HTBL Imst

AUFBAULEHRGANG FÜR INFORMATIK

Übungszettel –
Test Driven Development

Name: Michael Bogensberger

Datum: 09.03.2022

1 Inhalt

2 Grundbegriffe 3
2.1 Was ist Test Driven Development? 3
2.2 Red-Green-Refactor 3
2.3 Kent Beck 3
2.4 Testverfahren 4
2.4.1 Statische Tests 4
2.4.2 Dynamische Tests 4
2.4.3 White-Box-Tests / Black-Box-Tests 4
2.5 Testarten 4
2.5.1 Unit-Tests 4
2.5.2 Integrationstests 4
2.5.3 Funktionstests 4
2.5.4 End-to-End-Tests / UI-Tests 4
2.5.5 Akzeptanztests 5
2.6 Testpyramide 5
2.7 JUnit 5
2.8 Mockito 5
3 Einarbeitung in den gegebenen Code (Aufgabe 3) 6
4 JUnit Tests für KinoSaal Klasse (Aufgabe 4) 6
5 JUnit Tests für Vorstellung Klasse (Aufgabe 5) 6
6 JUnit Tests für Kinoverwaltun Klasse (Aufgabe 6) 6
7 JUnit Tests – Advanced (Aufgabe 7) 6
8 Mockito Einführung (Aufgabe 8) 7
9 GitHub Link 7
10 Abbildungsverzeichnis 8

2 Grundbegriffe

In diesem Abschnitt sind die wichtigsten Grundbegriffe zum Thema Test Driven Development zu finden.

2.1 Was ist Test Driven Development?

Durch Test Driven Development wird dafür gesorgt, dass Software gut durchdacht ist, bevor es an die Programmierung des eigentlichen Funktionscodes geht. Man schreibt also zunächst die Tests. Dies wird als **Test-First** bezeichnet und darum ist TDD keine Test-, sondern eine Designstrategie. Wichtig ist auch, dass die Tests nicht zu groß ausfallen. Durch diese Strategie, wird auch unnötiger Code gespart.

2.2 Red-Green-Refactor

Wie man Tests am besten gestalten soll wird unter anderem auch mithilfe des Red-Green-Refactor Modells beschrieben. Mehr Infos unter: TDD-Guide.

- **Red:** Man schreibt einen Test, der Funktionalität bzw. ein Verhalten prüfen soll. Da noch keine oder zumindest nicht ausreichende Funktionalität besteht schlägt dieser fehl.
- **Green:** Ändere den Programmcode mit möglichst wenig Aufwand ab, so dass der Test erfolgreich durchläuft.
- **Refactor:** Nun geht es darum den Test möglichst gut zu optimieren. Entferne Redundanzen und unnötigen Code.

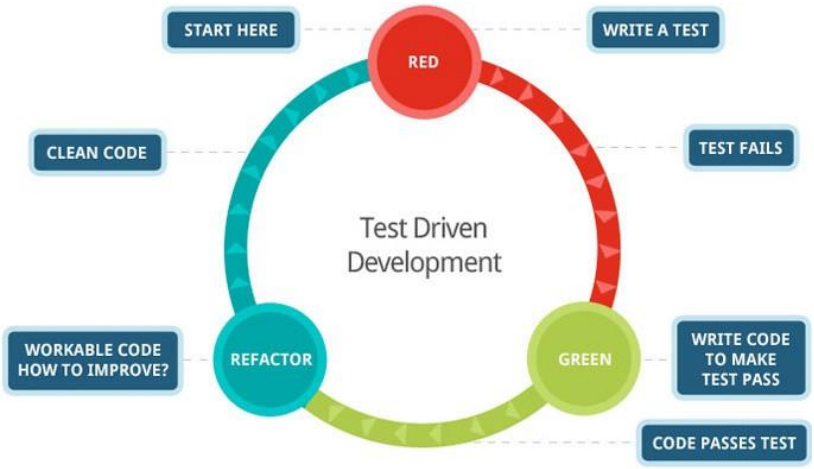

Abbildung 1 Red-Green-Refactor

2.3 Kent Beck

Kent Beck ist ein Softwareentwickler aus Amerika. Er ist der Erfinder des Testframeworks SUnit. Dieses portierte er mit Erich Gamma auf Java und veröffentlichte es als JUnit. Er spielt also eine wichtige Rolle, wenn es um TDD geht. Beim TDD nach Kent Beck geht es darum das Unit-Tests und mit ihnen getestete Units immer parallel entwickelt werden sollen. Die Programmierung dieser soll in kleinen Mikroiterationen erfolgen. Eine Iteration soll nur wenige Minuten andauern und hat drei Hauptteile, die man Red-Green-Refactor nennt.

