Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Seminararbeit

im Bachelor-Studiengang Computer Science

Javascript Testing Frameworks

von

Per Bernhardt

Betreuer:	Christoph Tornau
Eingereicht am:	9. Dezember 2013

Eidesstattliche Erklärung

Per Bernhardt
Schumannstr. 115
53113 Bonn

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

.....	.....
Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

TDD Test Driven Development
BDD Behavior Driven Development
CI Continous Integration
DSL Domain Specific Language
IDE Integrated Development Environment
DOM Document Object Model

1 Einleitung

1.1 Problemstellung

Die Scriptsprache Javascript, die im Jahre 1995 von Netscape entwickelt wurde, erfreute sich in den letzten Jahren enormer Popularität im Bereich der Webseiten- Entwicklung (Vgl. Wikipedia 2013b). Dort wird sie vor allem zur Verbesserung der Interaktivität von Webseiten genutzt. Inzwischen werden aber auch ganze Anwendungen clientseitig im Browser mit HIlfe von Javascript betrieben. Ein prominentes Beispiel ist hier der E-Mail-Client Google Mail, dessen gesamte Anwendungsoberfläche auf Javascript aufsetzt (Vgl. Google 13). Seit einiger Zeit beginnt Javascript nun aber auch bei der serverseitigen Entwicklung eine Rolle zu spielen. Als großer Vertreter sei hier beispielhaft eBay genannt, die Teile ihrer Infrastrutur mit dem Javascript-Server node.js betreiben (Vgl. eBay 2011). In jüngster Vergangenheit entstehen auch viele hochinteraktive Browser-Spiele, z.B. auf Basis der Aves Engine, die sowohl clientseitig als auch serverseitig Javascript einsetzen (Vgl. Ihlenfeld 2010). In beiden Bereichen hat sich die Komplexität von Javascript-Programmen damit so weit gesteigert, dass sie mit jeder anderen Programmiersprache vergleichbar geworden ist. Damit werden natürlich verschiedene Software-Engineering-Methodiken, die sich im Laufe der Zeit über alle Sprachen hinweg etabliert haben, auch für Javascript interessant. Dabei müssen die Lösungen für Javascript allerdings einige Besonderheiten abdecken, die mit der Sprache einhergehen: So dient Javascript nach wie vor vor allem für die Erstellung interaktiver Webseiten, die in verschiedensten Browsern und Plattformen funktionieren sollen. In all diesen Browsern gibt es verschiedene Javascript-Umgebungen, die sich zum Teil syntaktisch unterscheiden (Vgl. Kleivane 2011, S. 17 ff).

1.2 Zielsetzung

Das Testen von Software war schon immer ein wichtiger Bestandteil des Software-Engineerings (Vgl. Kleivane 2011, S. 1). Ziel dieser Seminararbeit ist es, auf diesen Teilbereich einzugehen und ihn vor allem in Hinblick auf die für die Sprache Javascript zur Verfügung stehenden Tools und Frameworks zu durchleuchten.

1.3 Vorgehensweise

Die Seminararbeit gliedert sich neben dieser Einleitung in vier weitere Abschnitte:

- Zunächst wird im 2. Kapitel der Arbeit der aktuelle Stand der Formschung im Bereich der Softwaretests beschrieben, indem bewährte Testarten und Testmethodiken erläutert werden.
- Anschließend werden im 3. Kapitel eine Reihe konkreter Tools und Ansätze für die Javascript-Entwicklung aufgelistet und mit einander verglichen. Dabei wird vor allem darauf eingegangen, in wiefern diese die allgemeinen Konzepte des Testens umsetzen.
- Im 4. Kapitel werden dann Anhand eines Fallbeispiels einige der Tools angewendet.
- Im 5. und letzten Kapitel werden die Ergebnisse zusammengefasst und ein Fazit gezogen.

2 Stand der Forschnung

Das Testen von Software dient vor allem der Vermeidung von Fehlern innerhalb des Programmablaufs. Man unterscheidet dabei unter anderem verschiedene Testarten und verschiedene Testmethodiken, von denen einige im Folgenden näher erläutert werden.

2.1 Testarten

Es gibt verschiedene Arten von Softwaretests. Diese unterscheiden sich vor allem darin, was getestet wird, aber auch darin, wer den Test durchführt. Neben den Modul- und Integrationstests, die vom Entwicklungsteam durchgeführt werden, gibt es noch System- und Abnahmetests, bei denen das gesamte Softwareprodukt vom Kunden bzw. späteren Benutzer entweder auf einem Testsystem oder auf dem Produktivsystem getestet wird (Vgl. Wikipedia 2013d).

