Norwegian University of
Science and Technology

Unit Testing with TDD in JavaScript

Tine Flåten Kleivane

Master of Science in Computer Science
Submission date: July 2011
Supervisor: Terje Rydland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Outline

Techniques as TDD makes testing the driving force in design, documentation, maintainability and code quality. In dynamic languages such as JavaScript, which completely lacks compile time warnings, testing is the glue that holds the big applications together. This thesis sets out to explore the techniques and tools available for doing unit testing with TDD in JavaScript.

Assignment given: 31. January 2011

Supervisor: Terje Rydland

Abstract

JavaScript has gained increased usage and attention the last years, but development and testing methods is still lagging behind.

To mitigate some of these issues, this thesis brings together unit testing and JavaScript, using test-driven development as a methodology. Through exploration of these topics, the differentiators in a unit testing framework are considered. Existing frameworks are also discussed and how the terminology in JavaScript differs from traditional xUnit family frameworks.

By creating and running a set of four test cases, both the general and unique features of JavaScript are tested in hand-picked frameworks, which were decided through an evaluation process. One of the contributions is based on this; a recommendation for a minimum set of test library features for a JavaScript unit testing framework.

Various factors were found to differentiate the frameworks, and so the thesis also provides a DISCOVERY test case to emphasize some of these aspects. This test case provides practitioners with a quick option for learning a new framework. The set of test cases can be applied to new frameworks to assess their functionality.

As the thesis explores an area with little current research, suggestions for further work present several topics, ranging from system level JavaScript testing to quantitative studies building on the set of test cases.

Preface

This report represents the work done in my master's thesis (Master of Science) in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The duration of the thesis has been from February 2011 to July 2011 and my supervisors have been Assistant Professor Terje Rydland at NTNU and Torstein Nicolaysen at BEKK Consulting.

I owe a great thanks to both my supervisors for support and feedback. Especially to Torstein, who has patiently endured questions and provided both encouragement and criticism, raising the bar of my achievements and the thesis in general.

Many thanks also to my team of proofreaders, Ruben, Nils and my dad, and fellow students at Fiol and the Sahara offices for coffee, lunches and great company.

And to my family, eternal gratitude for continuous support during 17 years of school, and encouragement on all other arenas.

And to NTNU and Trondheim, it has been five great years.

Tine Kleivane

Tine Flåten Kleivane

Skien, 7. July 2011

Contents

Abstract **i**

Preface **i**

List of tables **vii**

List of figures **ix**

List of listings **xi**

Glossary **xiii**

Abbreviations and acronyms **xv**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Research question 3
- 1.3 Contributions 3
- 1.4 Outline 4

2 Background **5**
- 2.1 Unit testing 5
  - 2.1.1 Unit testing frameworks 6
  - 2.1.2 Integration testing 8

2.2 Test-Driven Development . . . 9
2.2.1 The rules of TDD and their implications . . . 9
2.2.2 Programming with TDD . . . 11
2.2.3 Research results . . . 12
2.2.4 Behavior-Driven Development . . . 14
2.3 JavaScript . . . 16
2.3.1 History . . . 17
2.3.2 The language . . . 22
2.4 Summary . . . 33

3 Research method 35
3.1 Literature study . . . 35
3.2 Research methods . . . 36
3.2.1 Qualitative methods . . . 36
3.2.2 Quantitative methods . . . 37
3.2.3 Feasibility prototyping . . . 37
3.3 Adopting demo and proof of concept - methods . . . 38
3.3.1 Process description . . . 40
3.4 Limitations . . . 43
3.5 Summary . . . 43

4 An introduction to JavaScript and testing 45
4.1 Characteristics affecting tests . . . 46
4.2 Vocabulary . . . 51
4.3 Examining a JavaScript framework . . . 52
4.4 Summary . . . 61

5 State of the art 63
5.1 Jasmine . . . 63
5.2 JSpec . . . 64

5.3 JsTestDriver . . . 65
5.4 JsUnit . . . 66
5.5 nodeunit . . . 68
5.6 QUnit . . . 68
5.6.1 FuncUnit . . . 69
5.6.2 Pavlov . . . 70
5.7 Screw.Unit . . . 71
5.8 Sinon.JS . . . 71
5.9 TestSwarm . . . 72
5.10 YUI Test . . . 73
5.11 Others . . . 74
5.12 Feature mapping . . . 75
5.13 Summary . . . 80

6 Results 81
6.1 Test cases . . . 81
6.1.1 FizzBuzz . . . 82
6.1.2 Alias . . . 83
6.1.3 Twitter . . . 84
6.1.4 DOM manipulation . . . 84
6.2 Execution . . . 85
6.2.1 Standalone Jasmine . . . 87
6.2.2 Jasmine and Sinon.JS . . . 89
6.2.3 JSpec . . . 91
6.2.4 JsTestDriver and Sinon.JS . . . 94
6.2.5 Standalone QUnit . . . 96
6.2.6 QUnit and Sinon.JS . . . 97
6.2.7 YUI Test . . . 99
6.3 Summary . . . 102

7 Discussion **105**

7.1 Limitations of the research design and material 105
7.2 Personal experiences 106
7.2.1 Difficulties 107
7.2.2 Experiences with TDD in JavaScript 108
7.2.3 Effects from TDD in JavaScript 111
7.3 Level of testing 112
7.4 Reviewing JavaScript characteristics 114
7.5 Reviewing frameworks characteristics 121
7.5.1 Redefining framework characteristics 131
7.6 Recommended test library features for unit frameworks 132
7.6.1 A Discovery test case 133
7.7 Testing terminology for JavaScript 136
7.8 Summary 137

8 Conclusion **139**

8.1 Further work 144

Appendices **149**

A JavaScript support functions **149**

B Source code **151**

B.1 Jasmine timing 151
B.1.1 Abstracting away time 154

C Encapsulating timeouts **159**

Bibliography **161**

List of Tables

2.1 Summary of data findings by Shull et al.[1]. 13
2.2 Falsy values in JavaScript. 31

5.1 General characteristics of combined frameworks. 77
5.2 Release, support and community characteristics of combined frameworks. 78
5.3 Library features of combined frameworks. 79

List of Figures

2.1 Interaction with the SUT i unit testing frameworks. 7
2.2 Means of interaction with integration testing. 8
2.3 Best viewed with Netscape Navigator and IE. 19

3.1 Project method: step by step. 40

4.1 Execution of tests in the browser. 54

5.1 A Jasmine example. 64
5.2 The JsUnit test runner. 67
5.3 Results from TestSwarm. 73

6.1 The FizzBuzz test case passed in Jasmine. 87
6.2 The DOM test case in with JsTestDriver and Sinon. 95
6.3 The test runner in QUnit. 98
6.4 The Console module in the test runner outputting YUI test results. 101

7.1 Result of Listing 7.5 implemented and executed in the different test runners. 124

Listings

2.1 A JavaScript function and method. 22
2.2 A Java method and function. 23
2.3 An example of prototypal inheritance in JavaScript. 25
2.4 An example of functional inheritance in JavaScript. 25
2.5 Inheritance in Java. 26
2.6 Closures in JavaScript. 27
2.7 The three ways of creating a global variable. 29
2.8 Decimal floating point arithmetic in JavaScript. 31
2.9 Equality in JavaScript with the double equals operator. 32
4.1 JsTestDriver TDD syntax. 55
4.2 Jasmine BDD syntax. 55
4.3 JSpec BDD DSL syntax. 56
4.4 Stubbing on a function for behavior verification. 58
5.1 An example of the JSpec BDD DSL. 65
5.2 An example configuration of `JsTestDriver.config` 66
5.3 Generative row tests and stubbing of specs in Pavlov. 70
6.1 Pseudocode for test case 1: "FizzBuzz". 82
6.2 Pseudocode for test case 2: "Alias". 83
6.3 Pseudocode for test case 3: "Twitter". 84
6.4 Pseudocode for test case 4: "DOM". 85
6.5 Part of the setup method in Jasmine 89

6.6 Invoking endGame from the setTimeout function through the global scope. 90
6.7 The first test in the FizzBuzz test case in JSpec. 92
6.8 Setup with the use of JSpec's `fixture` method. 93
6.9 Insertion of a `div` element in setup before each test method. . 95
6.10 Creating a common `this` environment for the fake timers. . . 99
7.1 Excerpt from a w3schools tutorial on Ajax. 109
7.2 Pseudocode for test case 4: "DOM" with line 2 commented out.115
7.3 Testing the changed DOM test case in QUnit. 115
7.4 Examples of testing with fake timers and content of parameter. 118
7.5 A test to reveal the lifecycle of a framework. 123
7.6 Jasmine mocks. 128
7.7 Sinon spy. 129
7.8 Sinon mocks[60]. 130
7.9 A test case to get to know the features of a new framework. . 135
A.1 An implementation of super in JavaScript. 149
B.1 Tests for Alias in Jasmine. 151
B.2 Source code for Alias in Jasmine. 153
B.3 Tests for Alias with time abstracted away. 154
B.4 Source code for Alias with time abstracted away. 156
C.1 Encapsulating time. 159

Glossary

continuous integration (CI) is a continuous process of applying quality control, usually through automatic building and execution of tests. 68, 73

system under test (SUT) refers to the system currently being tested. Sometimes referred to as CUT (code under test or class under test). 6

Abbreviations and acronyms

API Application Programming Interface. 6, 7, 49, 50, 60, 129, 135

BDD Behavior-Driven Development. 3, 14–16, 33, 55, 56, 63–65, 69, 70, 74, 76, 79, 86, 122, 126, 134, 135, 137, 138

CI continuous integration. 49, 66, 68, 73, 75, 106, 121, 131

CMS content management systems. 53

CSS Cascading Style Sheet. 20, 106, 114

DOM Document Object Model. 18, 20, 21, 46, 48, 52, 53, 60, 68, 84, 88, 91, 95, 99, 114–116, 123, 126

DSL domain-specific language. 56, 65, 76, 91, 126

IDE Integrated Development Environment. 1, 2, 51, 54, 65, 110, 111, 125

IE Internet Explorer. 17–21, 43, 48

LOC line of code. 12, 48

MMF minimum marketable features. 16

MTTF mean time to fix. 12, 13

OS operating system. 2, 48, 54, 106

SRL software readiness level. 39

SUT system under test. 6–8, 60, 129

TDD Test-Driven Development. 1–5, 9–16, 33, 36–40, 42, 46, 47, 51–53, 55, 56, 58–61, 64, 70, 71, 79, 82, 86, 102, 106, 108–112, 118–120, 122, 125, 126, 135, 137–139, 141, 142

UI User Interface. 46, 50, 60, 73, 106, 135

W3C World Wide Web Consortium. 17, 19, 20

XHR XmlHttpRequest. 59, 85, 88, 94, 99, 100, 106, 109, 110, 113, 120

Chapter 1

Introduction

JavaScript has changed from a small language for blinking web pages to a full-fledged application programming language, available on client and server. This has happened without the support of a traditional Integrated Development Environment (IDE), and traditional testing and development methods. This support is still lacking, but as the reach of the language widens, the necessity for supporting technologies and frameworks grow. This thesis sets out to discover and bridge some of the ground between JavaScript and unit testing using Test-Driven Development (TDD). By utilizing techniques described in TDD, the development and design process can be made more iterative and the resulting application more correct, easier to maintain and able to produce more business value.

1.1 Motivation

TDD has been around since the dawn of programming[2], but was "rediscovered" by Kent Beck and made popular by his book "Test-Driven Development By Example"[3] in 2003. TDD as a design technique encourages and helps the developer to[2]

- Be explicit about the implementation scope.
- Simplify design.
- Grow confidence in functionality as the source code is expanded or changed.
- Separate logical design from physical design, and physical design from implementation.

It can also provide a common platform for clear communication among developers and other parties of interest, as well as being a valuable source of documentation.

To realize and perform TDD effectively, there is a dependency on frameworks. Efficient testing is easier achieved in IDEs like Eclipse, VisualBasic and IntelliJ where plugins can be run as a part of the build process or at the push of a button[1]. Currently there are no *de facto* standard either for JavaScript IDEs or JavaScript testing frameworks.

An additional challenge with JavaScript is the inherent problem of multiple environments. JavaScript runs in different browsers with different versions, running different operating systems (OSs). If there are dependencies to other JavaScript libraries, both these and their versions must also be taken into account.

JavaScript is a dynamic language[2], so developers with experience from static languages can find that the structured approach that TDD represents will manifest in better design. TDD also has a number of psychological effects, like reduced anxiety, a higher level of communication and change in team dynamics. The TDD process and effects are explained in Section 2.2.

[1] Examples of common frameworks are JUnit for Java and NUnit for .NET languages.
[2] JavaScript is explained further in Section 2.3.

Multiple JavaScript libraries exists for unit testing, TDD and Behavior-Driven Development (BDD)[3]. Choosing the best fit is important to ease the development and testing process. Tools already exist to help developers and there is no need to duplicate existing technology or make testing harder than necessary.

Motivated by the returns from utilizing TDD and how the techniques can help the development, it is beneficial to explore what features are necessary in a JavaScript TDD framework. This way, developers can easier choose the right tool for their projects and start to realize these benefits.

1.2 Research question

On the background of the problem outline given in the section above, the following research questions were defined

1. What available frameworks exist for unit testing with TDD in JavaScript?
2. Which test features are recommended for a JavaScript unit testing framework?
3. What effect does TDD have on JavaScript development?

1.3 Contributions

The most important contributions in this thesis are:

- A recommended set of test features for a JavaScript unit testing framework.

[3]BDD originates from TDD method and is explained in Section 2.2.4.

- A DISCOVERY test case created to quickly gain a lot of information about a new framework.
- A set of differentiator in frameworks .
- A clarification on terminology in JavaScript testing versus xUnit frameworks.
- A set of test cases that can be used to assess new frameworks.

1.4 Outline

Chapter 1 explains the motivation behind the thesis, presents the research questions and gives the main contributions of the work.

Chapter 2 introduces background material on JavaScript, unit testing and TDD.

Chapter 3 discusses and justifies the research method used in the thesis.

Chapter 4 introduces the combination of JavaScript and TDD, decomposes the existing frameworks and establishes a common ground for discussion in the following chapters.

Chapter 5 looks at the state of the art in JavaScript testing frameworks and compares central characteristics.

Chapter 6 explains the result from applying the research method on selected frameworks.

Chapter 7 discusses the findings of Chapter 6 with regards to the initial research questions and the information from Chapter 4.

Chapter 8 concludes the findings and looks to further research on the topic.

Chapter 2

Background

2.1 Unit testing

Unit testing have been around since the 1970's and was first introduced by Kent Beck[1] in the Smalltalk language. Today, the concept has been adapted to a myriad of other languages. The definition of a unit test is as follows[4]:

> A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function.

There are also several properties a unit test must adhere to[4]:

- It should be automated and repeatable.
- It should be easy to implement.
- Once it's written, it should remain for future use.
- Anyone should be able to run it.

[1]Who later rediscovered TDD, as described in Section 2.2.

- It should run at the push of a button.
- It should run quickly.

With a unit test, the system under test (SUT) would be very small and perhaps only relevant to developers working closely with the code[5]. A unit test should only exercise; logical code, code that contains branches, calculations or in other ways enforces some decision-making. Simple property getters and setters in Java are examples of non-logical code.

2.1.1 Unit testing frameworks

Unit testing frameworks help developers write, run and review unit tests. These frameworks are commonly named *xUnit frameworks*[2] and share a set of features across implementations[3, 4]:

- An assertion mechanism to decide the outcome of a test, usually an variant of `assertEquals(expected, actual)`.
- Fixtures for common objects or external resources among tests.
- A way of testing for exceptions.
- A representation of individual tests and test cases[3] to help structure test assets.
- An executable to run and review single tests or groups of tests.

Figure 2.1 shows the automation strategy of the xUnit frameworks. Tests are hand-scripted on through an Application Programming Interface (API) on a unit level. This gives more robust and maintainable tests that even can

[2]Where the x represents the language it is created to test.
[3]Test cases, also called test suites, are a way of representing a collection of tests.

be prebuilt. The tools required are simple and cheap, but require more skill to apply and an existing API. Looking at the front of Figure 2.1 other types of automation are given. Both scripted and recorded UI tests are available with the Selenium[6] tool. Also FuncUnit[4] is in this category. These tools work primarily on a system granularity level.

The recorded API testing exercises the SUT and logs interesting interaction points, and compares these with earlier runs or expected results. Currently, not many tool use this strategy. Recorded UI tests are performed by robot users and are currently very unstable and give small returns.

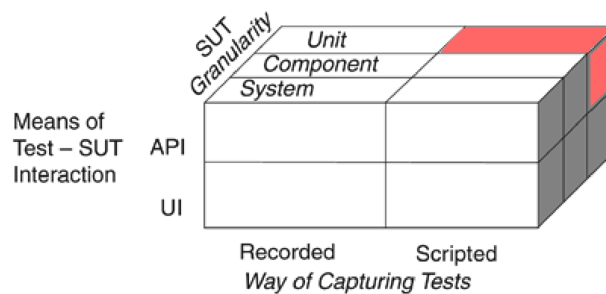

Figure 2.1: xUnit framework automation strategy and SUT granularity. Figure adapted from Meszaros[5].

Unit testing have a surrounding set of test patterns that help enforce the desirable properties described in introduction to Section 2.1. A small set of these are explained below:

- Fake Object : replace a component that the system under test depends on, with a light-weight implementation.
- Test Spy : capture the indirect output calls[5] for later verification.

[4]Described in Section 5.6.1.
[5]Examples of indirect output is number of calls and arguments in the call.

- Implicit Setup : build a common fixture for several tests in the setup-method.

A more complete terminology and description of xUnit test patterns can be found in Gerard Meszaros' "xUnit Test Patterns: Refactoring Test Code"[5].

2.1.2 Integration testing

This form of testing occurs on a higher level than unit testing, as demonstrated by Figure 2.2. The definition of the term is as following[4]:

> Integration testing means testing two or more dependent software modules as a group.

With this form of testing, an error in the underlying units will be cascaded up to the user or programmer with little clue of its originator. In general, integration testing can be said to exercise many units of code to evaluate on or many aspects, while a unit test will exercise a single unit in isolation.

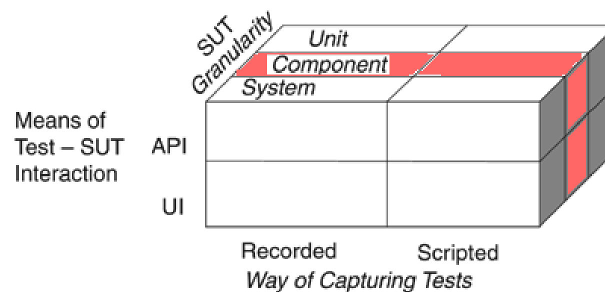

Figure 2.2: Integration testing with the component as SUT. Note that unit testing resides on a level of finer granularity. Figure adapted from Meszaros[5].

Integration testing can also be called *component test* as it verifies that some larger component provides an expected service, either through an API or UI. It is favorable that integration tests share the properties of unit testing with regards to automatization and repeatability, but this is not always achieved.

Many people confuse integration testing with the full system test. Testing on a system level through the UI or API is a *customer test* or *acceptance test* that validates the final deliveries of the system. Referring to Figure 2.2, this would reside on system level granularity.

2.2 Test-Driven Development

TDD is a development methodology that has gained more attention in the later years as different methodologies within the software business has emerged. Dissecting the name, *development* suggests a full-fledged process with analysis, logical and physical design, implementation, testing, review, integration and deployment, and *test-driven* implies how concrete, automated tests should drive the development process[7]. TDD also has the nickname *test-first programming*.

2.2.1 The rules of TDD and their implications

There are two imperatives within TDD[3]:

- Write new code only if an automated test has failed
- Eliminate duplication

These rules are simple, but generate complex individual and group behavior. As a programmer, the rules imply that you must write your own tests, as you cannot wait for someone else to do it. It also puts pressure on

the development tools to provide rapid response to even small changes. The design must be loosely coupled to make testing easy and it must be done organically to support decision making between tests.

In addition, the two imperatives dictate an order when programming [3]

Red : Write a test that does not work, perhaps does not even compile.

Green : Make the test work quickly, committing whatever sins necessary in process.

Refactor : Eliminate all duplication created in merely getting the test to work.

The implications of the imperatives increases as the defect density decreases; dedicated testers can shift from reactive to proactive work and estimation is easier as many bugs are taken out early. The "ultimate goal" in TDD can be seen as shipping new software every day with the confidence that it will work.

From this we can observe that TDD is an *outside-in* methodology. The motivation for undertaking this methodology, and work from the known outside domain and towards a programmatic solution, is to manage anxiety when programming[3]. Anxiety makes for less communication and feedback, which in many cases is exactly what is needed. When doing TDD, advancing in as small steps as necessary reduces this anxiety, while still knowing that the code previously written is not impacted. This can also be seen as a form of risk management for software development. It gives the programmer and a team more confidence; in themselves, each other and the produced code. Other benefits are tests as documentation, lower defect injection rate, decoupled design, openness to new business possibilities, easier maintenance and scope control. The latter is important as programmers tend to "gold plate" the code by adding *nice-to-have* features or designing the code for a case

not (yet) presented. By using TDD, the design follows from the refactoring phase and the code is built in such a way that it can easily be extended when a new case is presented.

2.2.2 Programming with TDD

The goal of TDD is described by Ron Jeffries as *"Clean code that works"*[3]. The red/green/refactor - mantra can be split up according to this goal, to allow the programmer to focus on the different tasks. The Green phase solves the *that works* while Refactor deals with *clean code*. The split allows a programmer to learn along the way and still get the task done.

It is important to keep in the red/green/refactor - rhythm to achieve the steady progress. This means that a small round trip time can be the right way to go, even if progress is achieved in tiny steps. If the steps taken are too big, it can be difficult to trace an introduced bug. Also if it is not obvious how to solve a problem, dividing it into even smaller tests, makes progress slow and steady as opposed to a big and demotivating upfront design effort. The programmers' experience, both with TDD and the problem will decide the size of the tests and the size of the refactoring steps.

