

Titel der Bachelorarbeit

BACHELORARBEIT 1

StudentIn Vorname Familienname, Personenkennzeichen

BetreuerIn Titel Vorname Familienname

Ort, Abgabedatum

Eidesstattliche Erklärung

Hiermit versichere ich, Vorname Familienname, geboren am **Tag.Monat.Jahr** in **Ort**, dass ich die Grundsätze wissenschaftlichen Arbeitens nach bestem Wissen und Gewissen eingehalten habe und die vorliegende Bachelorarbeit von mir selbstständig verfasst wurde. Zur Erstellung wurden von mir keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Ich versichere, dass ich die Bachelorarbeit weder im In- noch Ausland bisher in irgendeiner Form als Prüfungsarbeit vorgelegt habe und dass diese Arbeit mit der den BegutachterInnen vorgelegten Arbeit übereinstimmt.

Ort, am **Tag.Monat.Jahr**

Unterschrift

Vorname Familienname

Personenkennzeichen

Kurzfassung

Deutsche Zusammenfassung ...
... ungefähr 200 Worte ...

Schlagwörter:

Abstract

English abstract ...

... about 200 words ...

Keywords:

a few descriptive keywords

Inhaltsverzeichnis

1	Test-Driven Development	1
1.1	Die zwei Regeln von TDD	1
1.2	Motivation: Angst & Mut	2
1.3	Der TDD-Kreislauf	3

1 Test-Driven Development

- Einleitung TDD -

1.1 Die zwei Regeln von TDD

Für das Arbeiten mit TDD gelten zwei simple Regeln (Beck 2003).

1. Es wird nur neuer Code geschrieben, wenn einer der automatisierten Tests fehlschlägt.
2. Duplikationen werden eliminiert.

Durch die Anwendung dieser einfachen Regel ergeben sich folgende Implikationen.

- Es entsteht funktionierender Code, welcher in Kombination mit den Tests sofortiges Feedback zwischen Programmierentscheidungen liefert.
- Die Tests müssen von der/m EntwicklerIn selbst geschrieben werden. Die/Der EntwicklerIn kann selbstverständlich nicht auf Tester warten, denn sie/er ist abhängig von sofortigem Feedback um Programmierentscheidungen zu treffen.
- Die Entwicklungsumgebung muss auf jede Änderung schnell reagieren und das Feedback der Tests darstellen können.
- Um das Testen einfach zu halten muss hoch kohäsiver Code mit leicht gekoppelten Komponenten geschrieben werden.

Durch die Regeln sowie die Implikationen ergeben sich drei Aufgaben in folgender Reihenfolge.

1. **Rot:** Einen kleinen Test schreiben.
Der kleine Test muss nicht erfolgreich durchgehen, er muss auch noch nicht kompilieren.
2. **Grün:** Den Test *schnell* zum Laufen bringen.
Hier werden die einfachsten Änderungen welche sinnvoll erscheinen am Code vorgenommen um den zuvor erstellten Test erfolgreich abschließen zu können.
3. **Refactor:** Alle Duplikationen, welche durch das schnelle Hinzufügen von Code entstanden sind, eliminieren.

Wenn es möglich ist, dieses theoretische Modell als Programmierungsstil umzusetzen, erhält man Code, welcher beinahe ausschließlich durch fehlgeschlagene Tests gefordert wurde. Die Anwendung dieses Programmierstils ergibt nicht nur die oben angeführten technischen, sondern ebenfalls soziale Implikationen.

- Wenn die *Fehlerdichte* möglichst gering gehalten werden kann, kann aus *Qualitätssicherung* statt passiver zur aktiven Arbeit werden.
- Wenn die Anzahl von *negativen Überraschungen* reduziert werden kann, wird der gesamte Entwicklungsprozess besser plan- und schätzbar.
- Da durch TDD sauberer Code entsteht, welcher zusätzlich durch Tests dokumentiert ist, kann die Kommunikation zwischen Software-EntwicklerInnen geschärft und klarer werden.

1.2 Motivation: Angst & Mut

Beck 2003 stellt folgende Behauptung auf:

TDD ist ein Weg um die Angst einer/s EntwicklerIn zurecht zu kommen.

Hierbei geht es Beck um die legitime Angst einer/s EntwicklerIn „this-is-a-hard-problem-and-I-can’t-see-the-end-from-the-beginning“. Durch diesen Ausdruck wird die Angst von einer zu großen und schwierigen Aufgabe erdrückt zu werden beschrieben. Wenn ein/e EntwicklerIn sich mit diesem Problem konfrontiert sieht ergeben sich aus der Angst folgende negative Reaktionen.

- Angst macht EntwicklerInnen *zögerlich*.
- Angst macht EntwicklerInnen *weniger kommunikativ*.
- Angst macht EntwicklerInnen *scheu* vor *Feedback*.
- Angst macht EntwicklerInnen *mürrisch*.

Diese Reaktionen wirken sich alle negativ auf die Code-Qualität sowie auf die Lösung des Problems aus.