2.1.1 Modultests

Modultests (im Englischen auch Unit Tests genannt) sind Tests, die einzelne Module der Software testen. Für gewöhnlich versucht man hier möglichst kleine Module zu testen, um die Stabilität der einzelnen Tests zu steigern. Es wird also vor allem eine einzelne Methode bzw. Funktion getestet. Da der Test somit direkt mit einem Programmcode-Stück interagiert, ist klar, dass er selbst in der gleichen Programmiersprache verfasst wird. Er wird somit von einem Entwickler geschrieben. Der Test sollte sich dabei auf das gewünschte Verhalten der Methode konzentrieren und nicht auf deren konkrete Implementierung abstellen (Design-by-contract). Bei den Modultests hat sich im Laufe der Zeit eine bestimmte Definitionsweise etabliert, der die meisten Modultest-Frameworks folgen. Es gibt sogar eine Reihe von Frameworks (xUnit), die von Sprache zu Sprache portiert werden und alle auf das von Kent Beck entworfene SUnit für die Sprache Smalltalk zurückzuführen sind. Von ihm stammt auch die erste Portierung in die Sprache Java namens JUnit. Typisch für diese Frameworks ist, dass die eigentlichen Tests ihrerseits Methoden innerhalb einer Testklasse (Test-Case) sind. Mehrere dieser Testklassen können dann in Gruppen organisiert werden (Test-Suite). Außerdem gibt es fast immer die Möglichkeit, innerhalb einer Testklasse Methoden für die Initialisierung bzw. Deinitialsierung der Testumgebung zu definieren, die vor und nach jeder einzelnen Testmethode ausgeführt weren (Set-Up- und Tear-Down-Methoden). Innerhalb der eigentlichen Testmethode stehen dann weitere Hilfsmethoden zur Verfügung, mit denen einfach Behauptungen über das Testsubjekt formuliert werden können (Assertions), die dann während der eigentlichen Testausführung überprüft werden. Ein Modultest-Framework bietet neben einer einfachen Test-Definitions-Syntax auch ein Programm, Test-Runner genannt, welches die Auswahl bestimmter Tests erlaubt und nach deren Ausführung ein Testergebnis darstellt. (Vgl. Johansen 2010, S. 4 ff)

2.1.2 Integrationstest

Der Integrationstest dient dazu, das Zusammenspiel mehrerer Programmbausteine zu testen. Gerne wird er auch als Schnittstellentest bezeichnet, da er gegen eine bestimmte Schnittstelle des Programms arbeitet. Bei Webanwendungen bietet es sich hierbei an, die HTTP-Schnittstelle der Anwendung zu testen. Man spezifiziert also Aufrufe an den Webserver und überprüft dabei, ob die Antwort bestimmten Erwartungen entspricht. Meist werden auch die Integrationstests vom Entwickler selbst geschrieben und damit auch gerne wieder in der gleichen Programmiersprache wie das zu testende Programm selbst (Vgl. Wikipedia 2013a).

2.1.3 System- und Abnahmetests

Da bei den System- und Abnahmetests das fertige Softwareprodukt durch den Kunden bzw. späteren Benutzer getestet wird, ist für die Implementierung dieser Tests die der Software zugrunde liegende Programmiersprache unerheblich. Es gibt hier vor allem keine sprachspezifischen Frameworks.

2.2 Testmethodiken

Testmethodiken dienen dazu, das Schreiben der Tests in den eigentlichen Software-Entwicklungsprozess zu integrieren. In vielen Softwareprojekten haben sich dabei vor allem die folgenden Methodiken etabliert: Das Test Driven Development (TDD) (zu Deutsch "Testgetriebene Entwicklung"), das Behavior Driven Development (BDD) (zu Deutsch "Verhaltensgetriebene Entwicklung") und die Continous Integration (CI) (zu Deutsch "Kontinuierliche Integration").

2.2.1 Test Driven Development

Beim TDD handelt es sich um ein Vorgehen, dass der agilen Softwareentwicklung zugeschrieben wird. Im Gegensatz zur klassischen Softwareentwicklung, bei der Tests bei oder sogar erst nach der fertigen Implementierung des Programmcodes geschrieben werden, kann man vereinfacht sagen, dass der Entwickler beim TDD erst den Test und dann den eigentlichen Programmcode schreibt. Dies soll folgende Vorteile mit sich bringen (Vgl. Wikipedia 2013e):

- Der Ansatz garantiert eine sehr hohe Testabdeckung.
- Es wird verhindert, dass nicht testbarer Code entsteht.
- Zu Projektende werden Ressourcen (Zeit, Budget) typischerweise knapp und verhindert oft die Fertigstellung nachträglicher Tests.
- Das vorherige Schreiben des Tests begünstigt, dass der Tests auf das gewünschte Verhalten abstellt, da die Implementierung ja noch nicht vorliegt (Design-by-contract).

Das konkrete Vorgehen wird dabei gerne "Red, Green, Refactor"genannt und kann wie folgt beschrieben werden (Vgl. Kleivane 2011, S. 10):

1. Programmcode wird nur geschrieben, um fehlschlagende Tests zu reparieren; Am Anfang des Prozesses steht als immer zuerst das Anlegen eines neuen Tests; Der Test wird ausgeführt und schlägt aufgrund der fehlenden Implementierung fehl. ("Red")
2. Nun wird der einfachst mögliche Code entwickelt, der den Test repariert; Der Test wird ausgeführt und läuft fehlerfrei durch. ("Green")
3. Abschließend wird der Code solange Refactored, bis er keine Redundancen mehr enthält und die Tests dennoch fehlerfrei durchlaufen ("Refactor")

Da der Entwickler die Tests bei diesem Vorgehen sehr oft ausführt, nämlich vor der Implementierung und während der Implementierung so oft, bis der Code fertig implementiert ist, ist es sehr wichtig, dass das Ausführen der Tests sehr schnell und einfach vorgenommen werden kann. Man spricht gerne davon, dass die Tests äuf Knopfdruck"durchgeführt werden können müssen. Ein gutes Test-Framework bietet hierfür die Möglichkeit, den Test-Runner direkt aus der Integrated Development Environment (IDE) des Entwicklers aufzurufen.