To help the progress, TDD has adopted and adapted a set of test and design patterns that fits into the different phases. These patterns provide general solutions that can be adapted to the task at hand. Examples of patterns are:

- One to Many: if programming with a collection of objects, first make the test run with a single object, then implement the collection.

- Value Object: in an object that is widely shared and where state is unimportant, all methods on the object should return a new object.

- Extract Method: to make a complicated method easier to read, extract a small part of it into a separate part.

More patterns can be found in Kent Beck's "Test-Driven Development By Example"[3].

Not all programming tasks are as easily solved using TDD. Software security, concurrency, shared objects, performance and user interfaces are domains where the automated tests of TDD not yet works as a design driver. There are also challenges when adopting TDD in the middle of a project, TDD on enterprise level and when the dependency on third party code is heavy[3].

2.2.3 Research results

The research community has not yet converged on the effects TDD has on a project. There are many studies, but the differences between them, team programming experience, attitude, knowledge and support by customer or management, makes comparisons difficult.

A recent study by Forrest Shull et al.[1] summarized a range of earlier studies as well interviewing expert Grigori Melnik from Microsoft. The three areas of interest were delivered quality, internal quality and productivity. Delivered quality were taken from metrics like mean time between failures and time spent for quality assurance, as they are all pointing towards external quality. Internal quality was from object-oriented structure metrics like cohesion and coupling as well as code complexity and code density metrics. Evidence of productivity was measuring development or maintenance effort, effort per line of code (LOC), or effort per feature. The compared results are summarized in Table 2.1.

The practitioner Melnik argues on delivered quality, as TDD does not replace dedicated testers, but leaves them free to look for more serious bugs.

He also claims TDD reduces the mean time to fix (MTTF) bugs, which is a metric missing in the studies. Regards to internal quality, he strongly disagrees, having experienced first hand an increase in quality on a project after introducing TDD. Team maturity is pointed out as one of the main confounding factors, making the comparison difficult between research studies and a mature team at work.

On productivity, earlier studies are sending inconsistent messages, and such no certain evidences are found. Melnik points to the length of the studies as the learning curve can have a heavy impact on productivity, and argues that maintenance of the code is a missing metric. His experience suggested that TDD created less code, which also was easier to maintain, and lowered the MTTF.

Janzen[8] found that TDD projects have lower code complexity and higher test volume and coverage, but were unable to link this to lower coupling and higher cohesion. Furthermore other studies[9] points out less quantifiable benefits such as helping communication between developers, testers, clients and other parts of the business. Janzen also found moderate evidence that programmers who learned TDD, kept some practices when doing non-TDD projects, indicating a methodology appealing to developers.

Drawbacks mentioned are maintenance of both production and test code and the difficulties of getting the methodology "right"[10]. The latter factor is increased by a survey by Aniche and Gerosa[11], where 44% of the

Dimension	**Findings**
Evidence of delivered quality	Moderately positive evidence
Evidence of internal quality	No special effect
Evidence of productivity	No negative effects

Table 2.1: Summary of data findings by Shull et al.[1].

respondents says that they forget to refactor code. With reference to the Section 2.2.1, this would imply only *code that works*, which undermines the TDD design strategy. In the study by Aniche and Gerosa, there are no correlation between experience in TDD and forgetting the refactoring step. This goes against expert Melnik, who saw team maturity as a heavy influencer on how well the team managed.

Software experts Kent Beck and Robert Martin have both fronted for the method[3, 12], arguing that it gives a low fault injection rate and lower defect density.

In general, research shows no absolutes when it comes to TDD although experts vouch for the methodology. No studies were found that researched on the long-term consequences advocated by TDD such as extensibility, reusability, and maintainability.

2.2.4 Behavior-Driven Development

BDD was first introduced by Dan North in his article "Introducing BDD"[13] and is a methodology that evolved from TDD practices. BDD is a design technique centering on user stories, which should be written in a language understood by non-programmers. The latter fact allows business executives, tester, users and other stakeholders to take a more active part in the development. The user stories are centered on the syntax:

> Title (one line describing the story)
>
> Narrative:
> As a [role]
> I want [feature]
> So that [benefit]
>
> Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context] And [some more context]...
When [event]
Then [outcome] And [another outcome]...

As with TDD, BDD is an *outside-in* design technique. In BDD the stories are written first, then verified and prioritized by domain-experts, users and other non-technical stakeholder. The programmer then creates the code to accomplish the described stories.

BDD has three core principles[14]:

- Business and technology should refer to the same system in the same way.
- Any system should have an identified, verifiable value to the business.
- Up-front analysis, design and planning all have a diminishing return.

The first core principle is solidified through the user story, written as seen above, in a non-technical language. This notion of natural language follows many BDD frameworks, even on a code level, as will be seen in Chapter 4 to 6.

The second core principle of adding business value can be seen in the story as the *So That [benefit]* part. Advice is given to ask recursively why the *I want [feature]* is necessary until (max. 5 times) one of the following business values is seen[15]:

- Protect revenue.
- Increase revenue.
- Manage cost.
- Increase brand value.

- Make the product remarkable.

- Provide more value to your customers.

This allows stakeholders to more easily prioritize and helps focusing on the minimum marketable features (MMF) that will give the most value.

The third core principle has the same meaning as the TDD process. A small as possible upfront design is done, rather the design is emerging through testing and refactoring.

Together the principles help developers mitigate the risks of creating extra features or creating the wrong features. As of today, BDD has not yet received much attention in the research community, but this may relate to its age and because the research on its originating method, TDD, is still sparse and with contradictory results.

2.3 JavaScript

JavaScript popularity has grown in the recent years and it has been claimed as the worlds most popular programming language[16]. Almost all computers today have at least one JavaScript interpreter installed on their machine, most commonly inside the browser. Currently browser vendors boost JavaScript engines as the most important feature during product launches. Also with the `introduction` of HTML5 the JavaScript APIs are being standardized and are now even more powerful. "Classic" programmers has earlier shunned it as a lesser language, but training and innovativeness have opened the eyes of a new generation of JavaScript developers. JavaScript has been called the world's most misunderstood programming language[17] due to lousy implementations, amateur programmers and design errors, but this is about to change.

2.3.1 History

JavaScript was developed at Netscape by Brendan Eich[18], hired from Silicon Graphics. He wanted to write a Scheme[6] interpreter for the Netscape browser. Netscape gave the project a go, but with one exception, Scheme was not a common programming language, so they wanted it to look more like Java. Eich then took the functional model from Scheme, the syntax from Java and the prototype model from Self[7][20]. It is said that Eich was given ten days to complete the project, so not all the ideas implemented was good ones. The resulting language was called *LiveScript*.

At this point Netscape and Sun decided to cooperate to ensure a more competitive environment against Microsoft, who competed with its Internet Explorer (IE). Sun intended Java as a browser technology and LiveScript became one of the most important problems in the merger. Netscape did not want to kill the language, and it is said the someone as a joke suggested the name change to JavaScript[21]. The suggestion hit, and even though Sun had nothing to do with the development of LiveScript and that it was nothing like Java, they still got ownership of the JavaScript trademark and Netscape was given an exclusive license. Microsoft responded by reverse-engineering the JavaScript engine in the Netscape browser, creating their own version named JScript.

Netscape decided to try to standardize the project to avoid others extending their product, and approached World Wide Web Consortium (W3C). The W3C was not happy with Netscape, the browser vendor had made a lot of unstandardized additions to HTML, so they refused. Netscape at the end submitted JavaScript for standardization at the European Computer Manu-

[6] A functional language derived from LISP, Scheme has lexical scoping and a fairly simple syntax.

[7] An extreme dialect of Smalltalk, Self is a prototype-based dynamic object-oriented programming language[19].

facturers Association[22]. The new standard could not be named JavaScript as this trademark was owned by Sun, and the committee ended up naming the new language ECMAScript.

Today the name JavaScript, JScript and ECMAScript all refers to the same language, but as JavaScript is a trademark now owned by Oracle[8], ECMAScript would be its correct name. This project will use the name JavaScript throughout the report, as this is the name most commonly used "in the wild".

As Brendan Eich designed JavaScript he simultaneously made important decisions on designing the browser API that JavaScript was to interact with. This API was called the Document Object Model (DOM). The HTML elements made scriptable in this API mapped exactly onto the elements of the language HyperText[9]. The names from HyperText can still be seen today as in the attribute onClick. This was the zero level DOM.

Some years later, Scott Isaacs picked up the API and made some improvements. He added the iFrame and made all HTML elements scriptable. Later again, Netscape added the script tag, and Microsoft added the source attribute, allowing a general separation of concern for the first time.

The early work on the DOM was bad for web developers, and it is reasonable to say that it deserve some discredit for dragging JavaScript down as a language. During the browser wars [10] Netscape and Microsoft engaged in a spiraling attempt to out-innovate each other. They added unique features and quirks, which made it exceedingly difficult to develop pages that worked in both browsers. As a result images like the one in Figure 2.3) was common on websites.

[8] Through acquiring Sun in 2010[23, 24].

[9] HyperText was designed as a programming language to bring expressiveness back.

[10] The competition between Netscape Navigator and IE to become the dominating browser.

Figure 2.3: An example of earlier "best viewed with" image for IE and Netscape.

The W3C, the official standards body, tried to negotiate with the companies to achieve a standard, but were only partially successful. By looking at the HTML tree in the different browsers today, we can see how they differentiate by loading the same web page.

The HTML that was post-standardized after these events does not give any guidance on how to parse documents, and as a reaction browsers are mostly silently ignoring unknown tags. But in some cases, this has resulted in a fault correction mechanism; tag insertion. Browsers will as an example insert a missing head element, but the exact placement varies among browsers. All browsers, but IE, insert text nodes for blank spaces. In general, this makes it difficult for JavaScript to handle the tree traversal and other interaction correct across different browsers. As a patch, HTML5 contains a full parsing mechanism for web pages, and it is hoped that newer browsers will keep to this, though it will take time before old browsers are replaced.

The DOM also contains a pointer structure to parents, siblings and children. For retrieving nodes, there are two options available. Either traverse the tree structure or through methods as `getElementById`, `getElementsByName` and `getElementsByTagName`. The two latter are not recommended to use due to performance issues with the return of node lists. These nodes are possible

for JavaScript to modify and add to. In some of these modifications, it is needed to add to both IE specific attributes and W3C defined ones, making DOM modification a complicated process to attain the same results in all browsers. Also the DOM and the Cascading Style Sheet (CSS), although aware of each other, decided on the least compatible way of writing; CSS used hyphens to describe attributes, the DOM developers used camel case[11]. Today we still experience the pain caused by the browser wars and W3Cs early disagreement with the web community. Problems relating to event dispatching and the prevention of default event can also relate to this period and the intensely competitive environment that JavaScript grew up under.

Suddenly, with the demise of Netscape, it all disappeared. Microsoft decided that its .NET platform was the next thing, supported by researching companies all saying the Internet had played its part[25]. Microsoft had made the `XMLHttpRequest`, had made JavaScript for IE and made the DOM usable, if not pleasant.

The next event in JavaScript history was the dawn of Ajax[12] in 2005. The term was coined by Jesse James Garrett[26], user experience designer and co-founder of Adaptive Path. Together with a group of programmers he was planning a way of doing partial page replacements instead of a full replacement upon user interaction. This was to help site load speed and make interaction easier and more responsive.

With Ajax, the developers started innovating on with browser software, as opposed to browsers themselves, and using mainly on JavaScript as a tool. The reason of this shift was the after-effects of the browser wars. Though a

[11] Camel case means that words are joined with the first letter capitalized, e.g. iAmWritingCamelCase.

[12] There is some confusion on Ajax vs AJAX. Garret described *Ajax* as asynchronous JavaScript and XML, leading to the confusion an *AJAX* acronym. Today Ajax encompasses other technologies as well, especially JSON, so AJAX as an acronym does not refer to the whole concept. Due to this, this thesis will stick to *Ajax*.

great source of innovation, the web was now filled with users on old browsers, proving a barrier to the innovation of new software features. To make the most out of this web tangle, it was time for innovation not in the browser, but in libraries and browser add-ons. Web developers started to make Ajax libraries to mask the problems of the DOM and to heighten the interactivity, simply pushing the browser to an extent not thought possible. Ajax libraries provides portability of a web page across browsers, masking the differences and providing a simpler and more consistent API, leaving web developers able to focus their attention on a higher level.

Currently the diversity between the libraries is high. JavaScript gurus long predicted a shakeout between the libraries, but currently there are many different ones, all maintained by a highly dedicated community. The libraries all offer different programming models, widgets and inheritance model, leaving the choice between them to the most difficult problem.

JavaScript became popular even though it had bad parts and a difficult interaction API. Originally, Java was intended to be the language that brought interactivity back to the browsers, but this failed

JavaScript was needed to keep the web moving forward. HTML and CSS had long ago surpassed their original scope and web design was getting harder and harder to do correctly. George F. Colony, CEO of Forrester Research in 2000 said:

> "Another software technology will come along and kill off the Web, just as it killed News, Gopher, et al. And that judgment day will arrive very soon - in the next two to three years."[25]

Ajax helped release the potential of the web and of the JavaScript language and this makes web development significantly today. But historical effects still prevail, IE 6 still has a 12 % market share worldwide[13] and it was

[13] According to `http://www.ie6countdown.com/`.

released in 2001. The evolution has come a long way since then, but developing for IE6 still sets websites back by years. New standards like ES5 and ES5 Strict[14] go back and changes some on the inherent fault made in JavaScript, but only by discontinuing the support for old browsers can JavaScript and web standards move on.

2.3.2 The language

JavaScript is designed around some central features that make up the core of the language. These include functions, loose typing, dynamic objects, prototypal inheritance and an expressive object literal notation.

Functions The functions in JavaScript are first class objects and has, as discussed in Section 2.3.1, taken a lot from lambda languages like Scheme and LISP. They can be passed as any other variable and be an attribute of an object. The scope in JavaScript is lexical and functional, so the functions support something bigger; information hiding and encapsulations through the use of closures. This is explained further in the **Strengths** section.

```
1  var person = {
2    name: "Tine"
3  };
4  person.getName = function() {
5    return this.name;
6  };
7
8  function add(i, k){
9    return i + k;
10 };
11 person.canAdd = add;
```

Listing 2.1: A JavaScript function and method.

[14]The two sets of ECMAScript version 5.

```java
public class Person{

  String name;

  public Person(){
    this.name = "Tine";
  };

  public String getName(){
    return this.name;
  }

  public static int add(int i, int k){
    return i+k;
  };
}
```

Listing 2.2: A Java method and function.

Object literal The two examples in Listing 2.1 and 2.2 expose many differences and specialties of JavaScript. Firstly the *object literal* notation

```javascript
var person = {};
```

is exposed. This notation can add attributes as key-value pairs separated by a comma. The key can be anything but the reserved words, and the values have no restrictions. Secondly as functions are first class objects, it is possible to assign it as a property of another object as done in line 12 in Listing 2.1.

In Java, the matter changes. The class based structure makes the creation of new objects more cumbersome. Also the `add` function cannot be changed from a static function to an object method.

Loose typing *Loose typing* can be best explained through what it is not. A strongly typed language will impose restrictions on which types can interact

and the compiler will detect any "wrong" types interacting and throw an error. The compiler will at least stop the moment it reaches such a misuse. Fans of loosely typed languages will complain that the time spent on type conversion is not worth the time for bugs the compiler will fix. Returning to Listing 2.1 and Listing 2.2, we see the Java code specifies the return type of `getName()`, type of the `name` variable and input types in `add()`. The JavaScript example does neither.

Inheritance The inheritance module, or model of programming, is also different within JavaScript, which employs *prototypal inheritance*[15]. This can be a difficult concept to grasp for classically trained, object-oriented programmers. JavaScript has different patterns that can be applied to achieve inheritance, but the main characteristic is that an object can inherit from another object, as opposed to classes. The most common patterns are:

- Pseudoclassical: using a function that adds variables to `this`.
- Prototypal: adding variables to a function's `prototype` object, which will be inherited by all objects created by the function. Listing 2.3 provides an example.
- Functional: adding variables to an internal object and returning a object containing accessor functions. Cascading this pattern creates inheritance as exemplified in Listing 2.4.

The differences in the patterns are the syntax, degree of information hiding, flexibility and code reuse, as well as the use of functionality provided by `new` and `prototype`.

The examples below make the differences between Java and the different patterns in JavaScript visible. We have a car inheriting from a vehicle object and its `getWheels` method, while overwriting the `describe` method.

[15]Not to be confused with the prototypal inheritance pattern.

```javascript
function Car(spec) {
  this.typeOfGasoline = spec.typeOfGasoline;
  this.describe = function(){
    return "I'm a Car with " + this.wheels + " wheels running on " + spec.typeOfGasoline;};
  return this;
};
function Vehicle(spec) {
  this.wheels = spec.wheels;};
Vehicle.prototype.getWheels = function(){ return this.wheels; }
Vehicle.prototype.describe = function(){ return "I'm a Vehicle with " + this.wheels + " wheels"; }
Car.prototype = new Vehicle({wheels: 4});
```

Listing 2.3: An example of prototypal inheritance in JavaScript.

```javascript
var vehicle = function(spec){
  var that = {};
  that.getWheels = function(){
    return spec.wheels;
  };
  that.describe = function(){
    return "I'm a Vehicle with "+ spec.wheels +" wheels"; };
  return that;
};
var car = function(spec){
  spec.wheels = spec.wheels ||Âă4;
  var that = vehicle(spec);
  that.describe = function(){
    return "I'm a Car with " + spec.wheels + " wheels running on " + spec.typeOfGasoline; };
  return that;
};
```

Listing 2.4: An example of functional inheritance in JavaScript.

```
1  public class Vehicle{
2    int wheels;
3    public Vehicle(int wheels){
4      this.wheels = wheels; }
5
6    public int getWheels(){
7      return this.wheels; }
8
9    public String describe(){
10     return "I'm a Vehicle with "+ this.getWheels() + " wheels
          "; } }
11
12 public class Car extends Vehicle{
13   String typeOfGasoline;
14
15   public Car(int wheels, String typeOfGasoline) {
16     super(wheels);
17     this.typeOfGasoline = typeOfGasoline; };
18
19   public String describe(){
20     return super.describe() + ", more precisely; a Car with "
           + this.getWheels() + " running on " + this.
          typeOfGasoline;  }
21 }
```

Listing 2.5: Inheritance in Java.

There are many differences to be pointed out in the example listings above. The code in Listing 2.3 forces a car to have four wheels, while Listing 2.4 gives a car four wheels unless else is specified. The JavaScript examples works on an object level using functions and the Java example works exclusively on class level. It is important to note that Java has easy access to the inherited methods through `super`, this connection has to be created by the programmer in JavaScript. An example implementation by Douglas Crockford can be found in Appendix A.

The pseudoclassical pattern are demonstrated in Douglas Crockford's excellent book "JavaScript : The Good Parts"[27]. All the inheritance patterns have their strengths and weaknesses, but this alone is a big difference from other classical languages, exemplified by Java. In Java there are no "other" way of doing inheritance, the syntax and setup will look more or less the same across files and among web developers. In JavaScript the different inheritance patterns can create confusion and it is far easier to introduce bugs and unintended behavior.

Strengths

number type Starting of with a look at the types found in JavaScript it has a great strength in its simplicity. There is only one type of number, corresponding to Java's `double`, which help avoid a great set of issues related to overflow and other numeric errors. Though it introduces a weakness with regards to decimal fractions explained in the next section.

Variables The ease of the type systems is in general a great strength for JavaScript. Creating new variables are made easy and the loose typing system simplifies interaction between objects. New variables are created with the keyword `var` and objects and array are created through the use of literals:

```
1 var object = {name: "Tine", age: 23} // object literal {}
2 var array = ["one", 2, "three"] // array literal []
```

Closures The lexical scope in JavaScript provides the option for closures. This means that the inner function always has access to the variables of the outer function, even when the outer function has returned.

```
1 var person = function (name, age) {
2   var getName = function(){
3     return name;
```

```
4    };
5    var getAge = function(){
6      return age
7    };
8    var getOlder = function(){
9      age = age + 1;
10     return age;
11   }
12   return {  getOlder : getOlder,
13         getAge : getAge,
14         getName : getName};
15 };
16
17 var t = person("Tine", 21);
18 t.name          // undefined
19 t.age           // undefined
20 t.getName()     // "Tine"
21 t.getAge()      // 20
22 t.getOlder()    // 21
23 t.getAge()      // 21
```

Listing 2.6: Closures in JavaScript.

As seen in the listing above, a lot of features usually seen as object oriented can be achieved in JavaScript as well. The code in Listing 2.6 creates private variables for each person object that is created. The variables of person can only be interacted with through the functions in the returned object.

Closures are powerful for information hiding and encapsulation and a definitive strength of JavaScript, but at the same time, they can be difficult to understand and apply correctly.

Weaknesses

As Section 2.3.1 shows, the language was developed over a short period of time. This inevitably led to some bad design elements and in general, the features described in this section needs to be avoided as much as possible.

Global object One of the worst, or perhaps the worst part is the global object. This is the same as the `window` object in browsers. Any variable declared as a global variable is visible to all scripts running on a page. As the scripts increase in size the risk of a name collision increases, and they can end up modifying each other's variables, without any compiler or runtime warning. This is difficult to diagnose as the number of scripts increase. On today's websites JavaScript are utilized more and more, so it gets more and more likely for a name collision to happen. If the scripts are independent subprograms, it is a risk that this is not discovered during tests.

There are three ways of creating global variables, which are shown in Listing 2.7.

```
1 //By declaring a var outside a function
2 var a = {};
3 
4 //By using a variable without declaring it, it is implied
    global
5 a = {};
6 
7 //By adding it as a property to the global object in browsers
8 window.a = {};
```

Listing 2.7: The three ways of creating a global variable.

