Test-driven Development mit AngularJS: Ein Vergleich der Testing Frameworks Jasmine und Mocha

BACHELORARBEIT 2

Student Sebastian Slamanig, 1110601032
Betreuerin DI Brigitte Jellinek

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 Einführung **1**

1.1 Relevanz & Problemstellung 1

1.2 Methoden 1

1.2.1 Test-driven Development 1

1.2.2 AngularJS 2

1.2.3 Testing Frameworks 2

1.3 Forschungsfrage 2

1.4 Begrifflichkeit 2

2 Entwicklungsprozess **3**

2.1 Test-Driven Development 3

2.1.1 Die zwei Regeln von TDD 3

2.2 Motivation: Angst & Mut 4

2.2.1 Der TDD-Kreislauf 5

2.2.2 Test-Driven Development Patterns 7

2.3 Andere Entwicklungsmethoden 11

2.4 Gegenüberstellung 11

3 AngularJS **12**

3.1 Hintergründe 12

3.2 Anatomie von AngularJS 12

3.3 Model-View-Controller Design Pattern 12

3.4 Komponenten 12

3.4.1 Controller 12

3.4.2 Directives 12

3.4.3 Services 12

3.4.4 Dependency-Injection 12

3.4.5 Data-Binding 12

3.5 Entwickeln mit AngularJS 13

4 Testing Frameworks **14**
4.1 Jasmine 14
4.1.1 Hintergründe 14
4.1.2 Funktionen 14
4.1.3 Testen von asynchronem Javascript 14
4.2 Mocha 14
4.2.1 Hintergründe 14
4.2.2 Funktionen 14
4.2.3 Testen von asynchronem Javascript 14
4.3 Gegenüberstellung 14

5 Conclusio **15**

6 Ausblick **16**

1 Einführung

1.1 Relevanz & Problemstellung

Bessere Code-Qualität mit im Vergleich minimal erhöhtem Zeitaufwand ist ein erstrebenswertes Ergebnis, welches durch Test-driven Development (TDD) versucht wird, zu erreichen. Durch die Idee von TDD - nämlich kurze Iterationen von:

1. Tests schreiben (ohne sich Gedanken über den tatsächlichen Code zu machen)
2. Tests ausführen und fehlschlagen lassen
3. Code implementieren
4. Tests ausführen und den Code eventuell weiter implementieren, bis die Tests nicht mehr fehlschlagen
5. Tests und Code refaktorieren
6. Tests erneut ausführen

wird eine hohe Testabdeckung und ein sauberer Code durch das Refaktorieren erzielt.

AngularJS ist ein umfassendes MVC Framework und bringt viele eigene Komponenten mit sich, wie beispielsweise Direktiven und Services. Um test-driven mit AngularJS entwickeln zu können, ist es wegen der Vielfalt der eigenen Komponenten relevant, alle diese Komponenten testen zu können.

Mit der Relevanz von TDD mit AngularJS geht die Problemstellung der Auswahl eines passenden Testing Frameworks einher. Jasmine und Mocha sind zwei JavaScript Testing Frameworks, welche sich für diese Aufgabenstellung anbieten. Deshalb werden beide Frameworks in dieser Arbeit gegenüber gestellt und analysiert, um die Unterschiede beziehungsweise die Vor- sowie Nachteile jedes der Frameworks darstellen zu können.

1.2 Methoden

1.2.1 Test-driven Development

Eine theoretische Erfassung von Test-driven Development (TDD) und was die Vor- und Nachteile von TDD verglichen mit anderen Entwicklungsmethoden sind, ist grundlegend für den weiteren Inhalt der Arbeit. Andere Entwicklungsmethoden beinhaltet hier das Entwickeln ohne Testen sowie das Entwickeln und Testen im nachhinein.

1.2.2 AngularJS

AngularJS wird zusammen mit den dazugehörigen Komponenten theoretisch und praktisch dargestellt. Weitere Erläuterungen zu dem „Model-View-Controller (MVC)“ Design-Pattern werden zum Verständnis des Frameworks auch behandelt. Die Theorie und die Praxis arbeiten hier sehr dicht zusammen, um im Vorfeld alle Facetten des MVC-Frameworks abzuklären.

1.2.3 Testing Frameworks

Um abklären zu können, welches der beiden JavaScript Testing-Frameworks „Jasmine“ und „Mocha“ besser geeignet ist, um alle Eigenschaften von AngularJS abzudecken, wird ein direkter theoretischer Vergleich der Frameworks und deren Eigenschaften erfolgen. Darüber hinaus wird eine Applikation test-driven mit beiden Frameworks entwickelt. Die Applikation wird alle der relevanten Eigenschaften von AngularJS behandeln. Der Entwicklungsprozess wird innerhalb der Arbeit dokumentiert.