2.2.2 Behavior Driven Development

Die Idee des BDD ist es, die Stabilität und Wartbarkeit von Testdefinitionen zu verbessern, indem das zu Testende Verhalten fokusiert wird. Dazu werden die Tests als Prosa-Text verfasst, der keine technischen Details der Testimplementierung enthält. Dieser muss

entsprechend einer sehr eingeschränkten Syntax, einer so genannten Domain Specific Language (DSL), folgen, da er zum Ausführen des Tests in entsprechenden Programmcode übersetzt wird. Weiterer Vorteil dieses Ansatzes ist es, dass die Tests auch von Nicht-Entwicklern (z.B. dem Kunden) besser gelesen und verstanden werden können (Vgl. Trostler 2013, S. 5-6).

2.2.3 Continous Integration

Bei der CI geht es darum, die Test-Ergebnisse und andere Metriken teamübergreifend zu erfassen und kontinuierlich zu verfolgen, um so im Lauf der Zeit positive oder negative Tendenzen in der Entwicklung des Softwareprodukts erkennen zu können. Typischerweise werden die Testergebnisse und andere Metriken in einem sogenannten CI-Server archiviert, der sich darum kümmert, diese von Zeit zu Zeit abzufragen bzw. selbständig zu generieren. Oft tut der Server dies zum Beispiel jede Nacht (Nightly Builds). Viele solcher CI-Server bieten aber sogar die Möglichkeit, sie über jede neue Version der Software zu informieren, die in der Versionskontrolle abgelegt wird und daraufhin ein neues Ausführen der Tests usw. auszulösen. In Bezug auf die Tests ist es dazu notwendig, dass diese nicht durch einen Entwickler sondern programmatisch ausgeführt werden können und ihre Ergebnisse anschließend auf maschinenlesbare Art und Weise zur Verfügung stellen (Vgl. Wikipedia 2013c).

3 Javascript Testing

Nachdem eingangs verschiedene allgemeine Testarten und -methodiken aufgeführt wurden, sollen im folgenden konkrete Frameworks für die Programmiersprache Javascript aufgeführt und in Bezug auf diese allgemeinen Ansätze miteinander verglichen werden. Wie in der Einleitung erläutert, wird dabei auch untersucht, ob und wie sich die Tests in Browsern bzw. von einem eigenständigen Javascript-Interpreter ausführen lassen.

3.1 Modultest-Frameworks

Zunächst folgen Javascript-Frameworks, mit denen man Modultests schreiben kann.

3.1.1 JsUnit

Abbildung 1: Logo von JS-Unit

JsUnit ist das Javascript-Framework der xUnit-Reihe. Es handelt sich also um eine Portierung des von Kent Beck entworfenen Modultest-Frameworks SUnit. Demzufolge bietet sie sowohl die Möglichkeit, Set-Up- und Tear-Down-Methoden, als auch Behauptungen zu formulieren:

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="jsUnitCore.js"></script>
    <script type="text/javascript" src="myCode.js"></script>
    <script type="text/javascript">
      var counter;
      function setUp() {
        counter = 1
      }
      function testAddition() {
        assertEquals(2, counter + 1);
      }
      function testFalseAddition() {
        assertNotEquals(3, counter + 1);
      }
      function tearDown() {
        delete counter;
      }
    </script>
  </head>
  <body></body>
</html>
```

Abbildung 2: Beispiel-Datei JsUnit

Um den Test-Case auszuführen, bietet JsUnit einen Test-Runner, den man als HTML-Seite in einem beliebigen Browser öffnen kann. In diesem kann man über ein Text-Feld die oben gezeigte Test-Case-Datei referenzieren und ausführen.

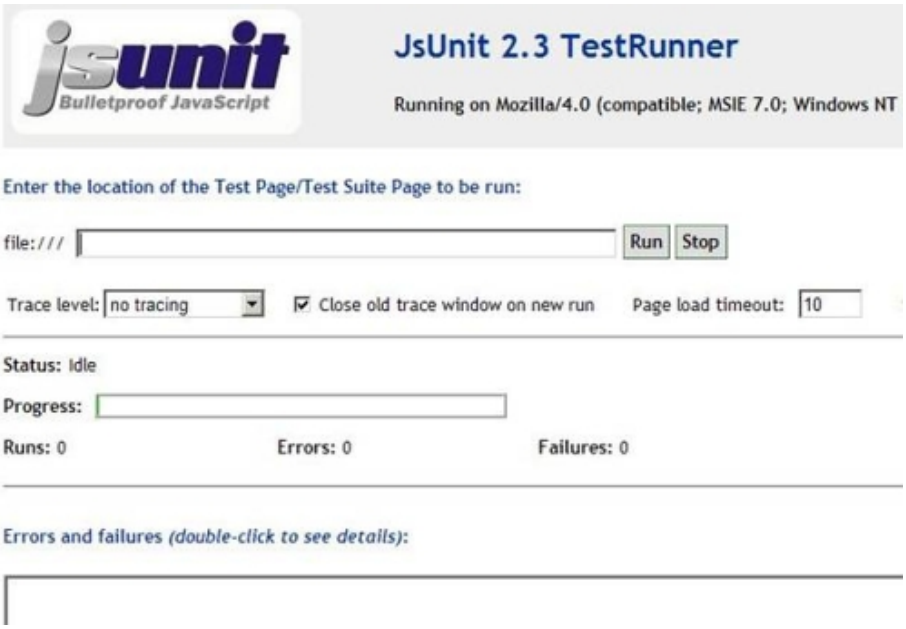

Abbildung 3: JS-Unit Testrunner

JsUnit bietet von Haus aus keine Möglichkeit, den Test-Case in einem Headless- Javascript-Interpreter auszuführen. Dadurch ist eine einfache, automatisierte Ausführung der Tests nicht möglich. Zudem bietet das Framework keine Möglichkeit, die Test-Cases in Test-Suiten zu organisieren. Die Entwicklung von JsUnit wurde inzwischen eingestellt (Vgl. Pivotal-Labs 2012).

3.1.2 YUI Test

Abbildung 4: Logo von YUI

YUI ist ein Framework zur Erstellung von interaktiven Webanwendungen. Teil dieses Frameworks ist die Komponente YUI-Test, mit der sich Modultests durchführen lassen:

```
var simpleTestCase = new Y.Test.Case({
  name: "Simple test",
  setUp: function() {
    this.counter = 1;
  },
  tearDown: function() {
    delete this.counter;
  },
  "1 + 1 is 2": {
    Y.Assert.areEqual(2, this.counter + 1, "1 + 1 should be 2");
  }
});

var simpleTestSuite = new Y.Test.Suite("Simple Test Suite");
simpleTestSuite.add(simpleTestCase);
Y.Test.Runner.add(simpleTestSuite);
Y.Test.Runner.run();

var xml = Y.Test.Runner.getResults(Y.Test.Format.XML);
```

Abbildung 5: Beispiel-Datei YUI

Wie man sehen kann, bietet YUI-Test einige Vorteile gegenüber JsUnit. Zum einen lassen sich die Tests besser beschreiben, da der Test- und der Test-Case-Name eine einfache Zeichenkette sind und keine Funktions- bzw. Dateinamen sein müssen. Es gibt außerdem die Möglichkeit, für fehlschlagende Tests eine Fehlerbeschreibung anzugeben. Die Test-Cases lassen sich zudem nach Belieben in Test-Suiten organisieren. Im Beispiel wird außerdem gezeigt, wie man die Testergebnisse z.B. als XML ausgeben kann. Der YUI-Test-Runner lässt sich also nicht nur innerhalb eines Browser sondern auch von einem Headless-Javascript-Interpreter ausführen (Vgl. Yahoo 2013).

3.1.3 Jasmine

Abbildung 6: Logo von Jasmine

Jasmine ist ein recht neues Modultest-Framework. Genau wie YUI-Test lassen sich Tests leicht beschreiben und gruppieren. Die Tests lassen sich ebenfalls sowohl in Browsern als auch in Headless-Interpretern ausführen. Im Gegensatz zu YUI bricht es aber mit der gewohnten xUnit-Syntax, um die Lesbarkeit der Tests zu erhöhen (Vgl. Pivotal-Labs 2013):

```
describe("A Simple Test-Case", function() {
  var counter;
  beforeEach(function() {
    counter = 1;
  });
  afterEach(function() {
    counter = null;
  });
  it("should be 2 if we add 1 to 1", function() {
    expect(counter + 1).toBe(2);
  });
});
```

Abbildung 7: Beispiel-Datei Jasmine

3.2 Integrationstest-Frameworks

Da Javascript (egal ob server- und/oder clientseitig eingesetzt) vor allem zur Entwicklung von Webanwendungen verwendet wird, sind vor allem Integrationstest-Frameworks entstanden, mit denen man einfach deren Http-Schnittstelle testen kann.

3.2.1 Phantom-JS

Abbildung 8: Logo von PhantomJS

Bei Phantom-JS handelt es sich um einen sogenannten Headless-Browser. So werden Browser genannt, die alle Funktionen eines Browsers bieten, dabei aber über keine graphische Ausgabe verfügen. Sie bieten sich also vor allem für eine programmatische Verwendung an (Vgl. Hidayat 2013).

```
var page = require('webpage').create();
page.open('http://localhost/testWebsite/', function() {
  var title = page.evaluate(function() {
    return document.title;
  });
  console.log(title);
  phantom.exit();
});
```

Abbildung 9: Beispiel-Datei PhantomJS

Diesen Test kann man nun auf der Konsole ausführen:

```
phantomjs /pfad/zum/test.js
```

Abbildung 10: Startbefehl PhantomJS

Dadurch wird der Titel der Testseite auf der Konsole ausgegeben. Auf ähnlich einfache Weise lassen sich nun auch Links auf der Seite ausführen oder Formulare abschicken. Hierdurch lassen sich viele Aspekte einer Seite sinnvoll testen. Leider fehlt die Möglichkeit, den entsprechenden Test in verschiedenen nativen Browsern auszuführen.

3.2.2 Selenium-Driver

Abbildung 11: Logo von Selenium

Selenium-Driver dient ebenfalls dazu, die Http-Schnittstelle einer Webanwendung zu testen. Im Gegensatz zu Phantom-JS implementiert dieses Tool aber keinen Browser sondern ist in der Lage über die sogenannte Web-Driver-Schnittstelle verschiedene native Browser ”fernzusteuern”. Dadurch lässt sich die Webanwendung tatsächlich in der echten Zielumgebung testen. Auf dem Entwicklungsrechner muss dafür lediglich der entsprechende Browser und ein entsprechender Treiber für Selenium-Driver installiert sein. Es gibt hierbei Treiber für Firefox, Internet Explorer, Google Chrome, Safari uvm. (Vgl. Selenium-Project 2013).