Scope The scope in JavaScript is also made to confuse. In most languages containing curly bracket blocks, `{}`, have block scope. This means the variables declared inside the scope are not visible outside. JavaScript has func-

tion scope; all variables declared inside a function are visible inside the function. This is why it is considered a best practice to declare all variables at the top of the function.[16]

NaN Another bad feature is the concept of `NaN` or *Not a Number*. This is the result of an operation not possible, like dividing by 0 or converting a non-numeric string to a number and can happen because of the loose typing apparent in JavaScript. A chain operation where this is the case with one of the operands, the end result will be `NaN`. The developer must then try to roll back to the operation that created the first `NaN` result. There are also some other interesting and non-intuitive aspects of `Nan`:

```
typeof NaN === "number" // true!
NaN === NaN //false
NaN !== Nan //true
```

The function `isNaN()` that checks whether the argument is `NaN`, but the safest way is to do a `typeof` check and `isFinite()`, as the latter also excludes the number value `Infinity`.

Arrays Arrays in JavaScript are also not as good as in other languages, mostly with regards to performance. The JavaScript array is an object simulating an array. The means that the `typeof` operator does not distinguish between an array and an object, though it is possible to test for `this.constructor === "Array"`. Since arrays are only modified objects they do not have the performance gain given in other languages, but it makes them easy to use. There is no default sort and there is no need to decide a length of the array, although the `length` property exists. The insertion in arrays are done through hashing, which in dense arrays, is more or less

[16] Ta med hoisting?

similar to a linear search. Because of this, arrays in JavaScript are best used when the keys are integers in a natural order.

Falsy values JavaScript has a set of so-called *falsy values*. These values evaluate to false when converted to booleans and they can be confusing in a while-loop or if statement. These values are shown in Table 2.2.

Value	**Type**
0	Number
`NaN`	Number
`""` : empty string	String
`null`	Object
`undefined`	undefined
`false`	Boolean

Table 2.2: Falsy values in JavaScript.

Decimal numbers JavaScript only have one number class, which in many ways are very convenient. The standard used are binary floating points, which introduce a major issue, as the decimal fractions in this standard are not handled. In the listing below, the problem is illustrated.

```
1 var i = 0.1;
2 var j = 0.2;
3 i+j                    // 0.30000000000000004
4 (i*100 + j*100)/100    // 0.3
```

Listing 2.8: Decimal floating point arithmetic in JavaScript.

As demonstrated, the decimals need to be multiplied before they can be accurately added and then reverted back to decimal numbers.

Equality The equality operator as we know it from other languages tests for equality in objects when used twice (==). JavaScript has two versions of equality tests (==) and (===). The latter works as expected, returning only true when the objects are of same type and have the same value. The former tries coerce if the objects are not of the same type, and the result renders the operator highly unstable, as there are no transitivity and many special cases. It is advised to always use the triple equality.

```
'' == '0'          //false
0 == ''            //true
0 == '0'           //true

false == 'false'   //false
false == '0'       //true
```

Listing 2.9: Equality in JavaScript with the double equals operator.

new In the pseudoclassical inheritance pattern the `new` operator is used on custom constructor functions. A call with `new` creates a new object that inherits from the objects prototype and then binds the operand to `this`. This gives the construction function the ability to modify the object before it is returned. If a constructor function is called without `new` the function uses the global object as `this`, using and updating global variable for each call. There are no warnings on this practice. The best practice of capitalizing constructor functions gives a visual clue that the function is to be used with `new`. An alternative implementation is to use a `create` function as demonstrated in Appendix A.

There are also other parts of the language that are better not used, either because of obscurity or the risk of doing it wrong, as well as known syntactic bugs. For a full reference of the recommended subset of JavaScript, Douglas Crockford's book "JavaScript : The Good Parts" [27] is recommended.

Here is also the syntactic code checker JsLint introduced. JsLint is a tool that scans the input code for syntactic errors and discrepancies from code conventions, providing developers with another set of "eyes" to check their code.

2.4 Summary

This chapter has covered a lot of ground on the separate topics unit testing, TDD and JavaScript. The concept of unit test and corresponding properties has been covered and the *xUnit frameworks* described. The methodology of TDD has been introduced, the red/green/refactor - mantra and how to use patters to achieve the right progress. Also current research on TDD has been covered and it was found that the method holds inconclusive results with regards to measurable effects, although experts vouch for it.

BDD has been introduced as an offspring of TDD and it was shown how it is utilizing natural language to ease communication with stakeholders. Integration testing has also been covered to attain insight on the possible higher levels of testing.

The main features of JavaScript have been pointed out, and how these differ from classical object-oriented languages, exemplified by Java. The strengths and weaknesses of the language have been discussed as well as its history.

Chapter 3

Research method

This chapter will look at and explain the research method applied during the work with this thesis. Further it will discuss the limitations with regards to the chosen method, and present the justification behind the choice.

3.1 Literature study

As an introduction to the topic, a literature study was conducted to evaluate the research done on this area. A search was done on the following keywords and combinations of them:

- *Frameworks evaluation.*
- *JavaScript.*
- *Unit testing*
- *Test-Driven Development.*

The search results were drilled down into to make it more relevant and discover potential studies.

The search was conducted on the following platforms:

- IEEE Xplore[28]
- SpringerLink[29]
- ACM Digital Library[30]

It was found that little study had been done on any of the topics combined, which lead to a revision of research method, as described in the next sections.

3.2 Research methods

This section explains the different research methods that were considered and why they were ultimately discarded. This provides the background for the choice of method, which is explained in the next section.

3.2.1 Qualitative methods

A qualitative approach to the research questions would be to gain a deeper understanding of performing and reasoning around tools for unit testing with TDD in JavaScript, as well as more information around context and evaluation. The research questions posed in Section 1.2 could be answered though interviews, group discussions and well as observations and field studies with experts and/or teams working with the topics. If previous studies had been published, analyzing a group of similar cases could give insight to a larger context, possibly testing a new hypothesis for Research Question 3.

The issue with this approach is the lack of previous studies on the topics and the relatively small known adoption of JavaScript TDD techniques in companies. Because the scope of this thesis could not include a mapping of companies using TDD tools on individual projects, the qualitative approach was discarded.

3.2.2 Quantitative methods

Quantitative methods are used to collect information in a structured way to answer a set of hypotheses. Using statistics or collected empirical data through observation, the research questions in Section 1.2 could give answers less connected to individual or a group of cases. E.g. a survey connected to the research questions conducted on groups developing with and without a TDD approach to unit testing JavaScript.

Problems with a potential survey would be that the sample of companies using TDD and JavaScript would be too small to give statistical verification to the results, and not many companies were known to develop this way. With regards to the duration of this thesis, only a small amount of time could be spent on finding companies and if this number came out to small, the thesis could not be concluded and such another method had to be chosen. This risk was not acceptable, and a survey approach could not be used.

Observations on different teams is similar to the research conducted and described in Section 2.2.3; the actual comparison is difficult to internal and external differences among teams, creating doubts on the exact source of discrepancies in measurements. This form of observation would be out of scope of this thesis, due to experience, duration and resources.

3.2.3 Feasibility prototyping

Feasibility prototyping is a form of software prototyping found within the method of systems analysis and is described as[31]:

> [...]feasibility prototyping is used to test the feasibility of a specific technology that might be applied to the business problem[...]

This would mean to create actual prototypes that can be tested in on an existing business problem to determine its fit within the domain.

The general goal of systems analysis is to dissect a system and study the interaction between the entities[31, 32] and the feasibility prototype will be one of these entities. The prototype sets out to prove a set of technical assertions and will verify that the architecture and total system solution will fulfill the business needs. It will act as a *proof of concept* for the details implemented in the prototype.

The problem with regards to feasibility testing in this thesis is the lack of a business problem to integrate the prototypes with. Is would be possible to formulate a business problem to reflect the research questions, e.g.

> The business want to use TDD and similar techniques to test its JavaScript application development.

Feasibility prototyping would then be applied to this problem with the different framework to find the best fit.

The issue with this approach is the "invention" of a business problem. This would be an insistence on a fit between the research questions and thesis methodology. The method needs a real business problem, and this leads to discarding feasibility prototyping.

3.3 Adopting demo and proof of concept - methods

The three methods described above have their origins in different parts of computer science. The two first methods belong with the traditional domain of computer science, while the last are place more within the domain of software engineering. The traditional sciences have formulation of hypotheses and testing these as central[34], while software engineering are more concerned with problems around design, construction and maintenance of industrial software[34]. Referring to the discussion above, the lack of resources,

time and experience hinder a traditional approach, while the lack of industrial application and a business problem hinders the feasibility prototyping.

The reason for this can possibly be the current software readiness level (SRL). SRL is defined by the US Department of Defense to assess hardware and software readiness, where level 1 is least ready and 9 is fully operational at no risk. Initially, JavaScript unit testing frameworks seem to be on level 6, defined as:

> Level at which the program feasibility of a software technology is demonstrated. This level extends to operational environment prototype implementations where critical technical risk functionality is available for demonstration and a test in which the software technology is well integrated with operational hardware/software systems.

The next level emphasizes full existing documentation, which is lacking for many frameworks, and the lower level only indicates prototype implementations.

The low SRL can be a reason for the low recognized adoption of JavaScript unit testing frameworks, and also the reason for the limited current research on the area.

With this backdrop it would be most appropriate for this study to provide an incentive to further research by providing a demo and a proof of concept for JavaScript unit testing with TDD. The study can this way give a starting point for new studies and discover new aspects that need clarification, while reducing the chance for choosing fruitless directions in the future. It is also expected that the study can illustrate a potential in the frameworks and TDD methodology, providing an incentive to framework developers and further business adoption.

3.3.1 Process description

Using the above arguments, the study's method will be based on exploration of the current state of the art and issues found when interleaving the chosen topics, TDD, unit testing and JavaScript.

The evaluation of the frameworks could be done with proof-of-concept test cases, designed to address a specific concern found within JavaScript testing. The test cases, when carefully chosen, will address these concepts and provide a better background for evaluating the frameworks.

The process in the thesis is described in the Figure 3.1.

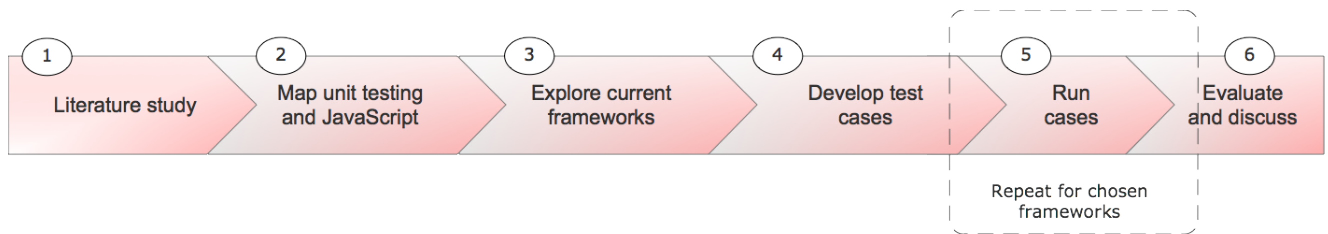

Figure 3.1: Project method: step by step.

The steps 2 to 6 can be done iterative to account for learning throughout the project. This can be a buffer since the concepts discussed have no found previous research. The iterations will allow revision of subparts according to experience gained when working in the domain.

Literature study

This part provides the background on the separate topics. Using the sources from the initial literature study in Section 3.1, TDD can be researched. When learning a new language, it was considered convenient to use recommended book, and here JavaScript guru Douglas Crockford's book "JavaScript : The Good Parts"[27] have been used together with "Object-Oriented JavaScript" by Stoyan Stefanov[35].

Unit testing has also been well described in books, here Gerard Meszaros' "xUnit Test Patterns : Refactoring Test Code"[5] and Roy Osherove "The Art of Unit Testing"[4] have been used.

This part maps to Chapter 2 of this thesis.

Map unit testing and JavaScript

This section would need to go from the general testing of JavaScript to concrete issues when doing unit testing. To answer some questions and provide insight, the Oslo XP meetup *"Test-Driven JavaScript with Christian Johansen and Fabian Jakobs"* was attended on the 24th of January. Also a copy of Johansen's book "Test-Driven JavaScript Development"[36] was read to learn from the challenges he met.

This section would provide some digging into the features of existing framework, so the work in this section would be somehow parallel with the research for Step 3 in Figure 3.1.

The result of this work maps to Chapter 4.

Explore current frameworks

Through using search engines, books and databases, a search for current framework could be conducted. Explaining the found frameworks according to the general unit testing features and JavaScript specifics would be important. Other characteristic could also be given weight, e.g. support and last time updated.

The result is found in Chapter 5 and this answers Research Question 1.

Develop test cases

Step 4 in Figure 3.1 takes aim to develop pseudocode for set of test cases that will cover the relevant features needed for general unit testing and specific

JavaScript issues. These will be uncovered by Step 2 in the research method. The test cases take aim to describe a program to be developed with a TDD process, and the descriptions will be minimal to ensure no assumptions are given.

The test cases need only to encompass a small set of features, just enough to cover the aspect necessary to evaluate.

This step maps to Section 6.1.

Run test cases

Before the tests are run, an assessment of the found frameworks needs to be done. The thesis may only need a subset of the frameworks to be covered in order to answer the remaining research questions.

When this is decided, the test cases will be developed with a TDD process with unit test. No minimum test coverage is set.

The development will happen on a personal laptop with a number of current browsers installed, a MacBook Air running Mac OS X version 10.6.7 with browsers Firefox, Chrome, Safari and Opera installed.

The result of this part is covered in Section 6.2

Evaluation and discussion

The last section will evaluate the development of the test cases in the different frameworks and discuss the findings. This section will answer the remaining research questions based on the results from the execution.

This is done in Chapter 7.

3.4 Limitations

In any research project there will be limitations due to different factors. In this thesis, the factors are

- Low personal experience will affect the difficulty of development, which can again affect results and evaluation.
- Research for JavaScript testing on enterprise level are out-of-scope.
- No known previous research or comparisons are done.
- The time for execution of the project are limited and such the programmatic output is limited to a proof of concept.
- The frameworks are evaluated outside a development project, making the conclusion non-applicable for all situations.
- Recreating the study may be difficult as it is dependent on programmatic experience.
- Hardware limitations does not allow the test cases to be executed in IE[1].

3.5 Summary

This section has explained the reasoning behind the choice of research method. As no previous research has been done, the process will result in a demo and proof of concept. This cannot be regarded as scientific evidence, but will provide a background and incentive, and be illustrative for traditional computer

[1] IE can run on Mac OS X through various extensions, but it demands other types of software not available for this thesis.

science and software engineering methods that seek to elaborate on specific factors, hypothesis or industrial applications.

The process itself is described in Figure 3.1, and consists of six steps. These range from background material to development and execution of test cases to cover general and specific issues related to unit testing in JavaScript.

Chapter 4

An introduction to JavaScript and testing

In loosely typed languages such as JavaScript, testing is what keeps the application together and give developers confidence to make changes to existing code. This chapter will introduce the reader to the concepts around JavaScript and testing. Firstly, an overview will be given of the JavaScript features that affects how and what kind of testing can be done. Secondly, a vocabulary is introduced to more clearly distinguish the differences and to assure consistency during the course of discussion. Lastly, a closer look will be taken at important differentiators in the frameworks.

The concepts introduced in this section are based on prestudies to frameworks described in Chapter 5 and the background gained in Chapter 2. The section seeks to establish a common vocabulary for discussion and further evaluation of JavaScript testing frameworks and unit testing in particular.

4.1 Characteristics affecting tests

In the following section a set of characteristics of the web language JavaScript will be presented. Their influence and level of testing are discussed, according to the research method presented in Chapter 3.

Interaction

> JavaScript is used to interact with the User Interface (UI) through HTML and CSS.

JavaScript acts through the DOM to interact with HTML and CSS to change the behavior and look of a site. The different implementations of the DOM have resulted in another characteristic of JavaScript, namely **cross-browser execution**, which is discussed later.

To be able to test JavaScript and HTML interaction, the relevant piece of HTML code must be available to the test. One solution would be to run the tests on a live site in a production environment. To be able to execute fast and independent of external resources as encouraged by TDD and unit testing[1], this is not a good solution. Instead of the test running in the HTML domain, the HTML can be inserted in the test domain. This means that the HTML code needed is included in the test cycle. This is an example of an external fixture. It would be important that the framework could handle correct setup and teardown to ensure test isolation.

Pure look or UI is beyond unit testing frameworks[2] and will not be considered when exercising the frameworks and reviewing the results. But as a professional domain, this is one of the areas where TDD development has not yet reached, mostly because of a lack of tools. Testing of user interfaces is

[1] This is discussed in Section 2.1.1 and 2.2.2.

[2] Referring to Figure 2.1 on page 7.

still in a large extent manual. Kent Beck has argued that TDD can be used. Variants of integration and acceptance testing[3] can be used to test reaction to user input on different granularities referring to Figure 2.2.

Responsiveness

A website that does not respond or respond slowly will lose users

This characteristic is about the users' experience of a website. A user perceiving a site as unresponsive will quickly dismiss the site out of frustration or distraction. It was found that an increase in server delay from 1 to 2 seconds, made users increase their time to click[4] from 1.9 to 3.1 seconds[37], which indicates a distraction. The same experiment found no changes in revenue/user for 0.2 seconds or lower, but from 0.5 to 2 seconds delay, revenue dropped from in a range from -1.2% to -4.3%.

To attain a responsive site, a lot of technologies need to cooperate and JavaScript is only one part. Load times, slow internet connection, server delays, non-optimal loading or round trips are examples of such issues. Simple tricks as minifying[5] and zipping files can make a difference. In general this makes benchmarking code on JavaScript engines less important, as it does not test in a natural environment nor tests effect on perceived experience.

Performance testing is a an area that is not very well covered, and on of the initial areas where Kent Beck states that TDD still has issues[3]. In his book[38], Christian Johansen describes how `new Date` can be used to time the execution of methods, but this can currently only be done with an HTML test runner. It is also dependent on a given time limit or multiple implementations to check who is the fastest.

[3]Integration testing is covered together with acceptance testing in Section 2.1.2.

[4]A measure of how long it takes until the user clicks on the site again.

[5]Minification refers to the practice of wiping a document of unnecessary characters, such as whitespace and comments, to reduce file size.

In general, JavaScript performance can be improved in many ways. First, speed is dependent on the right algorithms and data structures. Second, costly DOM operations can be minimized, e.g. always use lookup by ID instead of traversing the nodelist with class or name lookups. Lastly, some browsers have slow spots, e.g. IE should always have defined variables, its slow lookup back to the global scope is generally known to take time. The `eval` function is slow across browsers and should be avoided.

It is important to remember that execution speed (in LOC) is a trade-off with file size (as loading time), because all JavaScript scripts need to be downloaded. This usually leads to code optimization on hotspots together with minification of script files.

In total, responsiveness is primarily not an important aspect when doing unit testing, because of the limited testing and test isolation. This characteristic will be disregarded in the execution and results of this thesis.

Cross-browser execution

> The JavaScript code needs to work across browsers and operating systems.

Users expect the site to look and behave the same across browsers and OS's, but this is not achieved automatically due to the various versions of JavaScript and differences in DOM implementation in browsers[6]. This introduces another challenge for a testing framework, as the tests should optimally run in all browsers on all platforms in a quick and effortless way to fulfill the unit testing characteristics.

[6] Both are introduced in the History Section 2.3.1.

Complex logic

> JavaScript can be used for development of enterprise applications, indicating bigger and more complex logic.

JavaScript has gained popularity in application development, both on client and server and up to an enterprise wide level. Innovations such as node.js[7] also makes server-side[8] testing highly relevant.

The sheer number of files involved as the source code grows, puts more pressure on readability, separation of concerns, configuration and deployment of the testing tool. Also, the potential for integration with a continuous integration (CI) system can be important, depending on the way the code is managed.

When the number of tests increases, the speed of the individual test run is also more noticeable. This means that tools must have efficient testing mechanisms, and be able to abstract away external dependencies to avoid relying on systems outside its control.

Server-side code and parts of application code will have in common that they do not rely on the browser's API, which opens to new execution models outside the browser. This gives the testing tools new opportunities to achieve the desired speed, configuration and deployment.

Asynchronous code

> JavaScript uses asynchronous code in Ajax calls and timing functions.

Asynchronous code leaves a function and its current scope on the stack for later execution, even after the original invoking function has returned.

[7] A server-side JavaScript environment that uses a asynchronous event loop[39].
[8] Testing outside the browser and browser API.

This implies that testing an asynchronous piece of code must wait until this function on the stack has executed before validating the result, as the return value of the function under test will not be relevant. The function on the stack will run as a response to an event, a state change in the case of Ajax or a timer running out. This means that a testing tool must either be able to pause the test and wait a certain amount of time before validating, fake this interval or use xUnit patterns to abstract away the dependency.

When testing with the JavaScript timing mechanisms[9], the developer will know the correct interval to wait, as this needs to be specified when invoking the async function. The testing tool should be able to fake the duration of the actual waiting time, or change the design to abstract the wait away. If not, the developer risk waiting before the test results return, or not being able to test the timing mechanism at all. This will either result in lower test coverage or breach on unit test principles again causing an disturbance in the red/green/refactor cycle[10].

The reliance on Ajax is also an external dependency that creates an interval before execution that is not decided by the programmer. The external dependency refers to the site that receives the request. This site is beyond the control of the developer and the test is suddenly dependent on the current connection between the server site and the test machine, as well as any internal factors on the server. These are irrelevant to the code under test, and they are favorable to fake or abstract away to return the control of both time and the actual response to the programmer. If the framework does not provide faking, the testing must be done in a different manner, by changing design, reducing the test coverage or to introduce the dependency and increase wait time against the red/green/refactor cycle.

[9]Like `setTimeout` and `clearTimeout`.
[10]Explained in Section 2.2.1.

If a wait is introduced, the test risk failure even with correct response, if that response is returned after the wait interval set by the programmer.

The asynchronous characteristics and solutions are applicable on all system level granularities, but it will possibly demand more setup on coarser levels, depending on exposed API or UI and execution environment.

4.2 Vocabulary

The following terms will be used in the coming chapters to more accurately describe the available technologies for unit testing in JavaScript.

