Softwarequalitätssicherung

Testing

Gruppenmitglieder: Dennis Jongebloed
Julian Hinxlage
Johannes Theiner

Semester: Sommersemester 2020
letzte Änderung: 7. Mai 2020

1 Test-Driven Development

1.1 Entwicklungsablauf

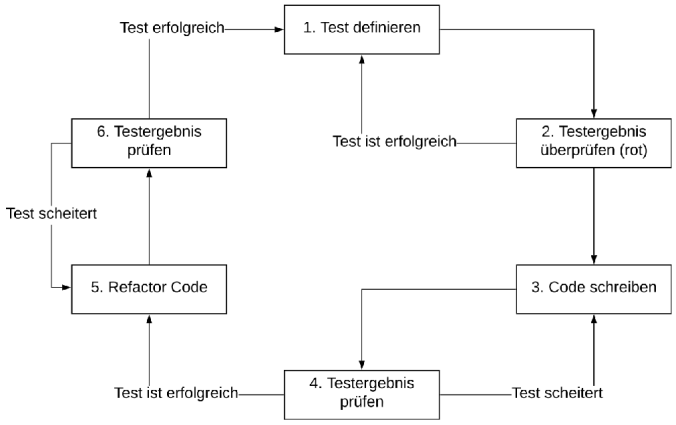
Nach jedem definierten Test wird die Software mit Code erweitert, d.h. TDD läuft inkrementell ab. Nach jedem Schritt wird die Software mit einer Funktion erweitert und wieder getestet. Schlägt ein Test fehl und wird rot, muss die Funktion zu dem Test geschrieben werden.

1. Der erste Schritt ist es, einen Test zu definieren. Der Test wird rot (Rote Phase).
2. Als nächstes muss der Code/Funktion zu den roten Test geschrieben werden. Im Idealfall wird der Test grün.
3. In der Refactoring Phase kann man den Code ändern und verbessern.

Rote Phase: In dieser Phase schreibt man einen Test über ein Verhalten, welchem implementiert werden soll. Hier wird kein Code geschrieben. Man kann auch sagen, dass sich der Entwickler in dieser Phase, wie ein Benutzer der Software verhält. Die Implementierung sollte hier am besten vergessen werden, der Test wird so geschrieben, als ob der Code schon implementiert wurde. Der Entwickler sollte sich auf die Funktion konzentrieren, welche benötigt wird.

Grüne Phase: In der grünen Phase wird Code geschrieben, um den Test zu bestehen. Die Lösung kann in dieser Phase unkompliziert sein, man darf gegen Best Practices verstoßen und auch duplizierten Code schreiben. Der geschriebene Code in dieser Phase, nennt man auch Produktivcode.

Refactoring Phase: In dieser Phase kann der Code verändert werden, im besten Fall, dass dieser besser wid. Auf eines muss man hier besonders achten, duplizierter muss entfernt werden.



1.2 Vor- und Nachteile

Vorteile:

- Man kann die Anforderungen oder das Design nochmal durchdenken, bevor eine Funktion implementiert wird.
- Die Qualität der Software, ist auf einem hohen Niveau.
- Die Software ist weniger fehleranfällig.
- Fehler zu entfernen ist einfacher und schneller.
- Der Code ist gut strukturiert, da die Funktionen nach und nach codiert werden.
- Redundanter und überflüssiger Code werden vermieden.
- Die Langzeitkosten sind geringer.

Nachteile:

- Der Zeitaufwand kann bei TDD hoch sein.
- Der Entwickler schreibt die doppelte Menge an Code, den Test und die Funktion.
- Die Tests müssen gut ausgewählt sein und codiert sein. Wenn ein Test Fehlerhaft ist, ist es wahrscheinlich, dass die dazu geschriebene Funktion ebenso fehlerhaft ist.