Die richtigen Reaktionen wären jedoch folgende.

- EntwicklerInnen sollten nicht zögern, sondern schnell und konkret an einem gegebenen Problem *lernen und wachsen*.
- EntwicklerInnen sollten nicht schweigen und sich zurückziehen, sondern *klarer kommunizieren*.
- EntwicklerInnen sollten aktiv nach *konstruktivem Feedback* suchen.

Laut Beck 2003 werden diese Reaktionen durch TDD erreicht. Das große, erdrückende und angst-fördernde Problem wird zuerst durch viele kleine Tests ausgedrückt. Jeder Test wird der Reihe nach zum Laufen gebracht. Wenn einer der Tests läuft, weiß die/der EntwicklerIn auch, dass der Test sowie der zugehörige Code funktioniert. Sie/er bekommt positives Feedback und kann sich auf das nächste Problem konzentrieren, wird also nicht mehr

abgelenkt. Sollte dieser Test durch weitere Änderungen am Code fehlschlagen bekommt die/der EntwicklerIn direktes Feedback. Durch kleine Iterationen und simple Schritte können Fehler einfach gefunden werden.

Weiters beschreibt Beck TDD als Bewusstsein der Lücke zwischen Programmierentscheidungen und Feedback sowie die Möglichkeiten um diese Lücke kontrollieren zu können.

1.3 Der TDD-Kreislauf

1. Test schreiben:

Der Test sollte widerspiegeln, wie ein/e EntwicklerIn sich die zukünftige Operation wünscht. Der Test ist eine Geschichte. Diese Geschichte soll alle möglichen Elemente beinhalten, welche notwendig sind um die richtigen Antworten für die Geschichte zu erhalten.

2. Test zum Laufen bringen:

Den Test *schnell* zum Laufen bringen ist das Wichtigste. Wenn eine klare, saubere Lösung offensichtlich ist, dann spricht nichts dagegen diese auch zu implementieren. Wenn diese Lösung jedoch zu lange dauert, um den Test *schnell* zum Laufen zu bringen - also das „grüne“ Feedback verzögern - sollte eine Notitz erstellt werden und die Konzentration sofort wieder auf das eigentliche Problem gelenkt werden. Oft werden hier auch „Fake“ Implementierungen angewendet.

3. Es richtig machen:

Schritt für Schritt können nun Duplikationen entfernt sowie etwaige „Fake“ Implementierungen aus Schritt 2 behoben werden.

Das Ziel dieser drei Schritte ist „clean code that works“, also „*sauberen, funktionierenden Code*“ zu erhalten.

Folgendes Beispiel illustriert diesen Kreislauf anhand einer einfachen Implementierung für simple Addition. Die erläuternde Beispiele zu TDD werden in JavaScript verfasst, da sich diese Arbeit primär mit JavaScript und AngularJS befasst. Das Beispiel kann im referenziertem GitHub-Repository in dem Verzeichnis „source/simple-tests/step-1“ gefunden werden.

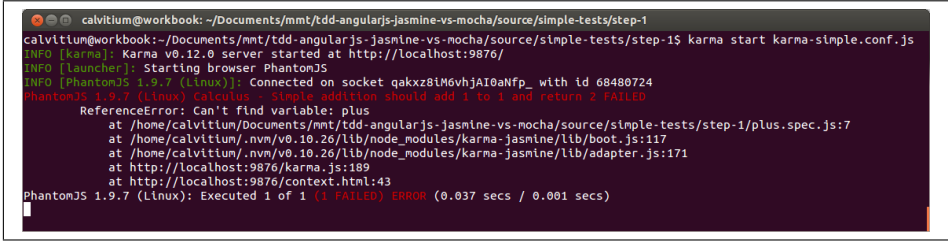
Schritt 1: Test schreiben. Der erste einfache Test beschreibt eine Geschichte. In dieser Geschichte soll die Funktion „plus(arg1, arg2)“ die zwei übergebenen Parameter addieren und ein Ergebnis zurückliefern. Für die einfachste Integer-Variante bedeutet das also: „1+1=2“

```
1 describe("Calculus - Simple addition", function() {
2
3     it("should add 1 to 1 and return 2", function() {
4         expect(plus(1,1)).toEqual(2);
5     });
6 }
```



```
6
7 });
```

Dieser Test wird per Kommandozeile im Browser „PhantomJS“ ausgeführt. Der Befehl dafür lautet: „karma start karma-simple.conf.js“. Der erste Test ist geschrieben, jedoch gibt es noch keine Implementierung, dass heißt hier wird ein negatives Testergebnis erwartet. Das erwartete Ergebnis ist auf Abbildung 1 zu sehen.