The terms are derived from Meszaros[5]:

- A *test library* is a collection of methods to help preform unit testing patterns. A test library has no way of running the methods on its own, but it will decide the lifecycle[11] of the tests.

- A *test runner* will execute the test methods programmed with a *test library* and return the results. Multiple interface variations exist, the most common are graphical and command line.

- A *test framework* includes a test library for writing tests and a test runner for execution.

The terms are taken from the xUnit framework family and can be extended to JavaScript tools. There seems to be some confusion on the naming of different parts in a framework, so these clarifications are meant to provide a common ground for discussion.

[11] The sequence of setup, individual tests and teardown methods.

4.3 Examining a JavaScript framework

JavaScript IDEs do not provide the same help as IDEs for static typed languages like Java and C#[12], which leads to a lack of a *de facto* JavaScript IDE. This again prevents the community to standardize on a single testing framework. Furthermore, debugging has in the past been manual through alert statements and browser extension like Firebug[40]. For TDD it is important that the language and environment support a short round trip time in the red/green/refactor cycle[13] to help realizing the design and test patterns.

This section will describe the most important differentiators in JavaScript unit testing frameworks:

- Environment
- Execution
- Syntax
- Test library

The four sections were chosen based on what was found during research for Chapter 5, as these were the main factors where the researched frameworks differed. These will act as reference points when evaluating the frameworks and they are furthermore discussed and revised in Section 7.5.

Environment

The most natural environment for a JavaScript application[14] is the browser, and in many situation the JavaScript will depend on the browser's DOM

[12]This is especially concerned refactoring operations.

[13]As described in Section 2.2.2.

[14]JavaScript application is a common denominator for JavaScript files, script or applications running in the browser.

in order to function correctly. As explained in Section 2.3.1, the DOM has many shortcomings that need to be circumvented, but in situations where the JavaScript does not communicate with the DOM, the code can in fact be executed in an environment simulating the browser's JavaScript engine.

Execution outside the browser turns more useful as JavaScript have gained popularity as a server-side language and applications grow larger and more complex[15]. At this point the internal logic of the application can be verified in a simulated environment as it is the JavaScript on the client-side or in a dedicated content management systems (CMS) that will handle the DOM communication.

Currently, Mozilla supports Rhino[41], a JavaScript engine implemented in Java. Rhino provides the general execution model of JavaScript without the browser DOM API, but this is possible to simulate with John Resig's Env.js[42].

Execution

The main difference when it comes to execution of tests, would be whether the programmer needs to consult the browser or not to get results. Refreshing the browser for each test run is easy to implement and configure, but is not as fast and effortless as TDD process dictates.

The problem will increase according to the number of browsers that need to run the tests as they all will have to be opened and refreshed manually. The advantage of this method is the transparency offered – the library itself will be a .js file, making it easy to explore and extend using already known techniques. Many test runners will utilize this type of *in-browser* testing, creating a HTML page where the test and source files are loaded through `script` tags. For test libraries it is possible for programmers to create an own

[15]This is a characteristic mentioned in Section 4.1.

Figure 4.1: Execution of tests in the browser, exemplified by QUnit.

in-browser test runner by utilizing the library's method. This is showcased in the book "Test-Driven JavaScript Development"[43].

The alternative to *in-browser* is called *headless* testing. This implies that the tests are run from the command line or from inside an IDE, and the results are returned to this interface. They can run the tests in different environments, some run tests in Rhino, while others push the tests to a browser[16] and showcase only the returned results.

Headless testing can have more demanding setup with regards to configuration of local servers and potential connections to browsers or remote machines. The source and test files needs configuration to ensure correct referencing and, in some cases, monitoring operations searching for file updates. The process can give returns when tests are run in multiple browsers with less effort.

Section 2.3.1 introduced the Ajax libraries as another layer on top of browsers. These libraries also need to be taken into account when testing.

[16] This might need a browser opened by the user, but the browser will still not be consulted in order to see the results.

This means that the tests need to be run $i * j * k$ times to account for browser version, OS version and Ajax library version.

Syntax

The syntax of the framework is created to support a chosen methodology, which in this thesis will be TDD or BDD. A TDD syntax is of pure JavaScript with library methods similar to the *xUnit frameworks*. This is showcased in the following code example from JsTestDriver:

```
1 TestCase("FizzBuzzKata", {
2   "test on 0 return 0": function(){
3     var result = fizzbuzz(0);
4     assertEquals("Zero",0, result);
5   }
6 };
```

Listing 4.1: JsTestDriver TDD syntax.

The `assert` statement is typical for this syntax.

BDD syntaxes are more diverse, but they have in common that they stay closer to a natural language, as this is one of the principles of BDD[17]. Below Listing 4.2 and Listing 4.3 give two examples of different BDD syntaxes.

```
1 describe("FizzBuzz", function() {
2   it("return 0 on 0", function() {
3     expect(fizzbuzz(0)).toEqual(0);
4   });
5 });
```

Listing 4.2: Jasmine BDD syntax.

[17]Referred to in Section 2.2.4.

```
1 describe 'FizzBuzz'
2   it 'should return null on null'
3     fizzbuzz(0).should.equal 0
4   end
5 end
```

Listing 4.3: JSpec BDD DSL syntax.

Listing 4.2 has a JavaScript syntax, but the library methods are named close to the BDD principles. The `assertEqual()` statement in Listing 4.1 is now `expect().toEqual()`, which is more readable.

In Listing 4.3, a custom domain-specific language (DSL) is shown, in this case belonging to JSpec[18]. Here another step is taken into the domain of natural language.

The syntax shown in Listing 4.3 has borrowed heavily from RSpec[44]. RSpec was developed by Dan North, the creator of BDD[13] and RSpec is considered the template for many BDD frameworks today, especially with regards to syntax and lifecycle.

Test library

The test library decides how testing is done, what can be done and how demanding it will be of the programmer.

The library consists of methods for determining the lifecycle of setup, individual tests and teardown as well as grouping of tests into suites. It also has comparison methods, called assert mechanisms in TDD and matchers in BDD, which determine the outcome of the tests. The range and coverage of assert methods and the lifecycle of setup and teardown between libraries can vary, but in general these are the backbones of a library, and how they are built will depend on design goals and supporting methodology.

[18] JSpec is discussed in Section 5.2 and Section 6.2

When these essentials are covered, a test library can add features to ensure easier testing, higher readability and less repetition for the developer. Below, some general features are described. These can be found in many xUnit frameworks across languages. This is followed by features specifically developed to JavaScript characteristics.

General features

When writing a test, there are two paths to verification of correctness: *state* and *behavior verification*. *State verification* ensures correctness by checking that the state of the object under test has changed into some expected state. This is achieved through the use of local variables and assert statements.

Behavior verification ensures correctness by recording if an expected method is invoked during execution. The result of the method is irrelevant, the mere fact that it has been invoked as a result of the test run, verifies correct behavior. More advanced verification can be a method that is called exactly once, or verification of its parameters and scope.

These two methods benefits from tools in order to help the programmer to test the system in the best way. Tools like stubs, mocks and spies can save a programmer from repeatedly typing and abstract away implementations of interfaces.

Stub A stub can be used for behavior and state verification, as well as a range of other testing patterns. A stub will need to be set up and restored after each tests, and this can lead to tedious repetitive coding. A dedicated stub function can relieve this and potentially be a part of automatic setup and teardown.

A stub can also be used as stand-in to force a specific code path, as a saboteur to force exceptions, to avoid unimplemented methods or to simplify an implementation of an inconvenient interface.

In the example in Listing 4.4, the stub acts as a stand-in for the `functionUnderTest` method, allowing only the invocation itself to be verified.

```
function stubFn() {
  var fn = function () {
    fn.called = true;
  };
  fn.called = false;
  return fn;
}

// Test

var functionUnderTest = stubFn();
//calls that shall invoke method
assertTrue(functionUnderTest.called);
//steps to restore functionUnderTest to its original form
```

Listing 4.4: Stubbing a function for behavior verification, courtesy of Christian Johansen[45].

Spy A spy is a way of performing behavior verification. A spy is a stub implementation[36] like the one seen in Listing 4.4, only that it also records the arguments and context of the call.

Mock A mock can *only* be used for behavior verification. As a stub, the mock has some pre-programmed behavior, but is also subject to pre-programmed expectations. This means that a mock changes the way of programming, demanding that the expectations are stated at the beginning of a test, and at the end of the test only verifies that the expectations were met.

Stubs, mocks and spies can be used interchangeably in many situations, so choosing between them can be based on personal preference or readability concerns.

Specific features

Testing JavaScript contains a lot of the general patterns found within TDD as discussed in Section 2.2.2. Features that need to be tailored to JavaScript is centered around asynchronous code[19], like Ajax and timing functions.

Asynchronous code To achieve the degree of isolation necessary to create good tests, the library need to fake an implementation of the XmlHttpRequest (XHR) object, that can act as a stand-in that does not fire an actual Ajax call. The most important parts of this task is to be able to trigger the browser's `onreadystatechange` method[20] and to fake responses. Abilities here are among others, possibilities to set headers in the response object, choose response text and which calls triggers which responses. The ability to trigger wrong responses to see how the code handles this is important to achieve robust code.

Simulation of time is encountered through the timeout mechanisms, like `setTimeout` and `clearTimeout`. For a library, it is important to be able to test events that are dependent on time, since this is a characteristic of JavaScript usage. The library can test this in two ways; either by pausing the test for the duration of the timeout, as specified by the programmer, or by faking the clock, and programmatically changing the time and forcing the timeout call to happen at once.

From a TDD perspective, the latter is definitely preferred, as it as it facilitates the vital speed requirements. A test with a timeout of one second,

[19] As described in Section 4.1
[20] This is triggered as the readyState of an XHR object is changed.

totals to an irritating wait when the number of tests are high. The chances are that the developer will start coding again before the result of the last test is run, reducing the usefulness of the TDD method. For libraries that have neither option, care must be taken to abstract away the notion of time or manually execute the inner function of the timeout. The former introduces a design change, and the latter does not give complete coverage, but is in many ways more compliant with TDD testing than the long wait for test results presented above.

Fixture Referring to Section 2.2.2, fixtures were discussed as a part of a TDD framework and it is also an xUnit pattern[3]. In JavaScript, external fixtures are commonly HTML code that is manipulated. With an *in-browser* test runner the developer can add an HTML element to the page, but care must be taken during setup and teardown to ensure the node is left in its original state as to not introduce cross-browser issues or dependencies across tests.

Fixtures could also be considered a general feature, but since it is confined to HTML fixtures, this thesis will regard it as a feature specific to JavaScript.

Event testing Some test libraries contain features for DOM event testing[21]. This is a property on a different level from the previously discussed, when referring to Figure 2.1, page 7. It usually happens on a coarser SUT granularity than unit testing and interacts through the UI, rather than the API. Event testing seeks to simulate a user's action on a web page or application, through events like `click`, `focus`, `hover` and `submit`. It is important with regards to JavaScript testing, but not applicable on a unit level. This thesis will not focus on event testing, but will describe it for the means of completion.

[21]Further references to event testing should be interpreted as DOM event testing.

4.4 Summary

This chapter has brought together JavaScript and unit testing with TDD and elaborated on the language features and characteristics affect testing. Interaction, responsiveness, performance, complex logic and asynchronous code have been explained and their specific addition to the area of testing has been unraveled.

A vocabulary has been introduced to clarify the terms used in the chapters to come. These terms are *test library*, *test runner* and the *test framework*, which describes the methods available, the form of running and the combination of those two.

The differentiators in a JavaScript framework have been examined: environment, execution, syntax and test library. The test library was divided into general features of any TDD framework, like stub, mocks and spies, and features specific to the language, covering the characteristics described in the first section.

Also during the course of the chapter, the terms *in-browser* and *headless* testing were introduced, as well as the two ways of ensuring correctness: *state* and *behavior* verification.

This chapter will equip the reader for evaluation of the framework in Chapter 5, results in Chapter 6 and discussion in Chapter 7.

Chapter 5

State of the art

This chapter takes a look at existing frameworks and libraries. The different aspects of the technology will be explained and explored according to the vocabulary, described in Section 4.2 and the differentiators of the framework, described in Section 4.3. As a summary, the main characteristics and features are presented in tables in Section 5.12.

Sections 5.6.1 and 5.9 described frameworks and tools that are not within the xUnit family. They are described to show the extended reach of JavaScript testing and the divergence currently found on the area.

5.1 Jasmine

Jasmine is a BDD testing framework developed by Pivotal Labs[46], the former maintainers of JsUnit[1]. The framework itself does not rely on browsers, the DOM or other JavaScript libraries, so it can be run standalone and *in-browser*, through a JsTestDriver plugin or with node.js. It also support continuous builds through maven, and has a Ruby gem and version for Rails.

[1] JsUnit is described in Section 5.4

It has a BDD syntax and follows the RSpec naming standards, with `describe` as test cases and `it` as individual tests.

Jasmine's test library has nested before and after functions, instead of more TDD-like setup and teardown. It has a wide range of built-in matchers and the ability to define custom ones. It handles asynchronous code through pausing tests and provides spies and stubs under the denominator *spies*.

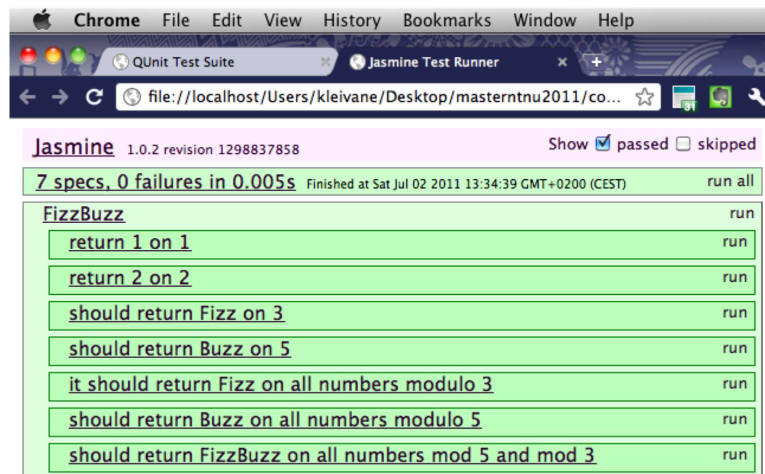

Figure 5.1: A result report created by the browser test runner.

5.2 JSpec

JSpec is a framework built to support BDD techniques. It is developed by TJ Holowaychuk and is one of the most feature-rich testing frameworks. It can be run in the browser or in a Rhino/env.js environment. It has a command line Ruby gem to automate testing on save and to initialize projects. The library features include support for faking Ajax call and time, spies, stubs

and a well of matchers. JSpec also has options for adding external fixtures, in JSON[2] format or HTML.

JSpec is heavily influenced by RSpec[44] and adopts a similar DSL. This gives a more human readable syntax as shown in Listing 5.1 helped by an abundance of available matchers. There also exist a grammar-less option for writing a JavaScript BDD syntax.

```
1 describe '.fizzbuzz()'
2   it 'should return 0 on 0'
3     fizzbuzz(0).should.equal 0
4   end
5 end
```

Listing 5.1: An example of the JSpec BDD DSL.

5.3 JsTestDriver

JsTestDriver is a test framework developed at Google and released in May 2009[47]. Developers Miško Hevery and Jeremie Lenfant-Engelmann recognized that the JavaScript community had not converged on a standard testing framework, as had happened for other languages, and set out to make the complete JavaScript testing framework. Specifically incorporated was command line control, parallel execution in browsers and instant feedback in IDE[47].

JsTestDriver's test library is extensive with regards to assertions and it also has an asynchronous API[48], but no tools for stubs, mocks, spies or faking time or Ajax calls. It has an own syntax for adding HTML fixtures `/*:DOC += <div id='foo'><div>*/`[49].

[2]JavaScriptObjectNotation.

The test runner of JsTestDriver is the main contribution. It runs command line or through an IDE plugin[3], allowing for execution in multiple browsers both local and remote, leveraging the scaling issues discussed in Section 4.1. A server is started through the command line and the browsers that are to run the tests are opened on an Uniform Resource Locator (URL) pointing to the server. JsTestDriver then pushes pending tests to the browser and returns the results to the command line. Configuration is achieved through a .config file, exemplified in Listing 5.2. It is also possible to include plugins, currently a module exists for measuring code coverage.

```
server: http://localhost:4224

  load:
   - src/*.js
   - src-test/*.js

  exclude:
   - somefile.js
```

Listing 5.2: An example configuration of `JsTestDriver.config`

The test runner is able to act as a runner for many other test libraries and frameworks. There exist adapters for YUI Test, QUnit and Jasmine and pure test libraries like Sinon.JS can run on top of the JsTestDriver library. The command line launching also makes it easy to integrate with a CI system.

5.4 JsUnit

JsUnit was the first testing framework for JavaScript and was started in 2001[50]. It is a direct port of JUnit[50], which makes it a part of the *xUnit*

[3] JsTestDriver can be integrated with IntelliJ and Eclipse. It also has a Maven plugin.

framework family. Today its main developer, Pivotal Labs, is no longer actively maintaining the project, as they are focusing on Jasmine[4][51, 50].

JsUnit comes with an in-browser test runner where the test file needs to be specified, as seen in Figure 5.2.

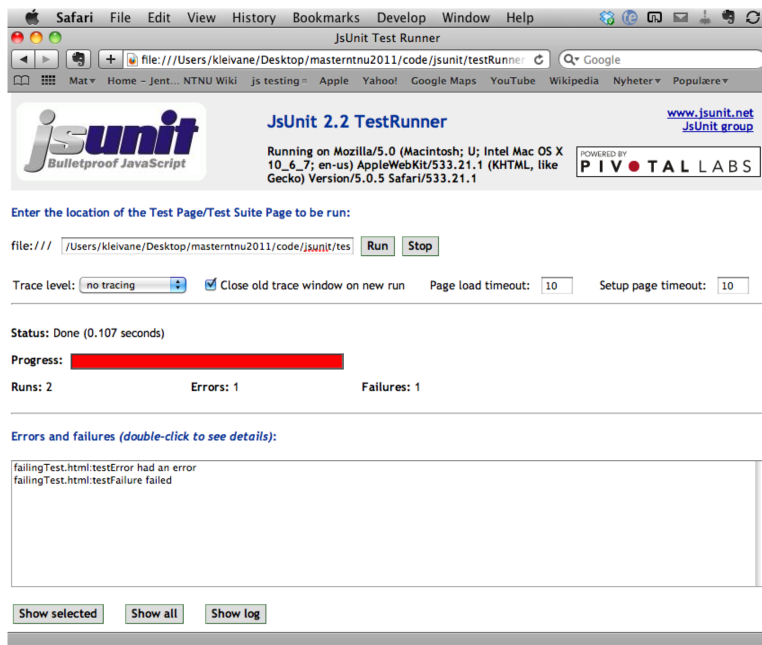

Figure 5.2: The JsUnit test runner.

There is no configuration apart from the file path, and test are written between `script` tags in the HTML file itself, while the code under test is included as separate files. It is also possible to combine a multiple of test pages to a test suite. JsUnit has a simple set of assertions, setup and teardown functionality, but no advanced features. A function for logging and tracking is provided and integrated with the *in-browser* test runner, allowing different priorities of warnings and messages.

[4]As referred to in Section 5.1.

JsUnit also provides a server that can run the tests from JUnit or Ant, to give the user an opportunity to run tests simultaneously in multiple browsers and on remote machines.

5.5 nodeunit

nodeunit is developed by Caolan McMahon and is a test framework developed for testing node.js applications[5][52], but it also supports normal client-side testing in browsers. Its syntax is adopted from the minimalistic style of QUnit[6]

The test library contains a range of assertions, options for grouping test cases, setup and teardown. It tests asynchronous calls out-of-the-box due to its asynchronous nature. The library contains the two special functions `expect(amount)` and `done()`, which are the enablers of this asynchronous testing. Each test needs to specify the number of assertions expected, and if this number does not match the counter when `done` is called, the test fails. The tests are run serial to allow the use of stubs, spies and mocks, though there are no built-in mechanisms to achieve this.

The test runner is available both in the browser and command line. It allows for building a custom test reporter, but also has built-in options for HTML and jUnit XML[7].

5.6 QUnit

QUnit is most commonly associated with jQuery, as it is developed by the same team and is used to test jQuery itself[54]. It is maintained by John

[5] node.js is completely asynchronous and such the test must be the same.
[6] QUnit is described in Section 5.6.
[7] jUnit XML reports are compatible with CI tool Hudson[53].

Resig and Jörn Zaefferer, both from the jQuery team, and is available as open source[76].

It is a test framework consisting of an *in-browser* test runner and a small test library covering assertions, exception testing and organization of test cases[8] with setup and teardown. It can perform asynchronous testing through pausing the tests and has an option for adding HTML fixtures though a special `#qunit-fixture` DOM element in the test runner. This HTML code is reset between each test.

The test runner has a nifty feature for detecting an introduction of a new global variable[9] as well as unexpected exceptions.

The framework can be integrated with test automation tools through a microformat created at runtime. QUnit can be integrated with JsTestDriver, extended with Pavlov, Sinon.JS and others through various adapters[54].

5.6.1 FuncUnit

FuncUnit provides functional testing based on a jQuery-like API[55]. Together with QUnit, the stack provides both low and high-level testing. FuncUnit provides simulated user input for event testing, which can be utilized in acceptance testing[10]. Examples of these actions are a click of a mouse or a drag movement. It can also test CSS attributes like `innerHeigh`, which can test a sites look across browsers. The tests can be run in a browser or through Selenium.

FuncUnit is a part of the JavaScriptMVC[56], but is also available as a standalone product.

---

[8] Referred to as `modules`.

[9] Refer to Section 2.3.2 on why global variables are not preferred.

[10] Section 2.1.2 explained the term acceptance testing.

5.6.2 Pavlov

Pavlov is a BDD test library exclusively for QUnit. It is developed by Michael Monteleone and available under a MIT license[57]. It provides a BDD syntax for QUnit and adds some higher level features. Pavlov has nested test cases and cascading setup and teardown mechanisms, extended matchers and the ability to add new ones. Pavlov also extends QUnit's wait with a `wait(ms, callback)` similar to JavaScript native `setTimeout`.

