Programmierung 3

im Wintersemester 2024/25

Praktikumsaufgabe 5

Zuverlässiger Code durch TDD und Unit-Testing

12. November 2024

In diesem Arbeitsblatt entwickeln Sie Unit-Tests mit JUnit und wenden Test-Driven Development (TDD) an.

In Aufgabe 1 setzen Sie JUnit 5 in einem Java-Projekt ein um Unit-Tests zu entwickeln und auszuführen, so dass Sie die Funktionalität einer einfachen Klasse testen und zielgerichtet reparieren können.

In Aufgabe 2 wenden Sie TDD an, um eine neue Funktionalität testgetrieben zu entwickeln. Dabei gehen Sie strukturiert in möglichst kleinen Schritten vor, um mit einer möglichst geschickten Reihenfolge der Tests unnötige Komplexität und Mehrarbeit zu vermeiden.

Heads up! Lesen Sie die einzelnen Aufgaben vollständig durch, *bevor* Sie mit der jeweiligen Bearbeitung beginnen. So erhalten Sie einen Überblick und können die Aufgaben effizienter bearbeiten. Die Aufgaben enthalten Hinweise und Tipps – wie diesen – die Ihnen bei der Bearbeitung helfen.

Arbeiten Sie bei der Lösung der Aufgaben in einem Team von 3 bis 4 Personen zusammen, so dass Sie die Aufgaben kollaborativ lösen und Ihre Ergebnisse untereinander diskutieren können. Lösen Sie dabei die Programmieraufgaben im Pair Programming und bearbeiten Sie die anderen Aufgaben in einer für Sie geeigneten Weise.

1 Unit-Testing mit JUnit

Lernziele

- ☐ Sie strukturieren ein Java-Projekt so, dass Produktions- und Testcode im Besonderen und Build-Quellen und Build-Ergebnisse im Allgemeinen getrennt sind.
- ☐ Sie verwenden JUnit 5 auf der Konsole, um Unit-Tests zu entwickeln und auszuführen.
- ☐ Sie schreiben einfache Unit-Tests mit JUnit 5, um Fehler in bestehendem Code aufzudecken und zu beheben.
- ☐ Sie verwenden die Annotationen `@Test` und `@DisplayName`, um Tests zu definieren und ihnen eine aussagekräftige Bezeichnung zu geben.
- ☐ Sie verwenden die Assertions `assertEquals` zur Überprüfung von Werten und `assertThrows` zur Überprüfung von Ausnahmen.[a]

[a] Daneben gibt es noch viele weitere Assertions, die Sie verwenden können, wie `assertTrue`, `assertNull` und andere.

In dieser Aufgabe entwickeln Sie Unit-Tests mit JUnit 5. JUnit ist ein Framework für das Schreiben und Ausführen von automatisierten Tests in Java. Es ist das am weitesten verbreitete Framework für das Testen von Java-Anwendungen und wird in vielen Projekten eingesetzt.

Als Vorbereitung auf das Thema *Projektaufbau mit Maven* richten Sie eine für Java-Projekte typische Verzeichnisstruktur ein, die üblichen Konventionen folgt.

Wichtig: Auch wenn Sie vielleicht bereits wissen, wie Sie JUnit in Ihrer IDE verwenden, ist es wichtig, dass Sie auch wissen, wie Sie ein Java-Projekt auf der Konsole bauen und testen. Dies ist eine wichtige Fähigkeit, die Ihnen helfen wird, die Funktionsweise von Build-Tools wie Maven zu verstehen sowie automatisierte Workflows für Continuous Integration zu entwickeln.

1.1 Vorbereitungen

Damit Sie JUnit 5 auf der Konsole nutzen können, sind einige Vorbereitungen notwendig.

1.1.1 Verzeichnisstruktur

Legen Sie folgende Verzeichnisstruktur an:

`src/main/java` Für die "echten" Klassen, die den Produktionscode enthalten.

`src/test/java` Für die Testklassen, die die "echten" Klassen testen.

`lib` Für die JAR-Datei, die Sie für JUnit 5 benötigen.

`target/classes` Für die kompilierten "echten" Klassen.

`target/test-classes` Für die kompilierten Testklassen.