```
var webdriver = require('selenium-webdriver');
var driver = new webdriver.Builder()
  .withCapabilities(webdriver.Capabilities.chrome())
  .build();
driver.get('http://localhost/testWebsite/');
driver.getTitle().then(function(title) {
  console.log(title);
});
driver.quit();
```

Abbildung 12: Beispiel-Datei Selenium

3.3 Test Driven Development

Wir haben uns nun einige Frameworks, die grundsätzlich Modul- und Integrationstests mit Javascript ermöglichen. Bislang haben aber zumindest die Modultest-Frameworks noch keine Möglichkeit aufgezeigt, wie man Tests nicht nur automatisch auf der Konsole (z.B. über einen Headless-Browser) sondern auch automatisch in verschiedenen Browsern

ausführen kann. Dies erforderte bislang immer das anlegen einer HTML-Datei, die man manuell in einem Browser öffnet. Hier kommen nun Tools ins Spiel, die genau diese Lücke füllen können.

3.3.1 JsTestDriver

JsTestDriver ist ein Java-Tool, mit dem man Modultests automatisch an Browser schicken kann, um sie ausführen zu lassen. Dazu wird zunächst mit Hilfe des Tools ein Webserver gestartet:

```
java -jar JsTestDriver.jar --port 9876
```

Abbildung 13: Startbefehl JsTestDriver-Server

Nun kann man beliebige Browser über diesen Webserver einfangen, in dem man in ihnen die URL *http://localhost:9876/capture* aufruft.
Anschließend legt man eine Konfigurationsdatei an, die die Webserver-Adresse und die auf ihr auszuführenden Tests definiert.

```
server: http://localhost:9876
load:
  - src/*.js
  - src-test/*.js
```

Abbildung 14: Konfigurations-Datei JsTestDriver

Die Tests können grundsätzlich mit verschiedenen Modultest-Frameworks geschrieben werden. JsTestDriver bietet die Möglichkeit, entsprechende Adapter zu schreiben. Für die gängigen Frameworks existieren diese bereits.
Zu guter Letzt startet man die eigentlichen Tests in den Browsern und lässt das Ergebnis z.B. in eine XML-Datei schreiben:

```
java -jar JsTestDriver.jar --tests all --testOutput result.xml
```

Abbildung 15: Startbefehl JsTestDriver-Tests

Damit kann ein Entwickler nun zum Beispiel zu Beginn seines Arbeitstages einmalig bestimmte Browser auf dem JsTestDriver-Server registrieren und diese dann zum automatisierten Testen verwenden. Damit ist grundsätzlich die Möglichkeit für TDD gegeben. Ein Punkte bleibt allerdings weiterhin offen: Jeder Test-Browser muss auf irgendeine Art und Weise vom Entwickler auf seinem JsTestDriver-Server registriert werden.

3.3.2 Browser-Mocking

Ein Nachteil der Browser-Fernsteuerung mit Hilfe von Selenium-Driver bzw. JsTestDriver ist die lange Ausführungsdauer der Tests. So müssen der zu testende Code, die einzelnen Testfälle und die Testergebnisse zwischen dem Testrunner und dem Browser übermittelt werden. Eventuell greift der Code sogar auf das Document Object Model (DOM) oder die Rendering-Engine des Browsers zu. Je nach Anzahl der Testfälle kann die Ausführungsdauern dabei so weit ansteigen, dass ein effektives TDD nicht mehr möglich ist. Es lohnt sich deshalb, die einzelnen Tests so weit wie möglich ohne solche nativen Browser-APIs zu formulieren. Hier bietet sich das Mocken von einzelnen Komponenten an. Da Javascript das Überschreiben von Systemvariablen und Funktionen erlaubt, gestaltet sich das Mocken der Browser-APIs sehr einfach. Ein Test kann zur Laufzeit beliebige Objekte und Methoden anstelle der Browser-API definieren, die sich genau so verhalten wie es der Tests erwartet. So können sogar teure HTTP-Aufrufe mit der XMLHttpRequest-Komponente (Stichwort AJAX) o.ä. emuliert werden, ohne wirklich die echte Browser-API zu verwenden. Der folgende Code implementiert bzw. überschreibt z.B. die native Browser-Funktion 'alert', mit der normalerweise ein Hinweisfenster im Browser geöffnet wird:

```
// Dieser Aufruf würde innerhalb eines
// Browsers ein Hinweisfenster öffnen
window.alert("Hello world");

var alertedText = null;
window.alert = function(text) {
    alertedText = text;
}