One of the features unique to Pavlov is the generative row tests and the stubbing of specs, which is both shown in Listing 5.3.

```
1  given([5, 4], [8, 2], [9, 1]).
2      it("should award a spare if all knocked down on 2nd roll"
           , function(roll1, roll2) {
3          // this spec is called 3 times, with each of the 3
               sets of given()'s
4          // parameters applied to it as arguments
5
6          if(roll1 + roll2 == 10) {
7              bowling.display('Spare!');
8          }
9
10         assert(bowling.displayMessage).equals('Spare!');
11     });
12
13  // stubs specs which yield "Not Implemented" failures:
14
15     it("should allow 2 rolls on frame 1-9");
16     it("should allow 3 rolls on the last frame");
```

Listing 5.3: Generative row tests and stubbing of specs in Pavlov, courtesy of Michael Monteleone[57].

5.7 Screw.Unit

Screw.Unit is a BDD test framework developed by Nathan Sobo and is available under a MIT license[58]. It has an *in-browser* test runner and a BDD syntax.

Looking at the library, it has nested describes and cascading before and after statements. It supports custom matchers and has options for specifying preconditions in tests. The library has no build-in mechanisms for mocks, stubs, spies or testing of asynchronous code. Event testing is also left to other frameworks as Prototype or jQuery[59].

It is possible to add custom logging by subscribing to test and the loading of events.

5.8 Sinon.JS

Sinon.JS is a test library designed and maintained by Christian Johansen and available under a BSD license[60]. The library provides test spies, stubs and mocks as well as fake timers, `XHttpRequests` and servers to deal with asynchronous code.

The library itself is a single .js file and can be added on top of existing frameworks. In his book, Test-Driven JavaScript Development[36], the framework used are a combination of Sinon and JsTestDriver. It can also be run with QUnit, Jasmine and nodeunit for node.js with various adapters.

Sinon has a rich set of features that extends the ones of the average test library. This means that simple assertions already existing are not taken into Sinon. It has no assertion mechanism for equals, null and other standards, rather it expands mostly on unit test patterns and faking browser behavior.

5.9 TestSwarm

TestSwarm is a distributor providing continuous testing across browsers and platforms for any test framework. It provides a solution for JavaScript developers to avoid the problems with scaling test environments[11]. The suite was originally designed by John Resig for jQuery testing, but has now become an official Mozilla Labs project[61].

The TestSwarm code is used to test the jQuery project, but it is still in Alpha state, and should not be completely relied on. The tests can be run online[12] or by downloading the source code and setting up a private swarm.

The swarm relies on a pool of clients connected with different browsers and platforms to run the tests. It has a lot of error correction mechanisms in case a client goes down in the middle of a test run, this to ensure the test results are not affected. This form of testing means that no browsers must be opened in order to execute the tests while they are still run in a real browser environment. As mentioned, TestSwarm needs test framework to handle execution on the client. Currently QUnit, UnitTestJS, JSSpec, JSUnit, Selenium, and Dojo Objective Harness are supported[61].

A run of test swarm may look like Figure 5.3, where the browser icons on top indicate different versions or different platforms. Red indicates at least one failing test, gray are tests not run and green are passed tests.

[11] i dependencies and j browser versions on k platforms as discussed in Section 4.1.
[12] Through the site `http://swarm.jquery.org/`.

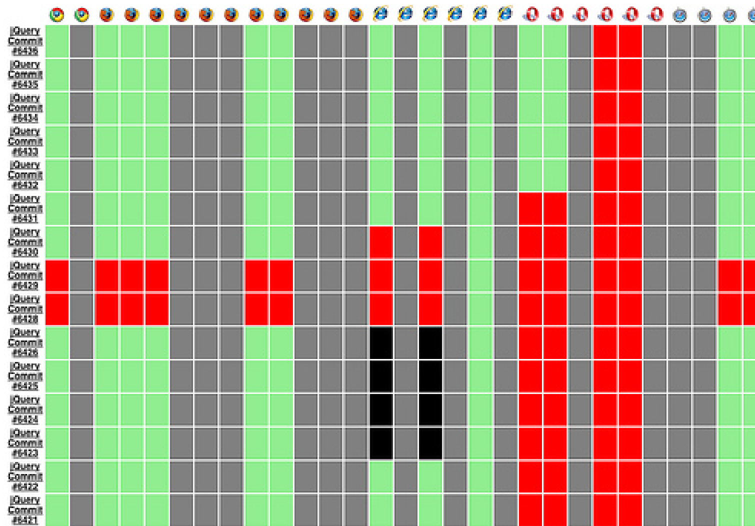

Figure 5.3: An example run of multiple commits in TestSwarm. Courtesy of John Resig[61].

5.10 YUI Test

YUI Test is a part of the YUI Library, which is

> a set of utilities and controls, written with JavaScript and CSS, for building richly interactive web applications using techniques such as DOM scripting, DHTML and Ajax[62].

It is available in two versions YUI 2 and YUI 3, where the latter is the one discussed here. The complete YUI library and its individual modules are available under a BSD license.

YUI Test contains a range of assertion mechanisms, options for mocking, exception and asynchronous testing and grouping of test cases.

Through the YUI Event library it is also possible to simulate some key, mouse and UI events[63, 64].

The *in-browser* test runner within YUI Test is run easily, but the options for viewing the result or exporting are many. For viewing the results in

a browser, they can either be pushed to a console, like Firebug[40], or be viewed through Yahoo!s own Console widget. YUI Test can also export to JSON, JUnit XML[13], YUI Test XML and TAP. It is also possible to post these exported results to a server. You can also build your own result viewer, as it is possible to subscribe to events posted by the runner-object.

5.11 Others

Other frameworks and libraries exist, but are not discussed in this chapter. This is due to a combination of factors, among them terminated support, time since last release and dependencies. Selenium is a special case, but is considered out-of-scope, as it does not use JavaScript.

- JsMock[65] : Mock object library last updated in 2007.
- Jack[66] : Mock and stub library last updated in 2009.
- JSSpec[67] : BDD test framework last updated in 2008.
- Crosscheck[68] : unit-testing framework capable of emulating multiple browser environments. No longer supported.
- Smoke[69] : older mocking and stubbing library for Screw.Unit.
- Selenium[6] : automated testing across platforms. Can use Java, C# and others to write browser tests, as well as record and playback testing[14].
- mockjax[70] : library for mocking jQuery's `ajax` method.

[13] jUnit XML reports are compatible with CI tool Hudson[53].
[14] The program records a browser session and plays it back when the test is initiated.

5.12 Feature mapping

To give a comparative overview of the frameworks and libraries mentioned in the above sections, Tables 5.1 to 5.3 sum up important characteristics. To give a realistic summary, the standalone test libraries have been given a test runner to make a complete framework. In Table 5.3 some features are not compared; the general assertions, setup and teardown mechanisms and organization of tests. They are removed because all the frameworks supported these features with only minor variations. The term fixture in this context also refers to HTML fixtures. Other features are i.e. the opportunity for custom matchers/assertions and the ability to register for events on the runner. These have been considered at the other end of the scale, as these are not needed often and would clutter rather that clarify the big picture.

Sinon.JS is the only framework-independent test library, and it is compared on top of frameworks Jasmine, QUnit and JsTestDriver. The two former have adapters for Sinon, and JsTestDriver can use Sinon out-of-the-box. It is possible to use the library with other frameworks as well, but the feature mapping in this case is left to the reader. TestSwarm is excluded, as it is a pure test distributor, not a testing framework.

Table 5.1 describes three differentiator; syntax, execution and environment. Execution mode is stated as in-browser or console in the environment column. Table 5.2 described characteristics important to judge the framework project, such as stable release and support. Table 5.3 describes the features in the last differentiating factor, the test library.

The criteria in Table 5.2 are stable release, support and community. These are chosen to be able to judge whether the framework is actively maintained, and if not, has a dedicated community that can continue the development. Support in this context means that the framework in under active development. The stable release and support is judged based on the information

found online, either at social coding spaces such as GitHub or the developers personal sites. Dead links, a long timespan between updates, old jQuery version are among others judged as a sign of terminated support. The community is represented by last GitHub commit or Google Group activity level. It could be argued that export formats is an important characteristic deserving a column, but this is considered out of scope. It would be beyond the research question of the thesis to set up and test out a CI environment.

All the material supporting the information in the tables can be found in the bibliography of the specific framework's section. Where this is not the case, references are provided. The icons are a part of the Silk icon package, created by Mark James[71].

Framework	Additional library	License[a]	Dependencies	Syntax	Environment	Node
Jasmine	-	MIT	-	BDD	In browser[b]	✓
	Sinon.JS	BSD	-	BDD	In browser	✓
JSpec	-	MIT	-	BDD DSL	Rhino, browser or console	⊖
JsTestDriver	-	Apache License 2.0	-	JavaScript	Browser through console	⊖
	Sinon.JS	BSD	-	JavaScript	Browser through console	⊖
JsUnit	-	GNU GPL 2.0	-	JavaScript	In browser	⊖
nodeunit	-	MIT	node.js	JavaScript	node.js, browser through console	✓
QUnit	-	GNU GPL 2.0 & MIT	-	JavaScript	In browser	⊖
	FuncUnit	Unknown	-	JavaScript	In browser or with Selenium	⊖
	Pavlov	MIT	-	BDD	In browser	⊖
	Sinon.JS	BSD	-	JavaScript	In browser	⊖
Screw.Unit	-	MIT	-	BDD	In browser	⊖
YUI Test	-	BSD	-	JavaScript	In browser[c]	⊖

Table 5.1: General characteristics of combined frameworks.

[a] Where an additional library is present, the library's license is stated. All licenses are open source[72, 73, 74, 75]

[b] Jasmine also has versions for Maven, JsTestDriver and Ruby.

[c] Other options exist, refer to Section 5.10

Framework	Additional library[a]	Stable release	Support	Community[b]
Jasmine	-	✔	✔	9 Mar 2011[77]
	Sinon.JS	✔	✔	2 May 2011
JSpec	-	✔	⊖	30 Sep 2010 [78]
JsTestDriver	-	✔	✔	High
	Sinon.JS	✔	✔	2 May 2011
JsUnit	-	✔	⊖	18 Feb 2010
nodeunit	-	⊖	✔	17 Mar 2011
QUnit	-	✔	✔	20 Apr 2011[79]
	FuncUnit	✔	✔	2 May 2011[80]
	Pavlov	✔	✔	29 Mar 2011
	Sinon.JS	✔	✔	2 May 2011
Screw.Unit	-	✔	⊖	29 Dec 2010
YUI Test	-	✔	✔	18 Apr 2011[81]

Table 5.2: Release, support and community characteristics of combined frameworks.

[a] Where an additional library is present, the following columns will contain info about the library.
[b] Latest GitHub update or Google Group activity level as of 2 May 2011.

Framework	Additional library	Stubs	Mocks	Spies	Pause tests	Fake time	Fake Ajax	Exception testing	Fixtures[a]	Event testing
Jasmine	-	✓	✗	✓	✓	✗	✗	✓	✗	✗
	Sinon.JS	✓	✓	✓	✓	✓	✓	✓	✗	✗
JSpec	-	✓	✗	✓	✗	✓	✓	✓	✓	✗
JsTestDriver	-	✗	✗	✗	✓	✗	✗	✓	✓	✗
	Sinon.JS	✓	✓	✓	✓	✓	✓	✓	✓	✗
JsUnit	-	✗	✗	✗	✗	✗	✗	✗	✗	✗
nodeunit	-	✗	✗	✗	✓	✗	✗	✓	✗	✗
QUnit	-	✗	✗	✗	✓	✗	✗	✓	✓[b]	✓[c]
	FuncUnit	✗	✗	✗	✓	✗	✗	✓	✓	✓
	Pavlov	✗	✗	✗	✓	✗	✗	✓	✓	✓
	Sinon.JS	✓	✓	✓	✓	✓	✓	✓	✓	✓
Screw.Unit	-	✗	✗	✗	✗	✗	✗	✗	✗	✗
YUI Test	-	✗	✓	✗	✓	✗	✗	✗	✗	✓[d]

Table 5.3: Library features of combined frameworks.

[a] In browser test runners, HTML fixtures can be achieved through manual setup and teardown.

[b] The QUnit runner even has a special `#qunit-fixure` element set up for this purpose which is reset for each test. This also applies for QUnit in combination with other libraries.

[c] Through jQuery. This applies for QUnit in combination with Pavlov and Sinon as well.

[d] Through YUI Event[64, 63].

5.13 Summary

This chapter has dived into the state of the art; the actual libraries, frameworks and runners available for TDD and BDD in JavaScript. They have been explained based on the framework differentiators, library, syntax, execution and environment, found in Chapter 4. As a summary, Tables 5.1 to 5.3 give a comparative, but non-exhaustive, overview of complete frameworks available to developers today. Other libraries and tools were considered, but found out of scope due to multiple factors, described in Section 5.11.

Chapter 6

Results

This chapter describes the results achieved when applying the research method outlined in Chapter 3. Section 6.1 introduces the test cases and Section 6.2 discusses the programmatic results of applying the cases to a selected subset of the frameworks described in Chapter 5.

6.1 Test cases

Four test cases were found to cover both the general issues and the issues specially related to unit testing in JavaScript. The test cases are created to test the frameworks as a proof of concept. They are not extensive nor complicated cases, so they will be easy to repeat when new frameworks need to be evaluated.

In order to get a complete picture of a framework, all the cases need to be executed. An exception could be the *FizzBuzz* case presented in Listing 6.1, as its goal mainly is to explore the basic features, setup and execution in a framework. An experienced developer might feel that the features will be familiarized through the remaining test cases and can choose to skip this test case.

The pseudocode presented is not intended as a template for the program, as that would be against the design techniques of TDD. It is meant as a high level description and to give the developer enough knowledge of the program in order to start the TDD process.

The tests are run in a single browser, unless anything else is specified. This is because each test case only cover a subset of the issues related to JavaScript, and cross-browser[1] discrepancies are one of them.

6.1.1 FizzBuzz

Covers setup, asserts, test organization and execution.

Listing 6.1 contains the pseudocode for a popular TDD code kata[2] named *FizzBuzz*. In the context of applying TDD to a new JavaScript framework, this test case allows a developer to get to know the configuration and installation, execution, syntax, assertion mechanisms and test case structure before diving into more complicated problems. It is possible to explore cross-browser running as well, but does not have to be a part of the execution, as no cross-browser issues exist.

```
1 SET number = user inputted number
2 CASE number OF
3   modulo 3 : Print Fizz
4   modulo 5 : Print Buzz
5   modulo 3 and modulo 5 : Print FizzBuzz
6   OTHERS
7   Print number
8 ENDCASE
```

Listing 6.1: Pseudocode for test case 1: "FizzBuzz".

[1]Cross-browser issues are covered in Section 2.3.1 and 4.1.

[2]A code kata is an exercise in programming which helps hone your skills through practice and repetition[82].

6.1.2 Alias

Covers testing of time and stubbing.

Listing 6.2 contains the logic for a round of the Alias board game. The game gives a team member 60 seconds to explain as many words as possible to his or her teammates without saying the word itself. Answers given after 60 seconds should not be awarded points. The words are read from a deck of cards.

An implemented version will have a call to `setTimeout` to invoke the end of a round after 60 seconds, and it will need to connect to an external database to fetch new words.

The framework will need to provide a solution to the timer without having the developer wait 60 seconds[3]. The framework also needs to stub the database to allow the test to run isolated. If not, the test would be hard to write due to setup of a word database. Since this is only testing the timing mechanism, stubbing the database will give the timing sufficient validation.

```
1 SET points = 0
2 SET timer to 60 seconds
3 REPEAT
4   Fetch new word
5   IF right answer THEN increment points by one ENDIF
6   IF wrong answer THEN decrement points by one ENDIF
7 UNTIL timer ends
```

Listing 6.2: Pseudocode for test case 2: "Alias".

[3]As this would be against the unit testing requirement of speed, described in Section 2.1.

6.1.3 Twitter

Covers testing of Ajax calls and exceptions.

Listing 6.3 shows the logic of a function that uses Ajax to retrieve the current Twitter trends in JSON format. If a failure occurs, the string *'Unable to connect to Twitter'* will be printed.

Ajax are one of the main usages for JavaScript and needs to be faked or otherwise abstracted to achieve repeatable tests[4]. If a failure occurs on a Twitter server, our tests should not fail, indicating proper test isolation. Neither should the test need to wait for the result of the call, as this is against the unit test requirement of speed.

The code in Listing 6.3 will test the framework's ability to deal with an Ajax request. It will also test how well the framework handles throwing exceptions and tests where exceptions are the correct result.

```
1 BEGIN
2   fetch Twitter trends
3   Print: top 2 trends
4 EXCEPTION failed fetch or empty result
5   Print Unable to connect to Twitter
6 END
```

Listing 6.3: Pseudocode for test case 3: "Twitter".

6.1.4 DOM manipulation

Covers cross-browser issues and HTML fixtures.

Listing 6.4 is the only test case that interacts with the DOM. The program fetches the element where `id = "text"`, validates its content and changes its text.

[4]As discussed in Section 2.1.

The framework will be tested for its ability to include fixtures, in this case an HTML element. It will also test how easy it is to discover cross-browser issues. `innerText` is a property containing the text content of a node and it exists in all browsers, but Firefox[83]. This test case will need to run in Firefox and at least one more browser in order to evaluate the framework.

```
1 Find HTML element with id = text
2 Check that content of element is 'SomeContent'
3 Update text content of element through innerText
```

Listing 6.4: Pseudocode for test case 4: "DOM".

To be able to test this correctly, the browser's same-origin policy[5] needs to be disabled in order to access the attributes of the XHR object. This can be done in Chrome by opening the program with the `-disable-web-security` flag.

6.2 Execution

To decide which frameworks should be used for executing the test cases, the important characteristics in Section 5.12 were reviewed. Rather than an opt-in solution, opt-out was chosen. Characteristics that singled out a framework would be the absence of support, community or stable release and missing library functions[6]. Also, some testing frameworks targeted testing on a higher level than the xUnit frameworks[7] and were as such disregarded. In total, the following frameworks were opted out:

[5] A policy that only allows scripts from the same site to access each other's methods and parameters. Scripts from different sources cannot access each other.

[6] As summarized in Table 5.3

[7] Discussed in Section 2.1.1.

- Standalone JsTestDriver - Table 5.3 reveal few library functions and the framework is most interesting when paired with a more feature-rich library, as Sinon.JS.
- JsUnit - the framework is no longer supported and is outdated both in runner and features, as seen in Table 5.2 and 5.3.
- nodeunit - the framework does not yet have a stable release and is primarily written to test node.js. It also has a minimal notion of features.
- QUnit and Pavlov - Pavlov only adds a BDD syntax and minor variations to QUnit, so testing standalone QUnit is sufficient to demonstrate its capabilities.
- QUnit and FuncUnit - FuncUnit provided scripted UI testing on a coarser granularity[8], and is such out-of-scope for unit testing through API.
- Screw.Unit - it is no longer supported and also lacks all features discussed in Table 5.3.
- TestSwarm and any additional test framework - since it only distributes the tests, it is considered out of scope.

The remaining frameworks are used to create the test cases in Section 6.1 using a TDD approach. The source code for the test cases can be found in attached to the thesis.

The frameworks were not executed in the order that follows in the remaining section, rather it was done iterative as described by the process in Chapter 3.

In the evaluation of each framework, each section is built on the following layout:

[8]Referring to Figure 2.1 on page 7.

- Getting started: setup, syntax and execution through the *FizzBuzz* test case.
- Execution of test cases 2 to 4.
- Documentation.
- Issues and/or oddities.
- Summary

6.2.1 Standalone Jasmine

Setting up Jasmine is easy as it has an *in-browser* test runner and a test project available for download from GitHub[77]. Figure 6.1 shows an example of the test runner after a passed run of the FizzBuzz test case.

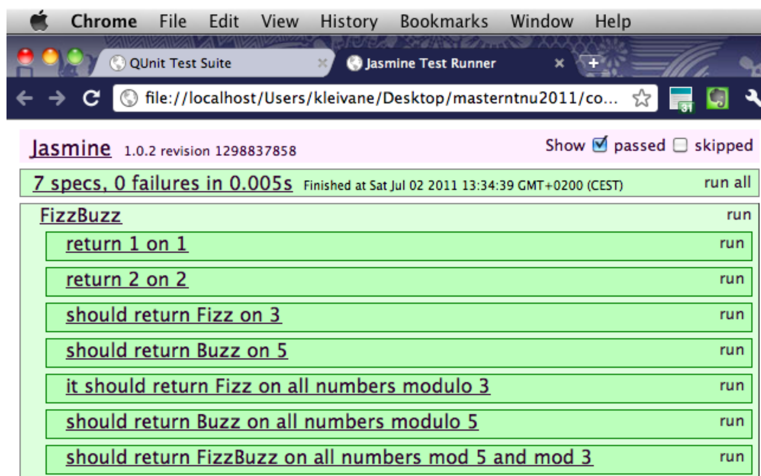

Figure 6.1: The FizzBuzz test case passed in Jasmine.

The Jasmine framework can test asynchronous code, but is dependent on `wait` and `run` methods that pauses the tests for a given time. This period of

waiting, which in the Alias test case is 60 seconds, feels wasted and there is a risk that the developer writes more code, either in tests or source, before the test result returns.

A partial solution is that a test can be excluded from running. This is done by changing the test function `it()` to `xit()`. This is not a recommended practice, but it means that you can write one async test case, run it, then skip it, instead of not running tests at all because they are slow.

A way of solving the problem is to abstract away the use of timeouts, as done in Appendix Section B.1.1. This is a solution were the call to the timeout function is ignored, but the inner function of the timeout mechanism is executed. This gives lower test coverage, but allows a faster progress. The original implementation is attached in Section B.1.

The stubbing of the word database works nicely due to the `spyOn(object, method).andReturn(value)` functionality.