1.3 Forschungsfrage

Wie ist der aktuelle Stand um AngularJS test-driven zu entwickeln und welches der beiden JavaScript Testing-Frameworks „Jasmine“ und „Mocha“ ist geeigneter für diesen Zweck?

1.4 Begrifflichkeit

- Begrifflichkeit -

2 Entwicklungsprozess

2.1 Test-Driven Development

- Einleitung TDD -

2.1.1 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. Dupplikationen 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 sfortigem 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, ermuss 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 Dupplikationen, 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 Überaschungen* 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.

2.2 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 negative 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 gegebenem 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 nacht 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.

2.2.1 Der TDD-Kreislauf

1. **Test schreiben**:
   Der Test sollte wiederspiegeln, 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“

```
describe("Calculus - Simple addition", function() {

  it("should add 1 to 1 and return 2", function() {
    expect(plus(1,1)).toEqual(2);
  });

});
```

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 erwartet Ergebnis ist auf Abbildung 1 zu sehen.

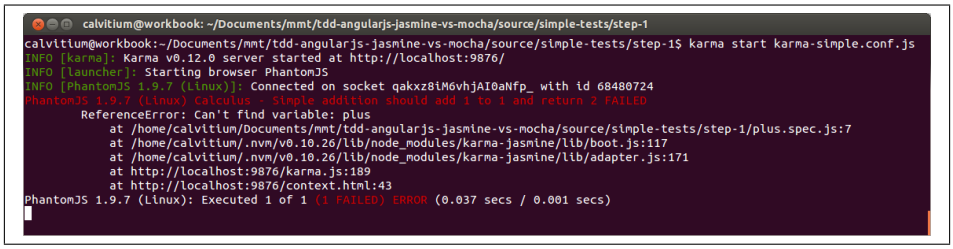

Abbildung 1: Terminal Ausgabe des ersten Tests

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

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

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).

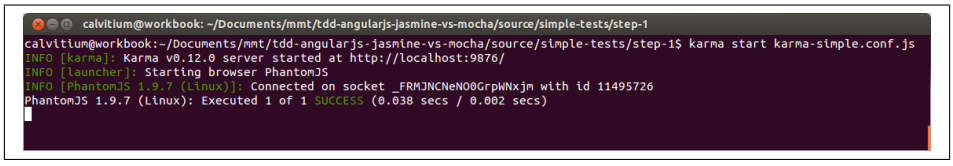

Abbildung 2: Terminal Ausgabe des ersten Tests nach naiver Implementierung

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).

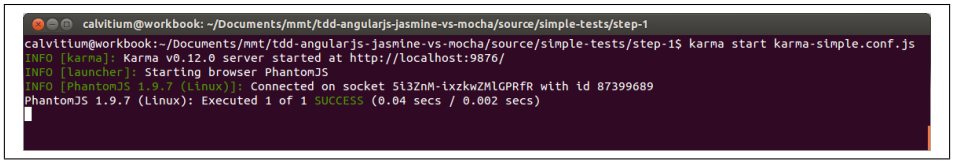

Abbildung 3: Terminal Ausgabe des ersten Tests nach Richtigstellung der Implementierung

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

2.2.2 Test-Driven Development Patterns

Red Bar Patterns

Diese Muster betreffen den ersten Teil des TDD-Kreislaufs: das Test Schreiben. Konkret behandeln sie die Fragen:

- „Wann“ sollen Tests geschrieben werden?
- „Wo“ sollen Tests geschrieben werden?
- „Wann“ soll mit dem Tests Schreiben „aufgehört“ werden?

One Step Test

Bei TDD handelt es sich wie bereits erwähnt um einen Prozess, bei welchem in kleinen Schritten und Schritt für Schritt Software aus Tests geformt wird. In diesem Prozess ist es notwendig sich für die Reihenfolge der Test-Schritte (in weiterer Folge auch für die Reihenfolge, in welcher die Komponenten der Software entstehen soll) zu entscheiden.

Beck 2003[134] liefert auf die Frage „Welchen Test sollte ich als nächstes schreiben?“ eine klare Antwort: Es gibt keine. Die Frage ist hinsichtlich der Anforderungen der Software (beziehungsweise deren Komponenten) sowie der/s Entwicklerin/s zu individuell. Beck gibt jedoch den Tipp, weder Tests mit einer offensichtlichen Implementierung, noch Tests,

welche zu kompliziert erscheinen zu wählen. Man sollte Schritt für Schritt jene Tests wählen, bei welchen man sich wohlsten fühlt. Sollten nur zu komplizierte Probleme und Tests vorliegen müssen die Schritte verkleinert und somit das Szenario noch weiter unterteilt werden.

