - Black-Box-Test = Tests ohne Kenntnis des Codes
  - Glas-Box-Test, White-Box-Test = Test mit Wissen um den Code (zB. Welche Fragen werden bei einem `if` gestellt)
- Grenzfälle (Corner Cases):
  - leere Texte.
  - Liste: leer, ein Element, viele Elemente.
  - Werte *die in einem* `if/while` *vorkommen*
  - Wenn x ein Sonderfall ist: x-1, x und x+1 (ebenfalls) testen.
- *schwierig*
  - GUI, Web
    - braucht man pixel-genaue Vergleiche?
    - oder nur zB. Text kommt auf Seite vor
  - Datenbanken

C Tipp

Tipp für alle, die noch keine Lösung für das Wörterzählen haben:

- Variablen `inWort` (`boolean`) und `counter`(`int`)
- Alle Zeichen durchgehen:
  - Wann ändert sich der Wert dieser Variablen?
  - Es gibt zwei Möglichkeiten für das Zählen: am Wortanfang oder am Wortende, …

D Programmierrichtlinien

Ab sofort gilt für jedes Programmierprojekt (sonst gibt es Punkteabzug):

- Javadoc (oder PyDoc – das Gegenstück in Python) – wie bisher
- TDD (Unit-Tests) jede Methode/Funktionalität wird mit (mindestens!) einem Testcase abgedeckt – Die Testfälle werden **vor** der Implementierung der eigentlichen Funktionalität programmiert.
- VCS (git) – nach jedem sinnvollen Programmierschritt ein commit mit sprechendem Namen[2].
- guter Code hat viele Kommentarte

[2] dh. Punkteabzug, wenn ein Commit z.B. *erster commit* oder *mir fällt gerade nix ein* heißt