The Twitter test case is performed using Jasmine's spying function, which essentially are stubs and spies bundled together. By stubbing the XHR object, behavior verification is performed on the `fetchTrends` `open` and `send` methods. The `onreadystatechange` is not tested through the browser implementation, rather it is triggered manually by a lightweight fake XHR. The spy methods of Jasmine made a difference here, easing a lot of the testing, especially of exception handling.

The DOM test case has some interesting aspects. Since Jasmine has no support for fixtures, care must be taken to make the tests isolated from each other. The setup and teardown must insert and delete a node in a way that works the same across all browsers in order not to introduce more dependencies and different behavior.

As Jasmine has a *in-browser* test runner, all browser tabs needed a manual refresh to rerun the tests in the DOM test case. This was a hamstring as

the number of browsers increases and the screen space available lessened for each.

```
insert = document.createElement('div');
insert.id = 'text';
var insertText = document.createTextNode('SomeContent');
insert.appendChild(insertText);
document.body.appendChild(insert);
```

Listing 6.5: Part of the setup method in Jasmine

An inconvenience with the Jasmine test runner is that is shows only a failed assertion. If a test has some passing assertions and some failing, only the failing ones will be shown, hiding potentially important information. It has some oddities with regards to vocabulary, the implemented spy object incorporates spies, stubs and mocks[9]. Also errors in the async test cases are not revealed until the test has finished. So if test breaks on the first line of code, the runner still waits the given time seconds before reporting it.

Summarizing, Jasmine has a lot of functionality and is easy to use, but is missing support for faking Ajax and time, as well as HTML fixtures.

6.2.2 Jasmine and Sinon.JS

In these tests Jasmine was run through its Ruby gem. The gem allows you to make a .yml file that defines what folders should be monitored and files imported, alleviating the need for `script` tags for test and source files as in the HTML test runner. Adding Sinon comprises of saving the sinon.js file in the right folder and potentially add a set of matcher fitting to the Jasmine syntax[84]. This has been done in the following tests. A problem with these matchers is that they override Jasmine's own functions `toHaveBeenCalled` and `toHaveBeenCalledWith`, so it forces the developer to use Sinon's stubs, mocks and spies.

[9]Jasmine mocks are discussed in Section 7.5 on page 128.

Running the tests consider of a `rake`[10] command which continuously updates a Jasmine *in-browser* test runner. The test runner is opened in the browser and is rerun on refresh. This is very convenient, because it automatically discovers new files and this alleviates the risk of loosing tests due to missing `script` tags or misspelled file paths.

The Alias test case uses the Sinon methods for faking time, not the Jasmine `waits` and `runs` methods, alleviating the need to wait 60 seconds to see the results. Setting up the timers can be a bit difficult and combined with closure issues it has been one of the biggest challenges: `setTimeout` does not maintain a reference to `this`, meaning the inner function in the timeout call can only see the global scope.

```
1  var alias = (function () {
2    [....]
3    var inPlay = false;
4    return{
5      newGame : function () {
6        this.points = 0;
7        this.inPlay = true;
8
9        setTimeout(function () {
10         alias.endGame();
11       }, this.SIXTY_SECONDS);
12     },
13     endGame : function () {
14       this.inPlay = false;
15     }
16   }
17
18 })();
```

Listing 6.6: Invoking endGame from the setTimeout function through the global scope.

[10]`rake` is a Ruby utility that, among others, automatically builds executable programs.

This is simple JavaScript, but demonstrates the difficulties with closures, as described in Section 2.3.2.

The Twitter test case could test the `onreadystatechange` attribute, due to Sinon's faking of Ajax requests. The earlier `printTrends` method was changed into a parse method, which had functions for returning the top n trends. Since the return object had associated methods, the exception return string "Unable to connect to Twitter" had to be remade into an object with a message attribute containing the same text.

The setup of the fake `XmlHttpRequest` object can be difficult. There are a lot of functionality to keep up with, and making mistakes is easy. At one point support functions were made to make sure that the `this` references were correct, and this can be an idea to keep clutter out of the test functions and improve readability.

The DOM test case is the same as with standalone Jasmine, and such it is not presented.

Adding Sinon.JS to Jasmine alleviates most of the original shortcomings of Jasmine. It is still missing HTML fixtures, but is a powerful testing tool and with the gem, the risk of loosing tests is minimal.

6.2.3 JSpec

Setup is easy due to the JSpec .zip file containing an executable. To create a new project, JSpec has an `init` command that initializes a directory containing a test project. Further, the command `jspec` monitors the JavaScript files and opens the HTML test runner in a new browser tab whenever a change occurs. The test and source files also need to be referenced in the `dom.html` file and such, when adding a file, the file must be added here and the `jspec` command run again.

JSpec has a custom DSL and a JavaScript syntax, where the former is used in the test cases. An example can be seen in Listing 6.7.

```
describe 'FizzBuzz'
  describe '.fizzbuzz()'
      it 'should return null on null'
        fizzbuzz(0).should.equal 0
      end
  end
end
```

Listing 6.7: The first test in the FizzBuzz test case in JSpec.

The grammar is easy to read, but it can be difficult to grasp the syntax rules, especially on the use of semicolons and `var` statements. This part of the framework is sparsely documented, but the site contains some helping examples.

The second test case, the Alias game, had the smoothest coding of all the frameworks; `tick(sec)` worked on the first try, faking the implementation of time. The stubbing of `newWord` works easily with the built-in stubs.

The mocked Ajax object is created by a single line: `mock_request().and_return()` and is restored automatically. The mocked object works as expected and with no hassle. One test is done with a successful response and one is done with an empty response. Two ways of testing an unsuccessful fetching were done, one with an empty response text and one by manually stubbing the `fetchTrends` method.

The fourth test case utilizes the `fixture()` method in JSpec. As expected, the tests behave differently in the different browsers. The use of the fixture can be seen in the extract in Listing 6.8. A setup and teardown still has to be done, but the creation of the inner HTML content is all in the `node.html` file, making it significantly more readable, and a lot simpler than the setup in Jasmine shown on page 89. The possibility of writing HTML to

create the fixtures, and not creating it with JavaScript is an advantage that grows as the HTML gets more complicated.

```
\\In the test file

before_each
  x = document.createElement('div')
  x.innerHTML = fixture('node.html');
  document.body.appendChild(x);
end

\\In node.html
<div id="text">SomeContent</div>
```

Listing 6.8: Setup with the use of JSpec's `fixture` method.

A problem occurring with the `jspec` command has to do with the number of files in the project. When this number gets too high, the monitoring for changes fails and has to be restarted with a smaller number of files under surveillance. Also, test runner has a way of invoking the tests that triggers the cross-site scripting protection in Chrome, so another browser had to be used, except for the Alias test case where Chrome was run with the `-disable-web-security` flag.

The `init` command did not add all the library files necessary to fake Ajax calls and the timing functions. This was also not documented on the site, which caused some frustration at the beginning.

JSpec has some difficulties with lack of documentation on syntax and setup, but when this is done, it supports both fake Ajax and faking of time as well as having a rich set of library functions. It also has a great advantage when working with large HTML files with its `fixture` function.

6.2.4 JsTestDriver and Sinon.JS

The setup with this framework is a bit more complicated than the others. A ruby gem can bee used, the `jstdutil` from Christian Johansen[85], to make the process a bit less tedious[11]. Adding Sinon meant only to download the JavaScript source code and add it to the .config file, as well as minor changes to the setup of test cases.

The FizzBuzz test case revealed that tests had to start with the word 'test', causing some initial fury prior to the discovery. Also some JavaScript errors are not displayed, but it helps that the framework provides a `jstestdriver.console.log` that outputs to terminal.

The Alias test case used Sinon's methods for faking the native browser timers. Getting this right proved difficult for the same reasons as the Jasmine case. The `setTimeout` function was not understood well enough, and so it tried to reach variables bound to `this`[12]. When these difficulties were sorted out, the timing functions worked as expected. The stubbing of the word database worked very well with the Sinon stub functions.

The third test case also posed some challenges. Sinon has both a fake server and fake XHRs making it difficult to choose the best for testing. The fake server was used and even though the setup was easy enough, it took a lot of effort before it worked correctly. This had to do with the combination of the `server.respond()` method, which triggered the Ajax response and the global `trend` variable that was used to record the responses. The original design did not consider the asynchronous behavior, but the manual response invocation in test made this emergent, causing the design to change.

[11]But JsTestDriver and the code examples can be run with the standard setup found on the Google Group[86].

[12]`this` is bound to the global scope in the `setTimeout` function.

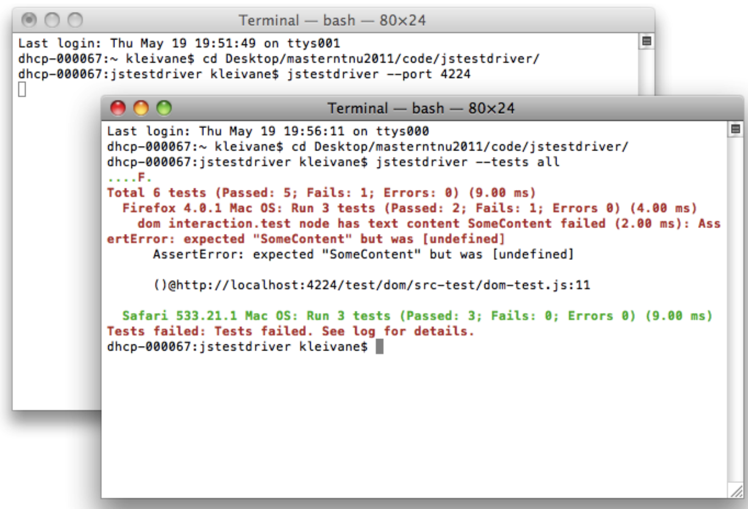

Figure 6.2: The DOM test case in with JsTestDriver and Sinon.

The result of the DOM test case can be seen in Figure 6.2. It is easy to see how it fails in Firefox and passes in Safari, indicating a cross-browser issue. Listing 6.9 shows the setup function with DOM insertion of the node. This is easy and very readable as long as the HTML is this small. No teardown is needed.

```
1 setUp: function(){
2   /*:DOC += <div id="text">SomeContent</div> */
3 },
```

Listing 6.9: Insertion of a `div` element in setup before each test method.

An annoyance became apparent when the test suite changed frequently, as this requires the runner to be restarted and all browsers captured again.

JsTestDriver diverges from the other frameworks with its console runner, which provides easier cross-browser testing, but lacks the interactive exploration found in browser consoles. Sinon provides all the features necessary and with the HTML fixture syntax, the framework is easy to use. Personal preferences and experience would decide how well this framework appeals.

6.2.5 Standalone QUnit

QUnit has a very minimalistic syntax with the least functions and syntactic sugar of all the tested framework. The setup is easy, downloading from GitHub and open the `index.html` in a browser. QUnit automatically outputs errors to the browser's console, making a significant improvement to debugging.

The Alias test case utilizes the async test case option of QUnit. This means that a wait of sixty seconds is needed in order to validate the correctness of the tests. This forces ahead a design decision of not hiding the `game.inPlay` variable. By setting this manually, the content of the timeout is tested and the test run with no waiting time. This exposure of the variable cannot always be done and it also does not test the actual invocation of the `setTimeout` call. The problem is the same as in the standalone Jasmine; it does not test that the call to timeout actually happens and if the containing function is correctly implemented in this scope. To do this, an asynchronous test, an abstraction or a way of faking time is needed. QUnit has no way of selecting which tests to be run in a module, so if complete test coverage needs to be achieved, the programmer must wait 60 seconds for each result.

The third test case was different, as QUnit has no support for faking Ajax calls. The `fetchTrends` method got an extra input, an `XmlHttpRequest`, done in order to use behavior verification to assure the `send` and `open` calls. This could have been done differently, e.g. by faking the prototype of the real `XmlHttpRequest`.

In the fourth test case the HTML element is inserted in the HTML test runner as a child element of `<div id='qunit-fixture'>`. QUnit automatically resets the contents of this element before each test, relieving the programmer of manually cleaning up. The pure HTML test runner has the same issues as Jasmine, when testing needs to be done on a number of browsers for each test.

The QUnit test runner has a readability that stands out and the presentation of messages in tests and assertions are well worth writing. A minor annoyance and hinder to readability is QUnit's insistence on writing `equals(actual, expected)`, instead of the more standardized `equals(expected, actual)`.

The test runner also has some other nice features, among them automatic failure if a global variable or any try-catch statement is introduced. The problem with global variables was introduced in Section 2.3.2, so this feature can help enforce code standards and increase quality of the tested code. The test runner also has a nice way of showing the differences between expected and actual values, as shown in Figure 6.3.

QUnit is lacking support for testing time and Ajax calls, but has good support for HTML fixtures. Its test runner appeals with its readability and easy syntax and setup makes this easy to use for testing *simple* JavaScript application.

6.2.6 QUnit and Sinon.JS

Adding Sinon to QUnit means downloading Sinon source and a QUnit extension and loading these in `script` tags in QUnit's *in-browser* test runner. The test methods are wrapped in a Sinon test object, making the functions of Sinon available.

The Alias test needed reworking for the fake timers to work. The clock needed to be stored in a separate object, created by an immediately executed function, as shown in Listing 6.10. This was different from the other frameworks with Sinon, and created a lot of initial confusion. The remaining test case worked as expected, with faking the timers and avoiding the 60 seconds wait.

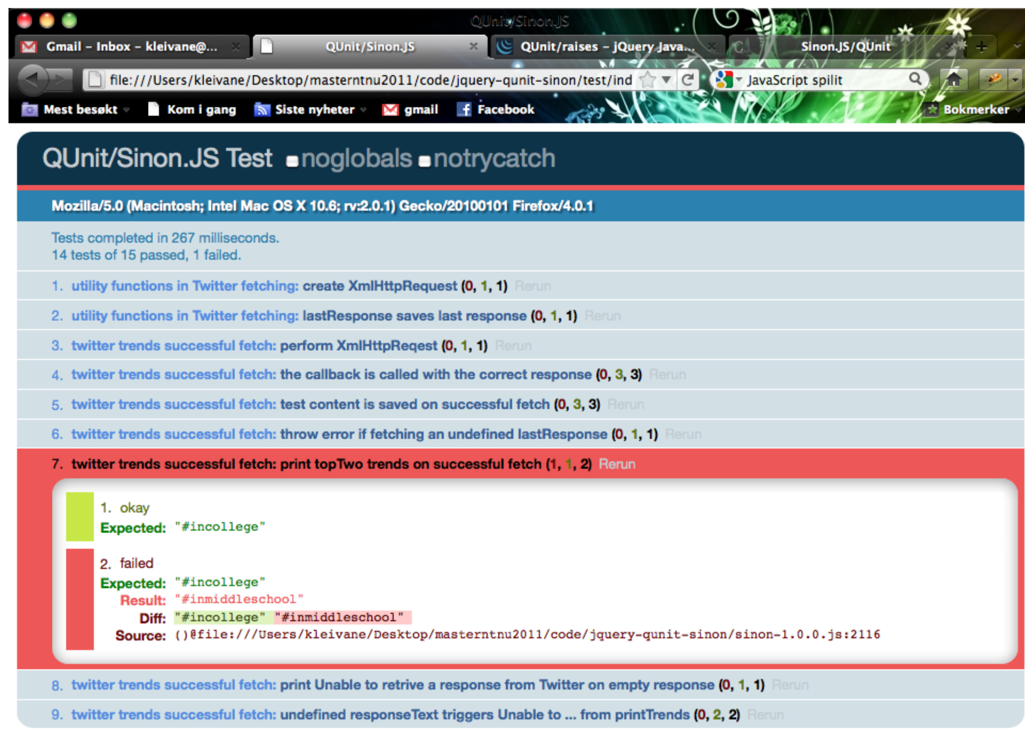

Figure 6.3: The test runner in QUnit. Note the presentation of differences of expected and actual result and the presentation of all assertions in a failed test.

```
var tinesfakeClock = (function () {
  var fakeClock;
  return { useFake : function () {
    this.fakeClock = sinon.useFakeTimers();
  },
    restore : function () {
      this.fakeClock.restore()
    },
    get : function () {
      return this.fakeClock;
    }
  }
})();
```

Listing 6.10: Creating a common `this` environment for the fake timers.

The Twitter test case tests all method calls and invocations with Sinon's capabilities for simulating XHR objects. Even the `onreadystatechange` method that was difficult with the standalone QUnit tests, are tested.

The DOM test case is the same as with standalone QUnit, and is not presented.

Adding Sinon to QUnit extends the frameworks ability to test asynchronous functions and leaves an almost complete framework for unit testing with TDD.

6.2.7 YUI Test

The setup of YUI Test was supposedly simple; create an HTML page, include a given set of dependencies, create a global YUI instance and run the tests. What is completely missing is how to display the results. A lot of time was spent digging through examples and looking at source code before it was found that an extra module, the YUI Console, needed to be imported before. It has to be mentioned that outputting to a browser console was managed

pretty fast, but this is not a viable option in all browsers. The setup can be seen in the attached source code, as the YUI Test does not need anything else than the .html test runner file.

It was also discovered that the test names needed *test* or *should* in them in order for the test runner to recognize the tests.

The second test case, Alias, uses the `wait` function, that works the same way as the native JavaScript `setTimeout`. This means a wait of 60 seconds is needed before the results are done and published. This gives the same problems as with standalone Jasmine and QUnit.

The third test case uses the same approach as the standalone QUnit, only that it uses a built-in mock for verifying the `open` and `send` calls. Since the send calls only are mocked with a fake XHR there are no way of testing the invocation of the `onreadystatechange` event, as this is triggered in the browser. This has resulted in a design that invokes the `saveLastResponse` with the response text or null if no response text is given. This implies that the `onreadystatechange` function will trigger one of the calls to `saveLastResponse`. With this in mind, testing for exceptions and correct results is done by faking calls to the `saveLastResponse` method, as if the fetch was already done and the save was triggered from there.

The fourth case, the DOM test case, suffers from the same problems as standalone Jasmine, as the fixture elements need to be created by hand. The setup of the fixture increases the chances of introducing errors or dependencies across tests. The *in-browser* runner in YUI Test also suffers from the same problems as the other *in-browser* test runners; as the number of browsers increase, it is harder to see a correct result at a glance. This is heightened even more with the problematic size and need for scrolling with YUI's test runner, as described below.

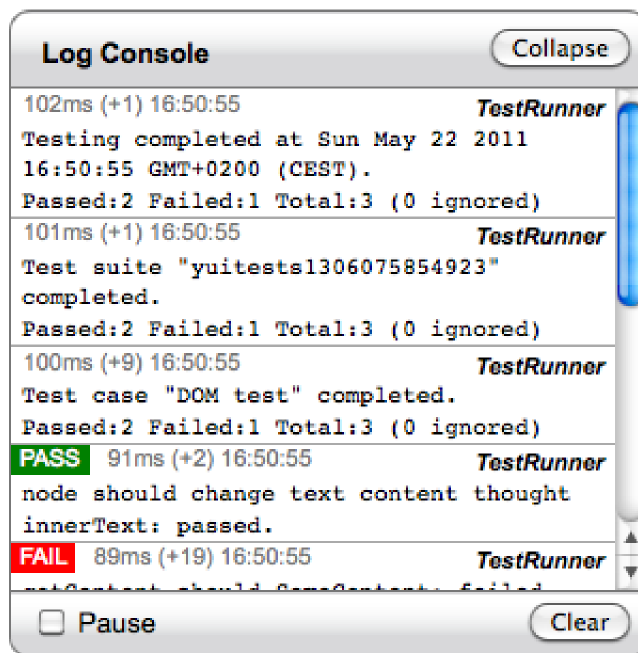

Figure 6.4: The Console module in the test runner outputting YUI test results.

When the console is set up, and the tests start to populate, the test viewer is too small by far. Looking at Figure 6.4, the DOM test case contains three tests, and only two are possible to see. This means that the developer actively needs to scroll to see if any test cases failed. It was tried to change the height of the console object and verbose output was specifically turned off, but nothing worked. It is possible to output the test in "newest first" order, but developers are still unable to see if this breaks an existing, older test without scrolling. This is very much against the TDD process and unit testing properties described in Section 2.2 and 2.1.

Other oddities consist of

- Test cases need to be added to a special runner object, which heightens the chance of "loosing a test" by forgetting to add it to the runner.
- The tests are written in `script` tags inside the HTML, lowering readability and acting against the separation of concern.
- The syntax is very verbose, YUI Test uses `Y.Assert.isTrue` as opposed to JsTestDriver's `assert` or QUnit's `ok`.

YUI Test has some oddities making it difficult to work with on a regular and effective basis. It also lacks an option for faking time and Ajax calls, and an option to add fixtures. The lack of separation between the test runner and the test code also proved a hindrance to readability.

6.3 Summary

To cover general issues and problems related to JavaScript, four test cases were developed to cover the areas of

- Testing time

- Testing Ajax calls
- HTML fixtures
- DOM manipulation
- Exception testing
- Cross-browser testing and issues

General tools like stubs, mocks and spies were indirectly used, as these are the tools used to achieve the above tests.

The tests cases were applied to a subset of the frameworks presented in Chapter 5. The selection was done on an opt-out basis with the feature mapping in Section 5.12 as a backdrop.

The result of applying the tests was noted, and both expected and unexpected problems emerged. A broad category was issues related to testing of time and Ajax, and the coverage of these tests in frameworks with no or poor solutions for asynchronous testing.

When testing cross-browser issues, the *in-browser* test runners represented an issue, but also the general presentation of the test results was surprisingly different and could act as a deal-breaker when testing the framework.

Chapter 7

Discussion

This section reflects and discusses around the results presented in Chapter 6 and uses these to answer the Research Questions 2 and 3. It also discusses the chosen method and its limitation when looking at the results.

7.1 Limitations of the research design and material

The limitation of the research method was its novelty. No known testing had been done on JavaScript unit testing frameworks, so the research method had little basis in applied research, though the notion of proof-of-concept and test cases are a documented approach.