Die Mechanik des „One Step Test"'s wird von Beck 2003[134] mit „known to unknown" beschrieben.

Starter Test

Mit welchem Test soll begonnen werden?

Eines der Grundkonzepte von TDD besagt, dass die Konzentration immer nur auf dem aktuellen Problem und Test liegen soll. Diesem Konzept steht jedoch die Möglichkeit mit einem realistischen Test zu starten gegenüber. Bei dem Versuch einen realistischen Test zu schreiben, wird der Fokus sofort auf einige andere Fragen gelenkt. „Was ist der richtige Input?", „Was ist das richtige Ergebnis?", „Welche Parameter sind notwendig für Lösung des Problems?" oder „Wohin gehört dieser Test eigentlich?". Man findet sich bei dem Schreiben realistischer Tests häufig dabei wieder über viele Probleme und Fragen auf einmal nachzudenken, verliert also den Fokus auf den aktuellen Test des aktuellen Problems.

Ein realistischer Test bedeutet eine Verzögerung des Abstands der „Rot-Grün-Refactor" Zyklen - diese sollen jedoch möglichst kurz gehalten werden. Der erste Test, also der „Starter Test" soll so trivial wie möglich sein.

„Start by testing a variant of an operation that doesn't do anything", Beck 2003[134].

Learning Test

Wie bereits erwähnt geht es bei TDD nicht nur um eine möglichst hohe Test-Dichte, sondern ebenfalls um andere Faktoren, wie Design, Schätzbarkeit, sauberen Code, psychologische Auswirkungen, etc.

Eine weitere Methode TDD einzusetzen betrifft das Lernen des Umgangs mit einer neuen API oder Bibliothek. Wenn man eine neue Bibliothek verwendet bietet es sich an, gegen diese Tests zu schreiben anstatt direkt Implementierungen damit vorzunehmen - ansonsten sollten keine Tests für externe Software mptwendig sein.

Regression Test

Auch TDD kann keine 100%-ige Fehler-Freiheit garantieren. Sollte ein Problem berichtet werden, ist der erste Schritt einen neuen Test für dieses Problem zu verfassen. Der Test muss den einfachsten Fall, welcher das Fehlverhalten hervorruft behandeln. Durch das nachträgliche absolvieren des TDD-Kreislaufs für das gemeldete Problem werden die

gleichen Vorteile wie auch für direktes TDD erreicht. ...

Green Bar Patterns

Die „Red Bar Patterns“ helfen dabei die notwendigen Tests zu verfassen. Die „Green Bar Patterns“ hingegen geben nun Ansätze, wie die geschriebenen und fehlschlagenden Tests schnellst möglich erfolgreich abgeschlossen werden können.

Fake it ('til you make it)

Um die Abstände der „Rot-Grün-Refactor“ Zyklen so gering wie möglich halten zu können wird immer die einfachste Variante aller möglichen Implementierungen verwendet: Eine Konstante zurück liefern.

Nachdem der Test grün ist wird Schritt für Schritt die Implementierung richtig gestellt und die Konstante mit Variablen in einen Ausdruck gewandelt. Hier stellt sich jedoch die Frage „Wieso offensichtlich falschen Code schreiben?“ - Um den Test schnell erfolgreich laufen lassen zu können. Laut Beck 2003[152] hat dieses Muster zwei starke Effekte. Wenn der Test grün ist fühlt man sich sicher. Man kann sich voll und ganz auf das richtige Refaktorieren konzentrieren. Also einerseits erfüllt „Fake it“ einen psychologischen Effekt. Andererseits hilft es bei der Abgrenzung von zukünftigen Problemen. Wie schon erwähnt dient TDD auch dazu den Fokus und die Konzentration immer bei dem aktuellen Problem zu halten. Als ersten Schritt eine Konstante zurück zu liefern unterstützt dabei die/den EntwicklerIn.

BAIIISPÜÜÜÜL + Duplikation zwischen Test & Implementierung removen undso.

Triangulate

Ein weiterer Weg um schnell einen erfolgreichen Test zu erreichen ist „Triangulate“. Bei Triangulate geht es darum, nicht nur ein mögliches Ergebnis zu testen, sondern zwei oder mehrere Ergebnisse.

Anhand des folgenden Beispielswird der direkte Unterschied zu „Fake it“ erklärt.