Heads up! In der Java-Entwicklung werden typischerweise "echte" Klassen von Testklassen getrennt, um die Übersichtlichkeit und Wartbarkeit des Codes zu erhöhen. Die Organisation der Tests in einem separaten Verzeichnis erleichtert die Testausführung und -verwaltung und fördert die Trennung von Produktions- und Testcode.

Die Verzeichnisstruktur oben erlaubt darüber hinaus die saubere Trennung von Quellen des Projekts (in unserem Fall die "echten" Klassen und die Testklassen in `src`), das Ergebnis des Bauens des Projekts (die kompilierten Klassen in `target`) sowie die Bibliotheken, die für das Projekt benötigt werden (in `lib`).

1.1.2 JUnit 5 JAR-Datei

Laden Sie die aktuelle Version der JAR-Datei *JUnit Platform Console Standalone* für JUnit 5 herunter und speichern Sie sie im Verzeichnis `lib`. Im Moment ist dies Version `1.11.3` vom 21. Oktober 2024. Diese JAR-Datei enthält die notwendigen Klassen, um Tests für JUnit 5 zu schreiben und auszuführen.

Mit folgendem Befehl können Sie ausprobieren, ob Sie JUnit 5 erfolgreich installiert haben:

```
java -jar lib/junit-platform-console-standalone-1.11.3.jar
 ↪  execute --class-path target/classes --class-path
 ↪  target/test-classes --scan-class-path
```

Danach sollten Sie eine Ausgabe erhalten, dass 0 Tests gefunden wurden und dementsprechend keine Tests ausgeführt wurden.

1.2 Die zu testende und zu fixende Klasse

Platzieren Sie folgende Klasse im Verzeichnis `src/main/java`:

```java
public class SimpleCalculator {
  public int add(int a, int b) {
    return a - b;
  }

  public int subtract(int a, int b) {
    return a + b;
  }

  public int divide(int a, int b) {
    if (b > 100) {
      throw new IllegalArgumentException("Division by zero is not
       ↪  allowed.");
    }
    return a * b;
  }
}
```

Im Folgenden werden Sie diese Klasse mit Hilfe von JUnit 5 testen und im Laufe dessen die (wahrscheinlich offensichtlichen) Fehler korrigieren.

Mit folgendem Befehl können Sie die Klasse `SimpleCalculator` kompilieren:

```
javac -d target/classes src/main/java/SimpleCalculator.java
```

1.3 Unit-Tests entwickeln und den Produktionscode fixen

Implementieren Sie die weiter unten folgenden Unit-Tests, indem Sie die passenden mit `@Test` annotierten Methoden in der Klasse `SimpleCalculatorTest` schreiben. Platzieren Sie den Quellcode der Testklasse im Verzeichnis `src/test/java`.

Mit folgendem Befehl können Sie die Klasse `SimpleCalculatorTest` kompilieren:

```
javac -cp
 ↪  target/classes:lib/junit-platform-console-standalone-1.11.3.jar
 ↪ -d target/test-classes
 ↪  src/test/java/SimpleCalculatorTest.java
```

Mit dem Befehl von oben können Sie Ihre Tests, die sie gleich implementieren werden, ausführen.

Da die einzelnen Methoden in der Klasse `SimpleCalculator` fehlerhaft sind, sollten Ihre Tests jeweils fehlschlagen. Korrigieren Sie die Methoden in der Klasse `SimpleCalculator` so, dass Ihre Tests erfolgreich sind.

1.3.1 Aussagekräftige Namen und Bezeichnungen

Wählen Sie für die Testmethoden aussagekräftige Namen, die beschreiben, was getestet wird. Dies erleichtert das Verständnis des Codes und die Wartung. Ein guter Ansatz ist, den Namen so zu formulieren, dass er das Szenario und das Ergebnis widerspiegelt. Eine übliche Struktur sieht wie folgt aus:

`methodBeingTested_condition_expectedResult`.

Verwenden Sie darüber hinaus die Annotation `@DisplayName`, um beschreibende Namen für die Testmethoden zu definieren. Dies erhöht die Lesbarkeit der Testausgabe, was Sie auf der Konsole leicht nachvollziehen können.