// Dieser Aufruf würde nun lediglich
// die Variable alertedText belegen.
window.alert("Hello world");
```

Abbildung 16: Mocken der 'alert'-Funktion

3.4 Behaviour Driven Development

Das UnitTest-Framework Jasmine ermöglicht es, die Definition der Tests so zu formulieren, dass sie nicht nur für Entwickler sondern auch für Kunden oder für Produktmanager verständlich sind. Wenn man die Tests entsprechend gut formuliert, lassen sie sich wie Prosa-Sätze lesen (Vgl. Pivotal-Labs 2013). Die Testfälle werden aber immer noch als Javascript-Code erfasst. Jasmine verzichtet also im Gegensatz zu BDD-Tools anderer Sprachen auf die Einführung einer eigenständigen DSL.

3.5 Continous Integration

Eine Möglichkeit der Durchführung von CI erscheint mit JsTestDriver zunächst nicht möglich. Es finden sich zwar Anleitungen, wie man sich wiederrum Selenium-Driver zu nutze machen kann, um Browser ferngesteuert im JsTestDriver-Server zu registrieren. Dieser Ansatz ist aber recht umständlich und fragil. Deshalb sind neue Tools entstanden, die die CI Methodik besser unterstützen.

3.5.1 Karma

Abbildung 17: Logo von Karma

Karma ist ein sehr neues Tool, dass einen ähnlichen Ansatz wie JsTestDriver verfolgt, dabei aber zusätzliche Funktionen mit sich bringt, die die Einführung von CI vereinfachen. Grundsätzlich bietet es auch die Möglichkeit, beliebige Modultest-Frameworks zu verwenden und diese auf verschiedenen Browsern auszuführen. Die Integration von neuen Frameworks bzw. Browsern läuft auf Basis von Plugins, die meist von der sehr aktiven Community bereitgestellt werden. So sind inzwischen sehr praktische Plugins entstanden, mit denen man direkt (ohne den fragilen Umweg über Selenium-Driver), über die Web-Driver Schnittstelle Browser fernsteuern kann, um die Tests auf ihnen auszuführen. Außerdem ist Karma auch in der Lage andere Dateien (z.B. Bilder oder HTML-Dateien) an die Test-Browser auszuliefern. Ein weiterer Vorteil gegebenüber JsTestDriver ist, dass Karma in der Lage ist, die konfigurierten Javascript-Dateien zu überwachen und bei jeglicher änderung sofort Tests auszuführen (Vgl Google 2013). Dies erleichtert auch das TDD:

```
module.exports = function(karma) {
  karma.set({
    basePath: './',
    autoWatch: true,
    frameworks: ['jasmine'],
    files: [
      'vendor/*.js',
      'src/*.js',
      'tests/*.js',
      {'web/*.html', included: false}
    ],
    customLaunchers: {
      'Remote-Firefox': {
        base: 'WebDriver',
        config: {
          hostname: 'some.other-server.com',
          port: 4444
        },
        browserName: 'Firefox',
        platform: 'Windows XP',
        version: '21'
      }
    },
    browsers: ['IE', 'Remote-Firefox']
  });
};
```

Abbildung 18: Beispiel-Datei Karma

Nun kann man die Test-überwachung über einen einfache Konsolen-Befehl starten. Dieser Befehl bietet außerdem einen SSingleRunModus, bei dem die Tests nur einmal ausgeführt und Karma direkt wieder beendet wird. Dieser lässt sich für die Integration mit einem CI-Server nutzen:

```
karma start --single-run
```

Abbildung 19: Startbefehl Karma

3.5.2 SauceLabs

Abbildung 20: Logo von SauceLabs

Bei der heutigen Vielfalt an Browsern und Plattformen wäre es sehr aufwändig und wenig effizient, wenn jeder Entwickler bzw. jede Softwarefirma eine vollständige Testumgebung aufbaut (Vgl. Nguyen 2013). Dies hat unter anderem auch der Anbieter SauceLabs erkannt und stellt als Cloud-Service-Lösung gegen eine vergleichbar günstige Nutzungsgebühr beliebige Testsysteme auf Abruf zur Verfügung.

Verwendet man Karma, so reicht die Installation des Plugins karma-sauce-launcher aus, um Tests direkt in der SauceLabs-Cloud auszuführen:

```
module.exports = function(karma) {
  karma.set({
    ...
    sauceLabs: {
      username: 'perprogramming',
      accessKey: '1234567890'
    },
    customLaunchers: {
      'SauceLabs-IE-11': {
        base: 'SauceLabs',
        browserName: 'internet explorer',
        plattform: 'Windows 8.1',
        version: '11'
      }
    },
    browsers: ['SauceLabs-IE-11']
  });
};
```

Abbildung 21: Beispiel-Datei Karma mit SauceLabs

Hierdurch kann nicht nur der Entwickler sondern eben auch der CI-Server die Tests ausführen lassen, ohne sich darüber Gedanken zu machen, woher die verschiedenen Browser und Plattformen kommen.

4 Fallbeispiel

Im Folgenden soll nun ein vollständiges Beispiel einer Javascript-Anwendung und entsprechender Test-Definitionen gegeben werden. Bei der Anwendung handelt es sich um einen sehr reduzierten Taschenrechner, der lediglich in der Lage ist, zwei Zahlen zu addieren. Der Anwendungs-Code glieder sich dabei in folgende Klassen:

4.1 Calculator

Die Klasse Calculator stellt den eigentlichen Taschenrechner dar. Er wird mit einer bestimmten Rechenoperation erzeugt. Zusätzlich erhält er zwei Objekte, die ihm die beiden Operanden zur Verfügung stellen, und ein Objekt, an das er das Ergebnis übermitteln kann:

```
var Calculator = function(
    firstOperantSource,
    secondOperantSource,
    operation,
    resultTarget
) {
    this.firstOperantSource = firstOperantSource;
    this.secondOperantSource = secondOperantSource;
    this.operation = operation;
    this.resultTarget = resultTarget;
}

Calculator.prototype.calculate = function() {
    this.resultTarget.set(
        this.operation.execute(
            this.firstOperantSource.get(),
            this.secondOperantSource.get()
        )
    );
}
```

Abbildung 22: Die Klasse Calculator

4.2 Addition

Die Klasse Addition stellt eine mögliche Operation innerhalb des Taschenrechners dar. Die Klasse erhält zwei Zahlen und liefert deren Summe zurück:

```
var Addition = function() {
}

Addition.prototype.execute = function(firstValue, secondValue) {
    return firstValue + secondValue;
}
```

Abbildung 23: Die Klasse Addition

4.3 HtmlInputOperantSource

Die Klasse HtmlInputOperantSource erlaubt es, einen Operanden aus einem HTML-Input-Element auszulesen. Sie wird dafür mit dem entsprechenden DOM-Element erzeugt und liest den Wert von dessen value-Eigenschaft aus. Anschließend versucht es, den Wert als Integer zu parsen und zurückzugeben:

```
var HtmlInputOperantSource = function(domElement) {
    this.domElement = domElement;
}

HtmlInputOperantSource.prototype.get = function() {
    return parseInt(this.domElement.value);
}
```

Abbildung 24: Die Klasse HtmlInputOperantSource

4.4 HtmlDivResultTarget

Die Klasse HtmlDivResultTarget dient dazu, das Ergebnis einer Berechnung in einem DIV-Element anzuzeigen:

```
var HtmlDivResultTarget = function(domElement) {
    this.domElement = domElement;
}

HtmlDivResultTarget.prototype.set = function(result) {
    this.domElement.innerText = result;
}
```

Abbildung 25: Die Klasse HtmlDivResultTarget

4.5 Unit-Tests

Die Klassen und ihr Verhalten lassen sich zunächst sehr leicht mit entsprechenden Unit-Tests abdecken. Dabei wird der Zugriff auf das DOM (HTML-Input zur Eingabe und

DIV-Element zur Ausgabe) und auf andere Module jeweils einfach komplett durch Mock-Objekte ersetzt:

```
describe("Calculator", function() {
    it("calculates", function() {
        var operation = {
            execute: function(a, b) {
                return [a, b];
            }
        };
        var firstOperantInput = {
            get: function() {
                return 'first';
            }
        };
        var secondOperantInput = {
            get: function() {
                return 'second';
            }
        }
        var resultTarget = {
            result: null,
            set: function(result) {
                this.result = result;
            }
        }
        var calculator = new Calculator(
            firstOperantInput,
            secondOperantInput,
            operation,
            resultTarget
        );
        calculator.calculate();
        expect(resultTarget.result[0]).toBe('first');
        expect(resultTarget.result[1]).toBe('second');
    });
});
```

Abbildung 26: Test der Klasse Calculator

```
describe("Addition", function() {
    it("is 3 if you add 1 to 2", function() {
        var operation = new Addition();
        expect(operation.execute(1, 2)).toBe(3);
    });
});
```

Abbildung 27: Test der Klasse Addition

```
describe("HtmlInputOperantSource", function() {
    it("loads a value from a HTML input", function() {
        var htmlInput = {value: 1};
        var operantSource = new HtmlInputOperantSource(htmlInput);
        expect(operantSource.get()).toBe(1);
    });
    it("can only read valid integers", function() {
        var htmlInput = {value: 'foobar'};
        var operantSource = new HtmlInputOperantSource(htmlInput);
        expect(operantSource.get()).toBeNaN();
    });
    it("will make floats to next lower integer", function() {
        var htmlInput = {value: 2.7};
        var operantSource = new HtmlInputOperantSource(htmlInput);
        expect(operantSource.get()).toBe(2);
        htmlInput.value = 2.1;
        expect(operantSource.get()).toBe(2);
    });
});
```

Abbildung 28: Test der Klasse HtmlInputOperantSource