2.4 Testverfahren

In folgendem Abschnitt wird auf verschiedene Testverfahren genauer eingegangen. Grundsätzlich unterscheiden man zwischen statischen und dynamischen Tests. Mehr dazu unter: Testverfahren-Guide.

2.4.1 Statische Tests

Bei statischen Tests wird der zu testende Code nicht ausgeführt. Man arbeitet hier mit einem Code-Analyse-Tool um die Qualität des Codes zu bemessen.

2.4.2 Dynamische Tests

Bei dynamischen Tests wird das Programm bzw. die Software in der Ausführung getestet. Hier geht es vorrangig um das Verhalten bzw. das Funktionieren der eigentlichen Software.

2.4.3 White-Box-Tests / Black-Box-Tests

Eine andere Art der Klassifizierung von Tests bieten die White-Box und Black-Box Tests.

- Bei einem **White-Box-Test** ist der Zugang zum Quellcode erforderlich um darauf basierend Tests zu spezifizieren. Als Vorgabe für einen Test dient also der Programmcode.
- Bei einem **Black-Box-Test** kennt man den Quellcode nicht. Man behandelt das System also wie eine Blackbox. Also Vorgabe für Tests dient nun also das Verhalten des Programms.

2.5 Testarten

In diesem Abschnitt sind die wichtigsten Testarten aufgelistet. Mehr unter: Testarten-Guide

2.5.1 Unit-Tests

Unit-Tests sind sehr einfach und erfolgen nah an der Quelle der Anwendung. Sie dienen zum Testen einzelner Methoden und Funktionen der von der Software verwendeten Klassen, Komponenten oder Module.

2.5.2 Integrationstests

Mit Integrationstests wird sichergestellt, dass verschiedene von deiner Anwendung genutzte Module oder Services problemlos ineinandergreifen. So kann beispielsweise die Interaktion mit der Datenbank oder das Zusammenspiel von Mikroservices getestet werden.

2.5.3 Funktionstests

Funktionstests konzentrieren sich auf die Geschäftsanforderungen einer Anwendung. Sie verifizieren nur die Ausgabe einer Aktion und überprüfen bei der Durchführung dieser Aktion nicht die Zwischenzustände des Systems.

2.5.4 End-to-End-Tests / UI-Tests

Bei End-to-End-Tests wird der Umgang des Benutzers mit der Software in einer vollständigen Anwendungsumgebung repliziert. Auf diese Weise wird das ordnungsgemäße Funktionieren von Benutzerabläufen überprüft. Die Szenarien können ganz einfach sein (z. B. Laden einer Website, Anmeldevorgang) oder auch sehr komplex (z. B. E-Mail-Benachrichtigungen, Onlinezahlungen).

2.5.5 Akzeptanztests

Akzeptanztests sind formelle Tests, mit denen überprüft wird, ob ein System die entsprechenden geschäftlichen Anforderungen erfüllt. Dazu muss die gesamte Anwendung funktionsfähig sein. Der Schwerpunkt liegt auf dem replizierten Benutzerverhalten. Darüber hinaus können die Tests eingesetzt werden, um die Systemleistung zu messen und Änderungen abzulehnen, wenn bestimmte Ziele nicht erfüllt werden.

2.6 Testpyramide

Die Testpyramide soll zeigen, von welchen Tests man viele bzw. von welchen Tests man wenige benötigt. Demnach sollte man am meisten Unit-Tests haben.

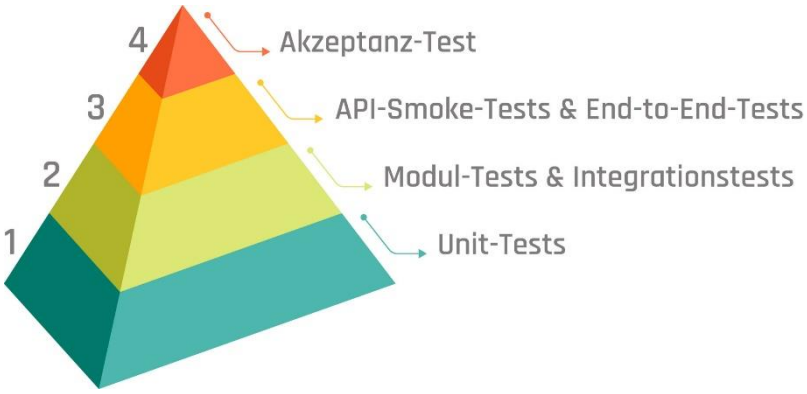

Abbildung 2 Testpyramide

2.7 JUnit

JUnit ist ein Framework zum Testen von Java-Programmen. Es eignet sich vor allem für automatisierte Tests von Units. Units sind in diesem Fall Klassen oder Methoden