Since no previous studies were found, a threat to the validity of the project is the risk of undiscovered studies or JavaScript unit testing frameworks.

Given that the level of personal experience was low when starting this thesis, other developers might execute the cases and get results that differentiate. This is a drawback of the research method and must be considered when discussing and concluding the findings.

Other limitations are the areas where the frameworks have not been exercised, such as:

- No connection to CI systems.
- No running on remote machines or different OS's.
- No real code dependencies from the code under test.
- No testing in other environments, like Rhino with Env.js.

A missing aspect of in the test cases is interaction with CSS properties. These could have be included in order to gain a more complete insight on the interaction characteristic of JavaScript testing. On the other hand, CSS properties are used for manipulation of UI, which indicates testing on a higher level than the xUnit frameworks that have been considered for this project.

The level of the test cases can also be discussed. The Twitter test case tests the invocation of the `open` and `send` methods on an XHR object. This is not a level an application programmer would usually test on, it is more a level for a learner to get to know behavior verification through mocks and the inner workings of the XHR object itself.

Note that explicit use of mocks and spies is not an element in any of the test cases, but is used as a result of the TDD process in all frameworks that provides them.

7.2 Personal experiences

This section will explain the personal experiences of developing unit tests with TDD in JavaScript.

Getting started proved easy because of the simplicity in the FizzBuzz test case. Some issues were met[1], but these were resolved fairly quickly. Many

[1] These are referred to in the corresponding Result Section of the framework.

errors were made on referencing source code in the in-browser test runners. The file path was spelled wrong or the wrong files were loaded. The test frameworks that had an associated gem for file discovery proved the simplest test runner in my opinion. The JsTestDriver runner also worked, but it had to be restarted to discover new file, and the recapturing of browsers did not always work smoothly.

7.2.1 Difficulties

The difficulties described in this section were the major ones met during the development of the test cases.

Timing functions

A bit into the Alias test case, it was found that the timing function `setTimeout` had not been well enough researched and a naming collision had happened.

This had happened because a string was given as an input to the timing function:

```
1 setTimeout('variable.methodname', milliseconds);
```

The `methodname` was evaluated in the global scope, not in the function scope where the function was initiated. A collision happened as the first variable was named the same in the test's global scope and source code's functional scope, which created a lot of confusion on why the code suddenly worked and then failed when other approaches was tried. When the anonymous inner functions were resolved as an input to the timing functions and the scope was sorted out, the timing and accompanying functions worked as intended.

Sinon integration

Getting the Sinon functions to work, both with time and faking server responses, was difficult due to many issues. Firstly, it was due to wrong usage of the timing functions as described above. Then it was getting to grips with the setup in the different frameworks. Jasmine and JsTestDriver, does not need customization to the Sinon test, but it needs a wrapper to work with the QUnit test cases.

This made the QUnit result from Section 6.2.6 stand out. Sinon automatically inserts fake timers on the `this.clock` variable in the test itself. This has three implications: *1*) call to timing functions in setup or teardown need are not automatically faked *2*) if setup calls a timeout function, the fake timers need to be inserted manually *before* any such call *3*) own fake timers *must* be bound to another variable that `this.clock`. This indicates the function for creating a `tinesFakeClock` could have been replaced with a normal setup on fake timers on another variable, e.g. `this.time`.

The difference in the application of the frameworks functions and the lack of documentation, create some frustration when trying to learn JavaScript with unit testing. In general, dynamic languages are harder to learn, as the source of an error can appear many places. Experience makes your own JavaScript less error-prone and gives more insight in where to look when problems arise.

7.2.2 Experiences with TDD in JavaScript

It is difficult to evaluate own progress and application of a methodology, but some examples can be seen as explanatory for application of TDD in the test cases.

XHR

When learning about the XHR object, a tutorial on W3Schools[87] were used. The tutorial presented this case on the XHR `onreadystatechange`:

```
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    //Some manipulation of the reponse text
    document.getElementById("myDiv").innerHTML=xmlhttp.
        responseText;
    }
  }
```

Listing 7.1: Excerpt from a w3schools tutorial on Ajax.

This was used as a skeleton for receiving objects in the Twitter test case. Later, it was discovered that this code makes two assumptions; the `readyState` attribute needs to be 4 and the `status` needs to be 200. The first is assumption is valid, but the second is not. In the case of the test case running in a browser with enabled same-domain policy, the `status` cannot be accessed at all. These assumptions were used when writing the tests as well, returning an empty response text and a 200 status. The tests worked fine, but the when the code was run in a real environment it failed due to the assumptions made in the tests and code.

Had this been done in a correctly TDD manner, the test would have been written first and only the ready state attribute would have been added to the if-statement, and it would likely have been discovered that the XHR status would be a non-readable attribute. Another possibility would be that it never was discovered, as the code would have worked with the `readyState` attribute alone, and no test would introduce a need to look at the `status` attribute.

This can refer back to Section 2.2.3 and what was posed as one of the risks in TDD, namely "getting the methodology right". In the case shown above the application of the methodology fails as a tutorial is being used to help with a problem. This leads to a breach of the TDD process, the test is not written first, which again gives tests that pass, but code that does not work in a real browser environment. To be able to avoid these assumptions in tests, careful consideration is required by the developer when learning new things and rewriting code from the outside to fit into the product. Other methods, like code review can alleviate this risk.

The problem above arose from an initial run of the Twitter test case in standalone Jasmine. In this framework there is no way to test Ajax, and only the correctness of the content within `onreadystatechange` be tested, not the invocation itself. This entails that it is very difficult to unit test for the XHR status issues.

That this issue has been introduced in the test cases that contain options for Ajax testing is likely a result from transfer of parts of former executions of test cases on to the same test case in other frameworks.

Refactoring

A major part of the TDD process is the Refactoring - step. This step is somehow harder to do right as the IDEs for JavaScript provides less help. Changing a variable name, extracting an interface and other patterns have dedicated functions in Java IDEs, but has to be done manually in JavaScript. As the type checking is dynamic, no warnings are given in case a variable is forgotten, in most cases it will be added as an implicit global variable. This can have consequences, such as a naming collision. QUnit was ahead of the others with its check of introduced global variables, but the other frameworks largely ignored this. This can be due to the fact that neither

of the frameworks tries to be an IDE, but some support could increase a framework's usefulness.

Code kata

An interesting aspect was the application of code katas in TDD. The code kata *FizzBuzz* was used as the first test case to get to know setup, execution and syntax through an easy test case. In a code kata, the developer are encouraged to analyze and improve the solution on each repetition, and this fits very well with the incremental process in TDD. Looking at this in action, the *FizzBuzz* test case changed minimally over the course of the different framework. This is a totally different experience when looking at the Alias and Twitter test cases. They have has gone from utilizing a shared global variables to self-invoking functions and advanced closures. The test cases could definitely be used to learn JavaScript and its application in a test-driven manner. It is also an argument for the size of code katas; the *FizzBuzz* case might have exceeded its usefulness when the syntax is learnt, but the Twitter and Alias cases use JavaScript special features and are more complex, inviting to a wider range of possible solutions.

7.2.3 Effects from TDD in JavaScript

Using TDD has definitely given a lot of experience when learning JavaScript. Development has gone from making it work, to trying to utilize more advanced features and more object-oriented design.

This section will try to answer Research Question 3 on the background of the experience gained executing the test cases in Chapter 6. As discussed in Chapter 3 and Section 7.1 this section will only give a general heeding and a personal answer, as a scientific conclusion would have needed a different research method.

When using TDD, the method encourages taking as small step as necessary to accomplish the task. This acts as a comfort when the problem at hand seems impregnable. It also encourages the developer to change a working code, through refactoring, into something better. This led to the case of self-invoking function and advanced closures in my case. This is one of the greatest advantages when using TDD as an individual.

The issue when trying to learn JavaScript and TDD at the same time occurs when you do not know how to test the problem at hand. Resolving to tutorials, as described in Section 7.2.2, resulted in my case to a short-term accomplishment, but introduced a bug in the form of an assumption, which was only discovered after a long time. So when using other's code to learn JavaScript with TDD; *always* pluck the code apart to see how it functions and introduce only what you really need. This applies for experienced developers as well, as they may copy and paste code to save time.

When finding issues, it can be difficult to see whether they are introduced in the test or in the code, often due to the dynamic nature of JavaScript.

On a group basis, theory suggests that new developers will have an easier time taking over a project with a high test coverage gained from TDD[2].

7.3 Level of testing

When entering this project, it was believed that the TDD to be tested would be on an unit level. Instead, it was found that some of the characteristics of JavaScript are so tightly bound to the browser, resulting in an arguable degree of isolation. It is possible to claim that all form of browser interaction is a form of integration testing[3], as there are events happening in the browser that are beyond control and demands more sophisticated mechanisms in or-

[2] As discussion in Section 2.2

[3] Described in Section 2.1.2 on page 8.

der to achieve isolation. Even libraries as Sinon.JS which provides a fake implementation for the timing mechanisms and Ajax though XHR manipulation, are doing integration testing, as they are dependent on the browser firing the faked events, and it is the browser that controls the execution stack and provides limitations to, e.g. the XHR object.

Many of these mechanisms are in the crossing between unit and integration testing, but this thesis argues that it is convenient to do this as a part of a unit test. This is because the cooperation between JavaScript and the browser are tight, and the resulting script would benefit from integration between the two parts. Many JavaScript applications contain Ajax or timing functions, and testing this on a integration level would be a artificial sectioning of tests. Testing JavaScript and the browser on a unit level can compare to utilizing another framework or utility; the exception is that some of the control is passed to the browser. But there is not way to attain a high test coverage and ensure correct behavior without testing these interaction points between JavaScript and the browser.

The JavaScript tested in the test cases is on a very low level. It would not be a common practice to verify that the send and open calls of an XHR object is performed. Using jQuery, the developer can move one level up as jQuery takes care of the most common cross-browser errors and discrepancies[4], e.g. `jQuery.ajax`, but still do testing on a unit level.

The level of testing, as presented by the framework themselves, is discussed in Section 7.7.

[4]This is called *browser normalization*.

7.4 Reviewing JavaScript characteristics

This section refers back to the characteristics that were discussed in Chapter 4 and discusses these in the light of the result found in Chapter 6. Responsiveness is not discussed as this was not considered a relevant aspect for unit testing[5].

Interaction

Interaction refers to JavaScript's modification of, and access to HTML and CSS elements through the DOM. The test cases have not tested CSS interaction, so this is not taken into account.

An important tool in order not to introduce any cross-browser errors in the tests, is the use of HTML fixtures. HTML fixtures allow a developer to insert HTML code that are removed and reinserted before each test. This leaves the framework responsible for not introducing errors and allows the tests to be isolated instead of running on an existing webpage.

The importance of being able to write HTML as HTML code, not with insertion through JavaScript, was found to be important. Referring to the standalone Jasmine version, where the setup was five lines of code versus JsTestDriver's one line, and JSpec's three. This does not seem too much, but looking at the code, the Jasmine version will increase faster than the other two, due to its usage of JavaScript. Possibilities exist for the developer to modify the framework or add their own fixture node, such as in QUnit, but this decrease the life of the software and its readability, as parts of the code are not documented and are created by one person only.

Combining **Interaction** and **Complex logic**, another feature is emergent. In larger test suites, the HTML might need to be set up on a test basis. In QUnit, the HTML fixtures are added on a *test runner* basis. This can lead

[5]Referring to Section 4.1 on page 47.

to a lot of unnecessary fixture code for each test, and it would be possible that the code affected each other and created dependencies and other side effects.

Cross-browser execution

Testing in a cross-browser environment has a lot of issues around setup, execution and how to view results, which is covered in Chapter 6 and Section 7.5.

A cross-browser error happens when a property or method is specific to one ore more browsers. The web contains a lot of these examples and it is easy to get tangled. Discovering these errors can be difficult, especially if the tests are executed in one browser only, if your tests coverage is low or the tests cover the wrong thing. An example would be a modified DOM test case, as seen in Listing 7.2, where the second line is taken out. When using only the innerText attribute to insert and test for the text content, the tests pass in all browsers. Still it introduces a cross-browser error in Firefox, which is not discovered due to the dynamic creation of variables in JavaScript, as demonstrated in Listing 7.3. This cannot be seen in the browser as the result of the test run is the only content displayed on the site.

```
1 Find HTML element with id = text
2 \\ Check that content of element is 'SomeContent'
3 Update text content of element through innerText
```

Listing 7.2: Pseudocode for test case 4: "DOM" with line 2 commented out.

```
1 //code
2 var fetchNode = function() {
3   var node = document.getElementById('text');
4   node.updateText = function(text){
5     node.innerText = text;
6   }
7 }
```

```
8  //test
9  test('update text content', function() {
10   var node = fetchNode();
11   node.updateText("newText");
12   equal(node.innerText, 'newText');
13 });
```

Listing 7.3: Testing the changed DOM test case in QUnit.

The `innerText` variable will exist for all browsers but Firefox, and in these it will change the DOM element as expected. In Firefox, the variable will be created at runtime in line 5. The assert statement on line 13 will check that the variable exists and contain the expected string, which it will in Firefox as it was just created by the test. But will not change the text of the DOM element, hence the wrong thing is tested.

If the test had done the assert like the following

```
1  equal(node.innerHTML, 'newText');
```

the assertion in line 12 in Listing 7.3 would fail. But given that the cross-browser error is not already known, testing with `innerHTML` would be a very unnatural way to test.

Instead of using alternative references like this, it might be needed to test on a higher level, like FuncUnit, to ensure that the test covers the intended area or use a library, like jQuery, for browser normalization.

A second approach would be encapsulation of all areas that touch the DOM and have these fully tested for cross-browser errors.

Features for HTML fixtures in a framework will provide help in preventing new cross-browser errors and dependencies in tests, as the developer does not need to do a manual setup and teardown of HTML fixtures.

Complex logic

Custom matchers can be a good feature when complex application logic needs to be tested. Imagine a scenario where JavaScript uses person objects in a JSON format between clients. To ensure that an object is correct, a developer can create a matcher `isValidPerson` that asserts a certain set of properties, instead of writing more asserts to check this.

A problem with complex logic, especially if it demands a lot of files, would be use of in-browser test runners needing `script` references. This could quickly lead to missing files, lost tests and a lot of clutter in the process. On the other hand, we have the problems with monitoring a large number of files, as experience with the `jspec` command.

Tip Mitigation of this problem can be through a Ruby gem that does not monitor files for changes, only monitors specific folders to see if new files are present. This means that the developer must run the tests manually[6], but the risk of lost test is significantly reduced. The Jasmine gem works this way, and no problems were met here when the number of files increased.

Asynchronous code

Asynchronous code referred to JavaScript timing mechanisms and ability to perform Ajax calls.

Time

In Section 4.1, three ways of testing time was discussed, waiting a given amount of time, faking the duration of the interval or abstracting away the dependency. Table 5.3 showed which of the framework that contained the

[6] See page 125 for a tip on mitigating this.

given functionality. The Alias test case was developed specifically to investigate the timing feature and from the results in Chapter 6, three points an be emphasized:

- Pausing test is not a viable option, as it breaches both unit testing characteristic and the red/green/refactor - mantra. It also causes a lot of side effect that have a negative impact on TDD, like producing code before the test results have returned.
- Faking time works efficiently, but can be difficult to achieve, as discussed in Section 7.2.
- It is possible to test only the inner content of the timeout function.
- Abstracting away time, as done in Section B.1.1, changes the design of the code.

Another variation of abstraction can be found when looking at the Java language. By creating an interface and replacing this with a stub implementation, this is another way to build fake timer. An example of this in JavaScript can be found in Appendix C. This example, together with Section B.1.1, shows that it is possible to achieve testing of time without support in a framework. Though this encourages a change from the natural design of JavaScript, which is considered to be a negative factor.

Discarding pausing of tests due to the introduced wait and abstracting due to the design it encourages, only two options remain for testing JavaScript timing functions; *1*) a fake timer *2*) calling the content of the timing function's function parameter directly in the test. Both are exemplified in Listing 7.4.

```
1 //in code
2 var bool = false
3 function someFunction() {
4   [....]
```

```
5    bool = true;
6  }
7  setTimeout(function(){
8    someFunction();
9  }, 1000)
10
11 // test code with Alternative 1
12 fakeClock.tick(1000);
13 assert bool == true
14
15 //test code with Alternative 2
16 someFunction.apply(window);
17 assert bool == true
```

Listing 7.4: Examples of testing with fake timers and content of parameter.

Alternative 2 with the `apply` function, evolved from the test cases to add correct scope to the test[7].

The code coverage will decrease when using Alternative 2, because the setTimeout call is never invoked.

In a test with the code coverage module in JsTestDriver the coverage changed from 94.4 % when using Sinon (Alt. 1) to 88.9% without (Alt. 2).

Dissecting further, duplication is seen between the source code and the test with Alternative 2. If the content of the timeout is changed, the test must be changed to reflect this. This is against the rules of TDD[8]. This would not be the case with Alternative 1, as no duplication exists between test and implementation.

It would be possible to change the inner workings of `someFunction`, or encapsulate the changes in a new method, but one line of duplication will still exist.

[7]The apply function calls its object with the given parameter as `this`. `window` is the global scope, the same scope that `setTimeout` is evaluated in.

[8]Described in Section 2.2.1.

Depending on the effects of `someFunction` in Listing 7.4, it may also be difficult to validate the effect. If `someFunction` only has indirect effects, Alternative 1 can validate by spying on `someFunction`. Alternative 2, which must invoke `someFunction` from the test, must find another change or method invocation introduced by `someFunction` to be able to validate the call. Depending on the function at hand, the difficulty can range from easy to untestable.

In total, this leads to the conclusion that a mechanism for faking time is necessary to perform testing of JavaScript's timing functions.

Ajax

With Ajax, there are three ways of testing *1*) fake the XHR object to it is possible to trigger `onreadystatechange` from the test *2*) invoke the content of the `onreadystatechange` method with a modified XHR *3*) pause the test and make real calls to the URL, waiting a given time for the answer.

Alternative 3, pausing the test, is not an applicable solution. The problems around waiting are the same as with timing explained above, and making a real call is against the test isolation demanded from unit testing.

Alternative 1 and 2 both demand that the developer gets to know the inner workings of the XHR object and how it interacts with the browser during an Ajax call.

Alternative 1 can be done in two different ways, either by having the framework taking control and trigger the `onreadystatechange` or manually invoke `xhr.onreadystatechange` from the test. The former can trigger the same-origin policy in the browsers, which can result in hindrances when it comes to cross-browser testing, while the latter can be create a lot of confusion when the XHR object is managed manually.

Alternative 2 has the same issues as invoking the content of a timing function, it creates duplication between test and source code, which is against

the TDD imperatives. It also shares the issues of manually altering the XHR object, which can quickly lead to errors

Duplication of code, problems with manual XHR objects and the impossible wait time leads to the conclusion that a mechanism for faking Ajax calls must also be a part of a unit testing framework for JavaScript. This will help ensure that a unit will function correctly on different responses, and as such providing developers with tools encouraging robust code.

7.5 Reviewing frameworks characteristics

This section discusses the four defining categories of frameworks, as found in Section 4.3 and their fit after the execution and result of the test cases.

Environment

The environment used in all the test cases was the browser, even though several had option for Rhino. This may be different if the system were to be integrated in a CI system or were completely disconnected from the browser. In general though, the browser environment may be the best fit for most developers. On this background, it would be proper to say that the execution environment is the browser, unless project specification opens for or have a need for otherwise.

Execution

The execution of the tests are bound to whether the tests run *in-browser* or *headless*.

An *in-browser* runner has access to the `console.log` function, which is important for debugging. Most browser consoles also have options for interactive exploration of a object, as opposed to a method outputting a

string. This was a disadvantage for JsTestDriver, as it only had a static function outputting to strings to the terminal. This lead to a large number of log strings, which made it harder to keep up with the execution flow.

As a note, JSpec was found not to be functional in Chrome due to its mode of execution. It needed to load the source files from specified directories, which invoked a cross-origin request. This is not favorable, as the framework would need to be run with the cross origin policy disabled on a permanent basis. The problem is mitigated if another browser is used as backup to make sure no unintended cross origin requests are made without the explicit need.

Setup

In the test runners where the source files needed to be referenced in `script` tags, this turned out to be a source of error. This was needed for JSpec, standalone Jasmine[9] and QUnit. Suddenly, tests were running or all failing because a name was wrongly spelled or the originating folder were wrong. With the `jspec` command this was even worse, as the command needed to be rerun for each time the test runner were updated.

YUI Test had runner code and test code in the same HTML file. Here it would have been appropriate with higher separation of concern to avoid cutter in the test and improve readability.

As a default setup and test, a `assert true == true` test as a starting point can be added. This gives a test that should always run and always pass. If the test does not run, it may indicate a wrong setup or syntax error.

Lifecycle

Two things determine the lifecycle of a test run: the implementation of failures, and setup and teardown. The latter is determined by the test library

[9]But not with the Jasmine gem.

and usually tied to its methodology[10]. The former refers to how the framework reacts to failing assertions, either by stopping or continuing the test currently being executed. Listing 7.5 shows a simple example of pseudocode that can reveal a lot about this.

```
1 assert true === true
2 assert true === false
3 assert true === true
4 assert 'thisString' === 'thatString'
```

Listing 7.5: A test to reveal the lifecycle of a framework.

Performing this test in multiple frameworks yields very different results, as shown in Figure 7.1.

Figure 7.1c and 7.1e show that YUI and JsTestDriver stop at the first failing assertion. Figure 7.1b shown how Jasmine shows both failing assertions, but none of the passing ones. Jasmine has an option for showing all passed tests, but none for showing all passed assertions. Figure 7.1a and 7.1d show that QUnit and JSpec present both the failing and passing assertions.

Stopping at the first failing assertion and providing no way of seeing the other assertions, hide potentially important information when debugging a failed test. Hiding passed assertion forces the developer to go back to the code to see the assertions that did not fail, and although this is not as severe as the first case, it is still unnecessary information hiding.

A minor difference is how the runners count the passed tests or assertions. QUnit and JSpec count assertions, JsTestDriver and YUI count tests and Jasmine counts number of tests run and number of failing assertion.