Hier ein Beispiel für die Methode `add`:

```java
@DisplayName("Adding 2 and 3 should equal 5")
void add_givenTwoAndThree_shouldReturnFive() {
  // Testcode
}
```

1.3.2 Arrange-Act-Assert

Der Arrange-Act-Assert-Ansatz ist eine einfache Struktur für das Schreiben von Unit-Tests, die den Testablauf klar und nachvollziehbar macht. Insbesondere beim Erlernen des Schreibens von Unit Tests ist es hilfreich, dieser Struktur zu folgen:

Arrange: In diesem Schritt werden alle notwendigen Voraussetzungen für den Test geschaffen, wie z.B. die Erstellung der benötigten Objekte oder die Deklaration von Variablen mit Eingabewerten.

Act: Hier wird die zu testende Aktion ausgeführt. Typischerweise ist dies ein Methodenaufruf. Dabei können die zuvor erstellten Objekte oder definierten Eingabewerte verwendet werden.

Assert: Abschließend wird das Ergebnis durch Vergleich mit der erwarteten Ausgabe verifiziert, um sicherzustellen, dass die getestete Methode wie erwartet funktioniert.

Hier ein Beispiel anhand der Methode `add`:

```java
// Arrange
final int a = 2;
final int b = 3;

// Act
final int sum = calculator.add(a, b);

// Assert
assertEquals(5, sum);
```

Dieser Ansatz zur Strukturierung der einzelnen Testmethoden hilft, die Testlogik übersichtlich zu halten und erleichtert das Verständnis des Codes.

1.3.3 Die Unit-Tests

Implementieren Sie die folgenden Unit-Tests und fixen Sie dabei jeweils anschließend die einzelnen Methoden in der Klasse `SimpleCalculator`, so dass der zuvor von Ihnen geschriebene Test erfolgreich bestanden wird.

Heads up! Ziel ist es, dass Sie *zuerst* einen fehlschlagenden Test ("red") schreiben und erst *danach* den Produktionscode so anpassen, dass der Test erfolgreich ist ("green").[1]

Test 1: Addition Mit der Methode `add` sollen zwei Zahlen richtig addiert werden. Schreiben Sie einen Test, um dies zu überprüfen. Ihr Test könnte z.B. überprüfen, ob die Addition von 2 und 3 5 ergibt.

Test 2: Subtraktion Mit der Methode `subtract` sollen zwei Zahlen richtig subtrahiert werden. Schreiben Sie einen Test, um dies zu überprüfen. Ihr Test könnte z.B. überprüfen, ob die Subtraktion von 3 von 5 korrekt 2 ergibt.

Tests 3 und 4: Division

Division durch erlaubte Werte: Die Methode `divide` soll zwei Zahlen richtig teilen, solange der Divisor nicht 0 ist. Schreiben Sie einen Test, der dies überprüft. Ihr Test könnte z.B. prüfen, ob die Division von 6 durch 3 korrekt 2 ergibt.

Division durch 0: Die Methode `divide` soll eine `IllegalArgumentException` werfen, wenn der Divisor 0 ist. Schreiben Sie einen Test, um dies zu überprüfen.

[1] Wäre man beim Schreiben der Klasse `SimpleCalculator` ähnlich testgetrieben vorgegangen, wären die Fehler höchstwahrscheinlich gar nicht erst entstanden.

2 Test-Driven Development

Lernziele

- ☐ Sie entwickeln eine neue Funktionalität testgetrieben.
- ☐ Sie wenden eine strukturierte Zerlegungsstrategie an, um dabei in möglichst kleinen Schritten vorzugehen.

Entwickeln Sie testgetrieben eine Methode, die die Anzahl der Vokale[2] in einer gegebenen Zeichenfolge zählt. Folgen Sie dabei dem Test-Driven-Development-Zyklus:

Think: Überlegen Sie sich vorab, welche Tests Sie schreiben müssen, um die Anforderungen an die Methode zu spezifizieren und in Form von automatisierten Unit-Tests zu überprüfen.

Red: Implementieren Sie *einen* Test, der zunächst fehlschlägt.

Green: Implementieren Sie eine *Minimallösung*, um den neuen Test zum Bestehen zu bringen. Und so, dass die bereits bestehenden Tests weiterhin erfolgreich sind.

Refactor: Machen Sie den Code jetzt *clean*, ohne die Funktionalität zu verändern. Und so, dass alle Tests weiterhin erfolgreich sind.