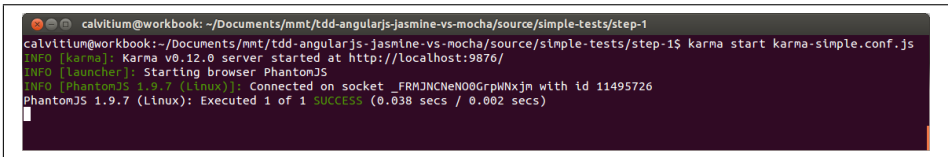
```
calvitiun@workbook: ~/Documents/mmt/tdd-angularjs-jasmine-vs-mocha/source/simple-tests/step-1
calvitiun@workbook:~/Documents/mmt/tdd-angularjs-jasmine-vs-mocha/source/simple-tests/step-1$ karma start karma-simple.conf.js
INFO [karma]: Karma v0.12.0 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Linux)]: Connected on socket qakxz8iM6vhjAI0aNFp_ with id 68480724
PhantomJS 1.9.7 (Linux): Calculus - Simple addition should add 1 to 1 and return 2 FAILED
ReferenceError: Can't find variable: plus
    at /home/calvitiun/Documents/mmt/tdd-angularjs-jasmine-vs-mocha/source/simple-tests/step-1/plus.spec.js:7
    at /home/calvitiun/.nvm/v0.10.26/lib/node_modules/karma-jasmine/lib/boot.js:117
    at /home/calvitiun/.nvm/v0.10.26/lib/node_modules/karma-jasmine/lib/adaptor.js:171
    at http://localhost:9876/karma.js:189
    at http://localhost:9876/context.html:43
PhantomJS 1.9.7 (Linux): Executed 1 of 1 (1 FAILED) ERROR (0.037 secs / 0.001 secs)
```

Abbildung 1: Terminal Ausgabe des ersten Tests

Schritt 2: Zum Laufen bringen. Um diesen Test möglichst schnell dazu zu bringen grünes Feedback zu geben, wird die einfachste und naivste Implementierung verwendet. Die einfachste Variante bedeutet also eine Funktion zu definieren, welche zwei Parameter übernimmt und „2“ zurückgibt.

```
1 var plus = function(augend, addend) {
2   return 2;
3 };
```

Wir wissen an dieser Stelle natürlich, dass hier eine klare, saubere und offensichtliche Lösung existiert. Es spricht natürlich nichts dagegen diese hier auch anzuwenden, um jedoch auch den dritten Schritt illustrieren zu können, wird hier naiv vorgegangen. Nach dieser naiven Implementierung ändert sich das Ergebnis des Tests von rot auf grün (siehe Abbildung 2).



```
calvitiun@workbook: ~/Documents/mmt/tdd-angularjs-jasmine-vs-mocha/source/simple-tests/step-1
calvitiun@workbook:~/Documents/mmt/tdd-angularjs-jasmine-vs-mocha/source/simple-tests/step-1$ karma start karma-simple.conf.js
INFO [karma]: Karma v0.12.0 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Linux)]: Connected on socket _FRMJNCNeN0GpWNxjm with id 11495726
PhantomJS 1.9.7 (Linux): Executed 1 of 1 SUCCESS (0.038 secs / 0.002 secs)
```

Abbildung 2: Terminal Ausgabe des ersten Tests nach naiver Implementierung

Schritt 3: Es richtig machen. Obwohl der Test nun bereits erfolgreich läuft, ist natürlich klar, dass diese naive Implementierung nicht korrekt ist. Es handelt sich um eine „Fake“ Implementierung. Um nun die richtige Berechnung durchzuführen ändern wir die „plus(arg1, arg2)“ Funktion folgendermaßen.

```
1 var plus = function(augend, addend) {  
2   return augend+addend;  
3 };
```

Nach dem dritten Schritt wird erneut der Test ausgeführt um zu garantieren, dass das Korrigieren der naiven Implementierung auch funktioniert. Es wird ein positives Ergebnis erwartet (siehe Abbildung 3).

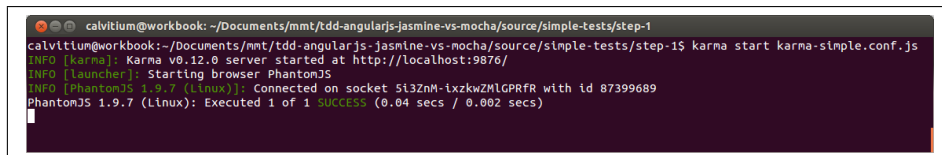


Abbildung 3: Terminal Ausgabe des ersten Tests nach Richtigstellung der Implementierung

Durch die Anwendung des TDD-Kreislaufs wurde hier *sauberer* und *funktionierender* Code produziert.

Abkürzungsverzeichnis

CRAN Comprehensive R Archive Network

Abbildungsverzeichnis

1	Terminal Ausgabe des ersten Tests	4
2	Terminal Ausgabe des ersten Tests nach naiver Implementierung	4
3	Terminal Ausgabe des ersten Tests nach Richtigstellung der Implementierung	5

Listings

Tabellenverzeichnis

Literaturverzeichnis

Beck, Kent. 2003. *Test-Driven Development - By Example*. Boston, Mass. [u.a.]: Addison-Wesley. ISBN: 0-321-14653-0.

Anhang

Datensets