2.8 Mockito

Mockito ist eine Programmbibliothek zum Erstellen von Mock-Objekten für Unit-Tests von Java Programmen. Ein Mock-Objekt ist ein Programmteil, der zur Durchführung von Modultests als Platzhalter für echte Objekte verwendet wird. Mit Mockito können also Objekte erzeugt werden, die so handeln als währen sie funktionierender Code. Mockito verwendet man meist, wenn Code getestet werden soll und man sicher gehen möchte, dass wenn ein Fehler auftritt, dieser auch in der getesteten Methode auftritt. Damit kann zum Beispiel eine Schnittstelle simuliert werden. Man kann Mockito aber auch verwenden, wenn man zum Beispiel noch nicht fertigen Code hat.

3 Einarbeitung in den gegebenen Code (Aufgabe 3)

Arbeiten Sie sich in den gegebenen Code zur Kinoverwaltung ein. Verwenden Sie die gegebenen Klassen KinoSaal, Ticket, Vorstellung, Kinoverwaltung in der App-Klasse (main-Methode), um ein Gefühl für die Funktionsweise des Programms zu bekommen. Führen Sie folgende Punkte durch:

- Kinosäle anlegen ✓
- Vorstellungen anlegen ✓
- Vorstellungen über die Kinoverwaltung einplanen ✓
- Tickets für Vorstellungen ausgeben ✓
- etc. ✓

4 JUnit Tests für KinoSaal Klasse (Aufgabe 4)

Testen Sie alle Methoden der Klasse KinoSaal (Testklasse TestKinoSaal). ✓

5 JUnit Tests für Vorstellung Klasse (Aufgabe 5)

Testen Sie alle Methoden der Klasse Vorstellung (Testklasse TestVorstellung). ✓

6 JUnit Tests für Kinoverwaltun Klasse (Aufgabe 6)

Testen Sie alle Methoden der Klasse KinoVerwaltung (Testklasse TestKinoverwaltung). ✓

7 JUnit Tests – Advanced (Aufgabe 7)

Falls nicht schon in den vorhergehenden Aufgaben passiert, testen Sie folgende Punkte unter Verwendung der fortgeschrittenen Features von JUNIT 5:

1. Schreiben Sie einen Test, der validiert, dass das Anlegen einer Vorstellung korrekt funktioniert. Der Test sollte eine fachliche Bezeichnung haben und die Assertions sollten bei Validierungsfehler eine Hinweistext liefern. ✓

2. Schreiben Sie einen Test, der validiert, dass das Einplanen mehrerer Vorstellungen korrekt funktioniert. Stellen Sie zudem sicher, dass beim möglichen Auftreten eines Fehlers trotzdem alle Validierungen ausgeführt werden.

3. Schreiben Sie einen Test, der sicherstellt, dass ein Fehler geworfen wird, wenn eine Veranstaltung doppelt eingeplant wird. ✓

4. Schreiben Sie einen parametrisierten Test, der mehrere Ticketkäufe mit unterschiedlichen Parametern überprüft. ✓

5. Schreiben Sie eine dynamische TestFactory die den Ticketkauf mit zufälligen Werten bombardiert. Der Test soll sicherstellen, dass der Ticketkauf entweder funktioniert oder nur einen der definierten Fehlermeldungen (z.B. new IllegalArgumentException("Nicht ausreichend Geld.")) ausgibt. Die Tests müssen reproduzierbar sein. ✓

8 Mockito Einführung (Aufgabe 8)

Lesen Sie sich in das Mocking-Framework Mockito ein (Links siehe Moodle im Abschitt „Input zu Mockito“). Verwenden Sie die wesentlichen Mockito-Möglichkeiten praktisch in kleinen Programmen. ✓

9 GitHub Link

Unter folgendem Link ist das GitHub Repository zum Übungszettel TDD zu finden: GitHub Repo

10 Abbildungsverzeichnis

Abbildung 1 Red-Green-Refactor 3
Abbildung 2 Testpyramide 5