Heads up! Nutzen Sie dabei die in der Vorlesung vorgestellte Strategie, um in *möglichst kleinen* Schritten vorzugehen.

Im Folgenden finden Sie eine iterative und inkrementelle Anleitung, die Sie anhand vorgeschlagener **Think**-Schritte durch den Prozess führt. Sehen Sie diese Anleitung als Beispiel, wie Sie vorgehen können. Wenn Sie noch kleinere Schritte gehen möchten, umso besser!

[2] Sie werden auch als Selbstlaute bezeichnet und sind im Deutschen durch die Buchstaben `a`, `e`, `i`, `o` und `u` sowie deren Umlaute `ä`, `ö` und `ü` repräsentiert

2.1 Kernschnittstelle

Think: Fokus ist das Design der Signatur der Methode, die Sie als geeignet erachten, um die Anzahl der Vokale in einer Zeichenfolge zu zählen. Was sind die benötigten Parameter? Was ist der Rückgabetyp?

Tipp: Der einfachste Fall ist eine Methode, die zunächst auf einem einzelnen Buchstaben arbeitet, also einen `char` als Parameter erhält.

Red: Im ersten Schritt schreiben Sie einen Test, der die Methode auf möglichst einfache Weise aufruft.[3]

Green: Damit der Test dann erfolgreich ist, können Sie die Antwort einfach fest codieren. Zum Beispiel können Sie die Methode so implementieren, dass sie immer 0 zurückgibt.

Refactor: Passt die Signatur? Passt der Name der Methode?

2.2 Berechnungen und Verzweigungen

2.2.1 Berechnung

Think: Die einfachste Berechnung ist die Überprüfung auf einen einzelnen Vokal.

Red: Schreiben Sie einen Test, der überprüft, ob die übergebene Zeichenfolge den Vokal `a` enthält. Wird der Methode `a` übergeben, sollte sie `1` zurückgeben. Wird ihr zum Beispiel `b` übergeben, sollte sie `0` zurückgeben.

Green: Implementieren Sie die Methode so, dass sie diesen Test besteht.

Refactor: Wie können Sie Ihren Code an dieser Stelle auf eine angemessene Art noch cleaner machen?

[3] In diesem ersten Schritt brauchen Sie also auch keine Assertion.

2.2.2 Verzweigungen

Gehen Sie hier *schrittweise* vor, also zunächst nur für einen weiteren Vokal, dann nach und nach für alle:

Think: Als nächstes Erweitern Sie die Methode so, dass alle Vokale berücksichtigt werden.[4]

Red: Schreiben Sie einen Test, der einen weiteren Vokal überprüft. Es sollte entsprechend wieder `1` oder `0` zurückgegeben wird.

Green: Erweitern Sie die Methode so, dass sie den jeweiligen Test besteht.

Refactor: Wie können Sie Ihren Code an dieser Stelle auf eine angemessene Art noch cleaner machen? Zum Beispiel bietet sich beim zweiten Vokal eine einfache Fallunterscheidung an. Bei mehreren Vokalen könnte es sinnvoll werden, eine alternative Lösung zu finden.

2.3 Schleifen und Verallgemeinerung

Think: Als nächstes Erweitern Sie die Methode so, dass sie alle Vokale in einer Zeichenfolge berücksichtigt, die länger als ein Zeichen ist.[5]

Wählen Sie geeignete, möglichst kleine Schritte, um sich dem Ziel iterativ und inkrementell zu nähern. Führen Sie den Zyklus für jeden Schritt einzeln durch.

2.4 Sonderfälle und Fehlerbehandlung

Think: Welche Sonderfälle gibt es? Was ist, wenn die Zeichenfolge leer ist? Was ist, wenn sie nur aus Leerzeichen besteht? Kann dann z.B. mittels `String.trim()`, `String.strip()` oder ähnlichem optimiert werden? Was, wenn **`null`** übergeben wird? Sollte dann eine `IllegalArgumentException` geworfen werden?

[4] Der Einfachheit halber können Sie sich auf Kleinbuchstaben beschränken.
[5] Bonus: Was ist mit anderen Sprachen als Deutsch?

Führen Sie den Zyklus für jeden Sonderfall und für jede Fehlerbehandlung *einzeln* durch.