```
describe("HtmlDivResultTarget", function() {
    it("sets a result to a HTML div", function() {
        var htmlDiv = {};
        var resultTarget = new HtmlDivResultTarget(htmlDiv);
        resultTarget.set(1);
        expect(htmlDiv.innerText).toBe(1);
        resultTarget.set(2);
        expect(htmlDiv.innerText).toBe(2);
    });
});
```

Abbildung 29: Test der Klasse HtmlDivResultTarget

4.6 Integrationstest

Natürlich ist es aber auch möglich, eine echte HTML-Seite zu definieren, die die Komponenten integriert. Diese Seite ließe sich dann, wenn auch langsam, mit Hilfe eines Webdrivers abfragen und testen.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Calculator</title>
        <script src="Addition.js"></script>
        <script src="Calculator.js"></script>
        <script src="HtmlInputOperantSource.js"></script>
        <script src="HtmlDivResultTarget.js"></script>
    </head>
    <body>
        <div>
            <input type="text" id="first-operant-source" /> +
            <input type="text" id="second-operant-source" /> =
            <div id="result-target"></div>
            <button id="calculate">Calculate</button>
        </div>
        <script>
            var Calculator = new Calculator(
                new HtmlInputOperantSource(
                    document.getElementById(
                        'first-operant-source'
                    )
                ),
                new HtmlInputOperantSource(
                    document.getElementById(
                        'first-operant-source'
                    )
                ),
                new Addition(),
                new HtmlDivResultTarget(
                    document.getElementById('result-target')
                )
            );
            document.getElementById('calculate').onclick = function() {
                Calculator.calculate();
            };
        </script>
    </body>
</html>
```

Abbildung 30: Integration aller Komponenten

5 Fazit

Insgesamt lässt sich sagen, dass für die Sprache Javascript sehr viele, sehr mächtige Test-Frameworks und -Tools entstanden sind. Diese ermöglichen es, sowohl clientseitige als auch serverseitige Anwendungen großer Komplexität zu testen und somit für eine stabile Entwicklung zu sorgen.

Auch moderne Testmethodiken wie das TDD, das BDD und das CI werden verfolgt, wenn auch im Falle von BDD noch nicht so weitgreifend wie in anderen Sprachen.

Dafür schaffen es die Frameworks, den besonderen Herausforderungen der Sprache Javascript gerecht zu werden. So lässt sich entweder durch die Browser-Fernsteuerung oder durch das weitgehende Mocking von nativen APIs auch Browser-übergreifende Tests realiseren bzw. Code entwicklen, der auf unterschiedlichsten Plattformen stabil funktioniert.

6 Literaturverzeichnis

eBay 2011
eBay: *Announcing ql.io.* http://www.ebaytechblog.com/2011/11/30/announcing-ql-io/. Version: 2011. – [Online; Stand 07. Dezember 2013]

Google 13
Google: *Closure Tools.* https://developers.google.com/closure/library/. Version: 13. – [Online; Stand 07. Dezember 2013]

Google 2013
Google: *Karma - Configuration File.* http://karma-runner.github.io/0.10/config/configuration-file.html. Version: 2013. – [Online; Stand 07. Dezember 2013]

Hidayat 2013
Hidayat, Ariya: *Quick Start | PhantomJS.* http://phantomjs.org/quick-start.html. Version: 2013. – [Online; Stand 07. Dezember 2013]

Ihlenfeld 2010
Ihlenfeld, Jens: *Aves Engine - Javascript-Engine für Browserspiele.* http://www.golem.de/1004/74719.html. Version: 2010. – [Online; Stand 07. Dezember 2013]

Johansen 2010
Johansen, Christian: *Test Driven JavaScript Development.* Addison-Wesley Professional, 2010

Kleivane 2011
Kleivane, Tine F.: *Unit Testing with TDD in JavaScript*, Norwegian University of Science and Technology, Diplomarbeit, 2011

Nguyen 2013
Nguyen, Lauren: *Sauce Labs Adds Expanded JavaScript Unit Testing Capabilities to its Cloud Testing Platform.* http://sauceio.com/index.php/2013/09/sauce-labs-adds/. Version: September 2013. – [Online; Stand 14. November 2013]

Pivotal-Labs 2012
Pivotal-Labs: *JsUnit.* https://github.com/pivotal/jsunit. Version: 2012. – [Online; Stand 07. Dezember 2013]

Pivotal-Labs 2013
Pivotal-Labs: *Jasmine.* http://pivotal.github.io/jasmine/. Version: 2013. – [Online; Stand 07. Dezember 2013]

Selenium-Project 2013
Selenium-Project: *Selenium Webdriver.* http://www.seleniumhq.org/docs/03_webdriver.jsp. Version: 2013. – [Online; Stand 07. Dezember 2013]

Trostler 2013
Trostler, Mark E.: *Testable JavaScript.* O'Reilly Media, 2013

Wikipedia 2013a
Wikipedia: *Integrationstest.* http://de.wikipedia.org/wiki/Integrationstest. Version: 2013. – [Online; Stand 07. Dezember 2013]

Wikipedia 2013b
Wikipedia: *JavaScript.* http://de.wikipedia.org/wiki/JavaScript. Version: 2013. – [Online; Stand 07. Dezember 2013]

Wikipedia 2013c
Wikipedia: *Kontinuierliche Integration*. `http://de.wikipedia.org/wiki/Kontinuierliche_Integration`. Version: 2013. – [Online; Stand 07. Dezember 2013]

Wikipedia 2013d
Wikipedia: *Softwaretest*. `http://de.wikipedia.org/wiki/Softwaretest`. Version: 2013. – [Online; Stand 07. Dezember 2013]

Wikipedia 2013e
Wikipedia: *Testgetriebene Entwicklung*. `http://de.wikipedia.org/wiki/Testgetriebene_Entwicklung`. Version: 2013. – [Online; Stand 07. Dezember 2013]

Yahoo 2013
Yahoo: *Test - YUI Library*. `http://yuilibrary.com/yui/docs/test/`. Version: 2013. – [Online; Stand 07. Dezember 2013]