Bei Projekten, wo der Entwickler keinen engen Zeitplan vergegeben hat, kann TDD eingesetzt werden. Im Gegenteil bei Projekten mit einem engen Zeitplan, würden wir TDD nicht empfehlen, da der Zeitaufwand höher sein kann.

1.3 TDD und PMD

Da in der Dokumentation und wir nichts in den Commits ablesen kommten, sind wir davon ausgegangen, dass PMD kein TDD verwendet. TDD wäre bei PMD aber eine interessante Sache, da PMD ein Tool ist zum überprüfen von Code, sollten die Funktionen von PMD sehr gut und möglichst Fehlerfrei funktionieren.

2 Testinfrastruktur

2.1 Warum sollte eine seperate Testinfrastruktur verwendet werden ?

1. Entwicklungs- und Testarbeiten am Produktivsystem werden vermieden.
2. Entwicklung und Testen kann unabhängig durchgeführt werden.
3. einheitliche Umgebung unabhängig vom Entwicklerrechner.
4. Die Umstellung auf eine neue Version kann getestet werden.

2.2 Zeit- und Kostensparender Aufbau einer Testumgebung

Um zu vermeiden das im späteren Verlauf eines Projektes unterschiedliche Testframeworks verwendet werden, sollte die gesamte Testumgebung frühzeitig und einheitlich erstellt werden.

2.3 Testinfrastruktur im Projekt

Im Projekt PMD wird TravisCI als Continuous Integration & Deployment Server eingesetzt. Dieser führt sämtliche Testfälle auf Linux, Windows und MacOS aus. Die Testfälle werden bei neuen Commits und Pullrequests ausgeführt.

Als Testframework werden JUnit 4 & 5, AssertJ, KotlinTest für Unit Tests, sowie pmd-test und pmd-lang-test für spezifische Tests verwendet. Zusätzlich werden die JUnit Erweiterungen JUnitParams und SystemRules verwendet. Mockito und WireMock werden fürs Mocking von Klassen und HTTP APIs angewendet.

3 Teststufen

3.1 Komponententest

Die Komponenten werden möglichst unabhängig voneinander getestet damit gefundene Fehler tatsächlich an der getesteten Komponente liegt und nicht an der Integration oder Zusammenarbeit von verschiedenen Komponenten. Wenn die getesteten Komponenten abhängig sind, können gefundenen Fehler aus unterschiedlichen Gründen entstehen. Fehler können auftreten in: der Komponente, abhängige Komponenten oder der Kommunikation mehrerer Komponenten. Da bei dem Komponententest aber Fehler in einzelnen Komponenten gefunden werden sollen, werden Abhängigkeiten vermieden. Wenn Komponenten von Ausgaben anderer Komponenten abhängig sind, kann diese Ausgabe im Testfall definiert werden und die Abhängigkeit vermieden werden.(Die Ausgabe ist teil des Testfalls) Komponenten sollten andere Komponenten nicht direkt verwenden, um Abhängigkeiten zu minimieren.

3.2 Integrationstest

Im Integrationstest werden Fehler gefunden die durch die zusammen Arbeit der Komponenten entstehen. Zum Beispiel kann eine Komponente die Ausgabe einer anderen Komponente falsch interpretieren, wenn die Komponenten von verschiedenen Datenformaten ausgehen. Auch können Fehler durch die Reihenfolge der Ausführung von Komponenten entstehen(wenn z.b. interne Zustände geändert werden).

3.3 Teststufen im PMD

In PMD werden Komponenten, Integrations und Systemtest durchgeführt. Abnahmetests gibt es nicht, da PMD ein open Source Projekt ist und keine direkte Kundeninteraktion stattfindet. Komponententests, Integrationstests und Systemtests werden über JUnit abgedeckt.

3.4 Systemtest

Testfälle für Systemtests werden basierend auf den gesamten Anforderungen erstellt. Systemtests können von use Cases abgeleitet werden. Dabei werden funktionale sowie nicht-funktionale Anforderungen beachtet.