BAIIISPÜÜÜÜÜHL

Bei Triangulate wird also nicht mittels direkter Eliminierung der Duplikation zwischen Test und Implementierung refaktoriert, sondern durch das Hinzufügen eines weiteren möglichen Ergebnisses.

Obvious Implementation

Wie die bisherigen Beispiele für „Fake it“ und „Triangulate“ zeigen, handelt es sich dabei um wirklich kleine Schritte. Speziell anhand dieses Beispiels lässt sich das dritte Green Bar Patterns erklären: „Obvious Implementation“.

Wenn eine Operation so einfach ist, wie in dem Beispiel, spricht nichts dagegen die offensichtliche Implementierung direkt zu verwenden. Wenn die Probleme allerdings komplizierter werden, ist es oft schwierig die Disziplinen „sauberer“ und „funktionierend“ gleichzeitig zu erreichen. Außerdem soll der Abstand zwischen den Zyklen möglichst gering gehalten werden, was bei einer offensichtlichen Lösung für ein großes Problem eventuell nicht eingehalten werden kann.

BAISPÜÜÜÜHL

One to many

Auch bei Operationen, welche eine Kollektion von Objekten benötigen, gilt: einfach und simpel starten - „one to many“(Beck 2003)[154].

Zuerst erfolgt die Implementierung mit einem Objekt und erst wenn der Test erfolgreich war wird Schritt für Schritt die Kollektion übernommen.

BEISPÜÜÜÜÜHL - Beachten: Änderungen Isolieren

2.3 Andere Entwicklungsmethoden

- Andere Entwicklungsmethoden -

2.4 Gegenüberstellung

- Gegenüberstellung -

3 AngularJS

- AngularJS -

3.1 Hintergründe

- Hintergründe -

3.2 Anatomie von AngularJS

- Anatomie von AngularJS -

3.3 Model-View-Controller Design Pattern

- Model-View-Controller Design Pattern -

3.4 Komponenten

- Komponenten -

3.4.1 Controller

- Controller -

3.4.2 Directives

- Directives -

3.4.3 Services

- Services -

3.4.4 Dependency-Injection

- Dependency-Injection -

3.4.5 Data-Binding

- Data-Binding -

3.5 Entwickeln mit AngularJS

- Entwickeln mit AngularJS -

4 Testing Frameworks

- Testing Frameworks -

4.1 Jasmine

- Jasmine -

4.1.1 Hintergründe

- Hintergründe -

4.1.2 Funktionen

- Funktionen -

4.1.3 Testen von asynchronem Javascript

- Testen von asynchronem Javascript -

4.2 Mocha

- Mocha -

4.2.1 Hintergründe

- Hintergründe -

4.2.2 Funktionen

- Funktionen -

4.2.3 Testen von asynchronem Javascript

- Testen von asynchronem Javascript -

4.3 Gegenüberstellung

- Gegenüberstellung -

5 Conclusio

- Conclusio -

6 Ausblick

- Ausblick -

Abkürzungsverzeichnis

CRAN Comprehensive R Archive Network

Abbildungsverzeichnis

1 Terminal Ausgabe des ersten Tests 6
2 Terminal Ausgabe des ersten Tests nach naiver Implementierung 6
3 Terminal Ausgabe des ersten Tests nach Richtigstellung der Implementierung 7

Listings

Tabellenverzeichnis

Literaturverzeichnis

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

Burnham, Trevor. 2011. *CoffeeScript: Accelerated JavaScript Development.* The Pragmatic Bookshelf. ISBN: 978-1-934356-78-4.

Green, Brad, und Shyam Seshadri. 2013. *AngularJS: Less Code, More Fun, and Enhanced Productivity with Structured Web Apps.* O'Reilly Media, Inc. ISBN: 978-1-449-34485-6.

Gärtner, Markus. 2012. *ATDD by Example: A Practical Guide to Acceptance Test-driven Development.* Addison-Wesley. ISBN: 978-0321784155.

Hahn, Evan. 2013. *JavaScript Testing with Jasmine: Javascript Behavior-Driven Development.* O'Reilly Media, Inc. ISBN: 978-1-449-35637-8.

Kozlowski, Pawel, und Peter B. Darwin. 2013. *Mastering Web Application Development with AngularJS.* Packt Publishing. ISBN: 978-1-782-16182-0.

MacCaw, Alex. 2011. *JavaScript Web Applications: jQuery Developers' Guide to moving State to the Client.* O'Reilly Media, Inc. ISBN: 978-1-449-30351-8.

Zakas, Nicholas C. 2012. *Maintainable JavaScript: Writing Readable Code.* O'Reilly Media, Inc. ISBN: 978-1-449-32768-2.

Anhang

Datensets