The test runners have much in common with the other runners supporting the same methodology, but this seems not to apply for the lifecycle.

[10]TDD has setup and teardown on a single test basis, while BDD usually has nested setup and teardown.

(a) QUnit

(b) Jasmine

(c) JsTestDriver

(d) JSpec

(e) YUI Test

Figure 7.1: Result of Listing 7.5 implemented and executed in the different test runners.

Running and viewing results

When viewing results, the general execution of all test cases was important, but especially the DOM case. This was because it was set to run in four major browsers; Firefox, Safari, Chrome and Opera. The main problem had to do with the execution characteristic mentioned in Section 4.1, namely *in-browser* and *headless* testing. With headless testing there are an opportunity to capture multiple browsers and run all tests on them simultaneously. The results of the execution will be returned to a common interface that summarizes the results, usually the terminal.

The in-browser runners had issues, as expected, with the refresh needed to rerun the scripts. A manual refresh had to be triggered in all four browsers, requiring a lot of tabbing. This changed quite when it was found that the text writer TextMate[88] had scripts the could be loaded on commands. A script to update all running browser pages was inserted on cmd-R. With two screens, the actual screen space ended up being the limiting factor, as a simple shortcut or command in the chosen IDE made the browser refresh and present results.

Tip Check if your text writer or IDE can load scripts on command. If not, external tools, like XRefresh and browser add-ons, can provide similar functionality[11].

A note on personal preferences was the `jspec` gem, which opened a new tab containing the script each time a save was done. I save a lot more often than I would like to test. I like to write the code I expect to work and then press run. Combined with the annoying tab mechanism, the `jspec` gem was dropped in favor of TextMate's customized cmd-R. In general it was the gems that kept track of the files to include, that was the most useful ones,

[11] This has not been testing during this thesis.

alleviating the need for editing `script` tags, rather than gems for refreshing on save.

The design of the result viewer was important, especially when trying to keep in a red/green/refactor-rhythm. The runner in YUI Test made it difficult to see the actual result of the running tests, or if the newly introduced test broke existing functionality. This is very against a TDD process, as any failures in existing functionality should have a high priority. With the DOM test case, this became even more significant.

The design of the result viewer also had some unexpected results, apart from the bad design of the YUI Test console. The QUnit test runner had an easy and elegant way to present the passed and failed assertions with a diff viewer and presentation of messages. I started writing more messages on each assert, especially when loops were tested.

Using the code in Listing 7.5 can also help visualize the difference in the result viewer, as it can show how the different assertions and assert messages are visualized on both passing and failing tests.

Syntax

In general, the differences between BDD and TDD frameworks were less that expected. It was mainly the syntax differences and the presentation of the result that changed. When reviewing back to the documentation and research from Chapter 5, the BDD libraries had nested describes, setup and teardown, but otherwise were quite similar to the TDD frameworks.

Some of the explanation may come when considering the level of testing. On a unit level, the tests are written by the programmer and provide documentation for other programmers, as well as validation of the small parts, often on a method level, in the system. Customers usually do not operate on this level, as the DSL here is the programmer's domain. When the system

is put together as a whole, it usually requires adaption to fit the business problem, and this is where the BDD DSL comes into play. On this level the customer can explain in its domain language how the system should act, and easier communicate with the developers through a natural language, close to the BDD syntax.

Test library

The library functions were mapped in Section 5.12 and a great deal of variation was seen in the supported functionality.

Dedicated test libraries introduce spies and stubs that help return special values or to check if a method has been executed. Unlike mocks, stubs and spies can be created by the developer, making a test a more standalone and easier to read for someone who does not know the framework. Whether you need the stubs and spies would depend on personal preferences and the size of the application to test, and this needs to be kept in mind when choosing a framework for testing.

In general, libraries with general unit testing functionality can be seen to improve readability to those fluent in its terminology. But due to the differences in invocation of the functions, documentation is still vital. Of the validation features, stubs, spies and mock does not all have to be included, but combination tools to provide state and behavior validation is necessary. Stubs are also needed to preform a range of unit testing patterns. This means that a combination of stub and spies or stub and mocks need to be present. Providing all three give developers a wider range of tools, and is certainly only positive for the framework.

The differences in methods and how the libraries were built could be seen influencing the design of tests and source code. Sinon had a `server.respond` method that made the solution of the Twitter test case change due to this

invocation of the return of an asynchronous event. In JSpec, this was different, as it provided a `mock().andRespond()` method that returned the object on response at once.

Methods needed to test asynchronous JavaScript and HTML fixtures have been discussed in Section 7.4[12] and found to be necessary for a JavaScript unit testing framework.

Naming confusion

With regards to naming, especially around mocking, there is some confusion. Jasmine has implemented `jasmine.spy` as a common denominator, described as such:

> Jasmine Spies are test doubles that can act as stubs, spies, fakes or when used in an expectation, mocks.

Firstly, this leads to confusion around the original name, spies. Even in their own description, *test double*, is used as a denominator, and a suggestion would be to stick with this, if they insist on mixing the stub and spy implementation.

Furthermore, it is stated that it can be used as mocks together with expectations. Their example are as following[89]

```
1 // foo.not(val) will return val
2 spyOn(foo, 'not').andCallFake(
3   function(value) {return value;}
4 );
5
6 // mock example
7 foo.not(7 == 7);
8 expect(foo.not).toHaveBeenCalled();
9 expect(foo.not).toHaveBeenCalledWith(true);
```

Listing 7.6: Jasmine mocks.

[12] Page 114, 117 and 120.

The example in Listing 7.6 are quite similar to Sinon's spy example seen in Listing 7.7[60].

```
var spy = sinon.spy();

PubSub.subscribe("message", spy);
PubSub.publishSync("message", undefined);

assertTrue(spy.called);
```

Listing 7.7: Sinon spy.

Sinon spies also have an advanced API to assert if a method was called, number of calls and arguments, the same thing Jasmine does with its mocks.

Referring to Meszaros[5], a mock behaves as following

> When called during SUT[13] execution, the Mock Object compares the actual arguments received with the expected arguments using Equality Assertions (see Assertion Method on page 362) and fails the test if they don't match. The test need not make any assertions at all!

This leads to the conclusion that the Jasmine mocks are not mocks at all, they are spies! Mocks are objects that verifies a run by comparing pre-stated expectations to the actual run, usually triggered by a `verify` method. A correct mock implementation would be as the one in Listing 7.8. The programmer provides no assertions, but the mock has stated expectations, which it compares to the actual run. This validation is usually triggered by call to a dorm of `mock.verify`.

[13] System under test (SUT).

```
var myAPI = { method: function () {} };

var mock = sinon.mock(myAPI);
mock.expects("method").once();

PubSub.subscribe("message", myAPI.method);
PubSub.publishSync("message", undefined);

mock.verify();
```

Listing 7.8: Sinon mocks[60].

Issues with add-on libraries

It was a general trend that adding Sinon made the development process harder. It might be that the background for this is small, as there was only one other framework with time and Ajax mocking built in, namely JSpec. It must also be mentioned, that it may possibly be a drawback that the same test cases were executed across different runners with the same add-on library. This may have given ground for believing that Sinon worked one way, and that it would be the same with another test framework. This can also be seen as a drawback for Sinon; what works in one framework, does not necessarily work in another. The lack of documentation is a source of frustration as well, and an improvement here would likely make it easier to integrate the library.

With Sinon, the issue manifested with the QUnit/Sinon combination. Section 6.2.6 and 7.2.1 showed how the `clock` variable was automatically faked with QUnit and Sinon, but only in the test scope itself, not in the setup.

7.5.1 Redefining framework characteristics

Based on the experience with the test cases, a modified version of important framework characteristics differentiator is:

- Execution
  - Setup
  - Environment
  - Lifecycle
  - Result viewer
- Test library
- Documentation

Syntax is taken out as a characteristic, as it did not contribute on a unit level[14].

Environment is made less important, running Rhino was not needed in the test cases, but it is still important to be conscious about alternatives to the browser. Rhino and other environments supporting CI only grow more important as the JavaScript code reaches an enterprise level.

Setup, lifecycle and result viewer is taken into account as areas where the frameworks differ a lot from each other. The test library should contain the recommended set of features presented in Section 7.6.

Documentation is added as an important point to the list of characteristic. At many points during the work with this thesis, a lack of documentation made the process a lot harder that necessary.

[14] As discussed in Section 7.5 on page 126.

7.6 Recommended test library features for unit frameworks

The recommendations have emerged from the experience with the test cases and the previous sections of the discussion. The set of requirements will depend on the characteristics of the project to test, as well as personal preferences.

The general features of unit testing frameworks, setup, teardown and test organization are needed. A combination of stubs and spies or stubs and mocks are also needed to perform general unit testing patters. Event though some of these can be created by hand, the readability, lifespan and correctness of the code increases when using a publicly available library. It also allows developers to use patterns developed by unit testing experts.

The JavaScript unit testing framework should also contain the following mechanisms:

- A method for faking Ajax: As discussed in Section 7.2.2 page 117, code coverage decrease if this option is not present and mistakes are easily made invoking the `onreadystatechange` function manually.

- Options for faking time: Also due to discussions from Section 7.2.2 page 117; code coverage decrease if this option is not present and it does not force unnatural design.

- Option for adding HTML fixtures: Because setup of HTML in JavaScript is error-prone as discussed in Section 7.2.2, page 114.

This entails that QUnit with Sinon, JsTestDriver with Sinon and JSpec are relevant options. Depending on whether you would need to add fixtures or possibly add few, Jasmine with Sinon can also be considered. JSpec are

no longer supported, so depending on the lifetime of the project to test, this might need to be reconsidered.

Jasmine recently launched a `jasmine.Clock` class for faking time, and other actively maintained framework might also see how the other implementations are changing, and adapt accordingly.

As seen, the thesis arguments for a set of requirements for JavaScript unit frameworks, but on the other hand, e.g. HTML fixtures can be created by the programmer and the same with simple stubs and spies. These recommendations must always, due to the extensibility of JavaScript, be compared against developer experience and project requirements.

Having the test framework features described in this section is important, as well as knowing the framework through research on the factors from Section 7.5.1. This is important because changing a framework halfway through a project can be difficult. If the existing functionality can be found in the new framework, there is a possibility of writing a custom adapter to map the old methods onto the new ones. But if currently utilized functionality cannot be matched in the new framework, the tests may need to be rewritten line by line.

7.6.1 A Discovery test case

The experience with the test cases as a way to get to know a framework, were positive. The code in Listing 7.9 can help to see how configuration, running, test cycle and presentation are in a framework.

The first test is fairly simple and allows the developer to get to know the test setup, syntax and assertion mechanism. The second and third test case uses fake timers both in test and setup to see how this integration is done. If this had been the first test case instead of the FizzBuzz kata, I believe that I would have saved a couple of days of work.

For completion, an Ajax call should have been added as well, but this demands more code, which is not convenient for a this proposal. The test case seeks to give a lot of information in a short time, not a complete review. If you need to utilize fake Ajax functionality, this test case will only provide a good starting point.

```
//1.st test

//test
assert true == true
assert false == false
assert true == true
assert 'thisString == 'thatString'

//2.nd test

//test
setup false timers
fake 'numberOfSec' amount of time
assert bool == false

//3.rd test
//in setup:
setup false timers
timeout(numberOfSec);

//in test
fake 'numberOfSec' amount of time
assert bool == false

//in source file for 2.nd and 3.rd test
var bool = true;
function timeoutThenChangeBool(sec) {
  setTimeout(function(){
    bool = false;
  }, sec);
}
```

Listing 7.9: A test case to get to know the features of a new framework.

7.7 Testing terminology for JavaScript

When looking for JavaScript unit testing frameworks, little was found. Self-descriptions were often "testing framework" or BDD framework. Some of the frameworks mentions their roots in unit testing, but other JavaScript testing framework seem generally unaware of the terminology connected to unit testing. Apart from FuncUnit, which provides higher-level testing through UI[15], all the frameworks provide mechanisms for testing on unit level only[16].

The non-BDD frameworks described do not force TDD as a method and could very well be used for regression testing and test-last methods. This is another matter for the BDD style frameworks. They emphasize on the BDD features and does not present themselves as general unit testing framework. Referring to the background covered on BDD in Section 2.2.4, this indicates a misunderstanding. BDD is a methodology and not a framework. As mentioned earlier, this thesis regards BDD as less important on a unit level as this is the domain of the programmer, not necessarily the customer. Unit tests act as documentation for other programming peers, and this does not require a natural language, rather it may benefit from a precise, technical language. The BDD style frameworks *can* be used as a test-last and regression testing suite and even with a TDD methodology.

This confusion emphasizes the original motivation of the thesis; that traditional development and testing methods have not yet caught up with JavaScript, and that traditional unit testing have been encompassed by newer methodologies as TDD and BDD. Applicants today should strive to bring the terminology down to a unit testing level. This will help new users to reap benefits of existing knowledge through the use of general patterns, and expert users can use this vocabulary to discuss and pass on knowledge. This will

[15]Rather interaction through an API as the xUnit frameworks.

[16]This will depend on the following the level classification in Section 7.3 and seeing event testing as disconnected to the core frameworks.

also help when discussion testing issues specific to JavaScript, and potentially add a new patterns to the existing set.

7.8 Summary

This section has discussed the results from Chapter 4 in light of the results uncovered in Chapter 6. Important changes have been discussed relating to how unit testing must be performed and what the different JavaScript characteristics demands of the framework. The differentiators in a framework have also been revised and are presented in Section 7.5.1. The most important changes were adding documentation, lowered weight on execution environment and removing syntax. Minor additions have also be made on subsections on both test library and execution.

A recommended set of test library features has been identified in Section 7.6 to answer Research Question 2. Here it is important to remember that project specific needs must be taken into account and these can change the requirements to find the framework with the best fit.

In Section 7.6, the DISCOVERY test case was created to provide developers with a simple test to discover general settings, lifecycle, execution and presentation as well as testing of time in both setup and test. This case can save developers from initial issues when using a new framework, as the test case is easy and allows focus to be on the inner workings of the new framework.

Research Question 3 have been discussed, but not answered in Section 7.2.3. This is due to the research method chosen in Chapter 3. The method used will only reflect personal views, and cannot scientifically answer the question, but can provide an incentive for further research on the topic.

Chapter 8

Conclusion

During the work with this thesis, Sinon.Js released a new version, Jasmine released a module for faking time and BDD framework Cucumber[1] was ported to JavaScript. This shows that the area is of interest to the industry and developers, and it is likely that the topic will gain even more publicity in the time to come.

The project has contributed to the body of knowledge within unit testing with TDD in JavaScript and this chapter will suggest how the thesis can be elaborated in the future. As discussed in Chapter 3, the goal of this thesis was to act as a demonstration of new aspects within the domain. This is elaborated in Section 8.1.

[1]Originally a Ruby framework for automation of plain text tests.

Research Question 1: ***What available frameworks exist for unit testing TDD in JavaScript?*** Eight frameworks were found and two test libraries. Five frameworks and all the libraries are still actively maintained. The most important with regards to functionality, community and support were, in alphabetical order:

- Jasmine
- JSpec[2]
- JsTestDriver
- QUnit
- Sinon.JS[3]
- YUI Test

Some BDD frameworks are present on this list as well, due to the fact that BDD has TDD as its originating methodology[13]. A description of the remaining frameworks can be found in Chapter 5.

Research Question 2: ***Which test features are recommended for a JavaScript unit testing framework?*** To discover these, a set of test cases was developed to cover testing issues in JavaScript and get to know the different aspects of a framework. Information and research was done in Chapter 4 and the test cases presented and justified in Chapter 6.

Executing the tests in Chapter 6 gave new insight, and based on the discussion in Chapter 7 the following recommendations for necessary library features in JavaScript unit testing frameworks:

[2] No longer supported.
[3] Test library.

- General unit testing features
  - Test organization; setup, teardown and test
  - Stubs and spies, or stubs and mocks
- JavaScript specific test features
  - Faking Ajax
  - Faking time
  - HTML-fixtures

This entails that the best suited frameworks are JsTestDriver with Sinon.JS and QUnit with Sinon.JS. JSpec also fulfills the recommendations, but is no longer supported, and Jasmine fulfills all recommendations, apart from HTML fixtures. When choosing a framework, it is important to consider unique factors of the project as well.

Research Question 3: ***What effect does TDD have on JavaScript development?*** Personal experiences with TDD in JavaScript testing answered this research question. It was an inherent limitation of the research method, as resources and other constraints did not allow a more scientific approach to this question. Personally, TDD allowed progress to be achieved in steps that felt comfortable and the structured approach helped provide a starting point when a task seemed to big or too difficult. It was also experienced what happens when TDD slips, as an unintended assumption came through when using a tutorial. Other effects were increased confidence and a support for experimentation with more advanced features, not only encouraging to make it work.

From earlier research, TDD has been proven to decrease mean time between failures and time spent for quality assurance. Experts argue that is

also lowers the bug mean time to fix (MTTF). Researchers have not found concrete evidence for higher (nor lower) productivity and lower coupling and cohesion. Long time effects like higher maintainability and code reuse has not been found researched.

Even through research does not fully support the benefits of TDD, software experts like Robert Martin and Kent Beck vouch for methodology[3, 12] and researches found that developers who learned TDD stuck to the practice after the TDD project was at an end[8].

Other contributions During the work with this thesis, multiple contributions have been made apart from the above mentioned. This is due to the lack of previous research and industrial application of the combined methods and technology. It is believed that the work done in this thesis is in front on this area. When regarding the interest and contacts that have been met during the course of this thesis, personal experiences vouch for this.

The terminology on the existing frameworks has discussed and it have been suggested that these should move closer to the terms found among xUnit frameworks. This will allow JavaScript unit testing to learn from the existing knowledge and pattern library within unit testing, as well as helping beginners and providing a consistent terminology for experts

The differentiating factors in the frameworks have been discussed and revised with the experience gained from executing the test cases. The differentiators suggested are:

- Execution
  - Setup
  - Environment
  - Lifecycle
  - Result viewer

- Test library

- Documentation

As a response to the differentiators in the framework, the DISCOVERY test case was proposed in Listing 7.9. This allows developers to explore the execution factors in a quick and consistent way. It also has two tests on JavaScript timing mechanisms, checking the ability to fake timing methods initiated by both test and setup. As an endnote, it is important to stress the fit between the project to be tested and the framework, and these considerations can give way to other priorities.

This thesis has resulted in a novel and practical approach, through the DISCOVERY test case, to quickly gain an understanding of a new framework. When adding the recommended library features and differentiator is a JavaScript unit testing framework, a more complete view can be achieved as developers know what to look for. Combined with knowledge of his or her own project, it is believed that the contents of this thesis can help developers make an informed decision when choosing a unit testing framework.

Further contributions are:

- As new frameworks emerge, the four test cases can be used to assessing these, making comparisons easier and more consistent.

- Hints for smoother development have been posed, a automatic browser refresh tools was introduced and the use of Ruby gems to relive the risk of loosing tests with an in-browser test runner.

- For beginners in JavaScript, hints on how to start can be used, and lessons can be learned from the mistakes made with TDD in the process with these thesis.

8.1 Further work

Referring to the research method described in Chapter 3, the thesis will aims to act as a incentive to further studies and background for industrial application. Below are suggestions that were out of scope for this thesis, but could be interesting for further research:

- Performing the four test cases with different groups, e.g. with and without TDD.
- Research on the effects of TDD in a dynamic versus static language. Some patterns, like interfaces, are not applicable in dynamic languages. How does this affect the emerging design?
- Quantitative analysis of JavaScript unit testing, e.g. code coverage.
- Qualitative studies interviewing a team using JavaScript unit testing with or without TDD.
- Unit testing with browser normalization through jQuery: what difference does it make?
- Acceptance testing in JavaScript.
- Component or system level testing of Javascript, e.g. with FuncUnit.
- How to use TDD for learning JavaScript?
- Compare the existing framework to the xUnit family to investigate the terminology confusion deeper and see the degree of alignment possible.
- Empirical research on the effect of TDD in JavaScript for a test case or set of tasks.

- Research on patterns specific for JavaScript; can existing ones be adapted or does it demand an expanded set of patterns?
- Performing the test cases in Rhino with env.js and explore the differences.

As this section illustrates, the area still has a wide rage of topics that can interest researchers and practitioners in a long time to come.

Appendices

Appendix A

JavaScript support functions

```
1  Object.method("superior", function(name){
2    var that = this,
3        method = that[name];
4    return function() {
5      function.apply(that, arguments);
6    };
7  });
8
9  Function.prototype.method = function(name, func){
10   this.protype[name] = func;
11   return this;
12 }
```

Listing A.1: An implementation of super in JavaScript by Douglas Crockford[90].

Appendix B

Source code

An extract of the source code for the test cases. The remaining is found in the attached .zip file.

B.1 Jasmine timing

```
var SIXTY_SECONDS = 60 *1000;

describe("startGame", function() {
  it("should start with 0 points", function() {
    var game = startAlias();
    expect(game.points).toEqual(0);
  });
});
describe("ALIAS round", function() {
  var g;
   beforeEach(function() {
      g = startAlias();
      g.newWord = function () {};
    });

```

```
16   it("should have 1 point if one 1 word is correct", function
        () {
17     var game = startAlias();
18     game.newWord = function () {};
19       runs(function(){
20       game.newWord();
21       game.ok();
22   });
23     waits(SIXTY_SECONDS);
24     runs(function(){
25       game.ok();
26       expect(game.points).toEqual(1);
27       });
28   });
29
30   it("should have -1 point if no words are correct", function
        () {
31     runs(function(){
32       g.newWord();
33       g.pass();
34   });
35     waits(SIXTY_SECONDS);
36     runs(function(){
37       g.pass();
38       expect(g.points).toEqual(-1);
39       });
40   });
41
42   it("should have 1 point after a a round of 6 ok and 5 pass"
        , function() {
43     var alias = startAlias();
44     alias.newWord = function () {};
45     runs(function(){
46       round(alias, 6, 5);
47   });
```

```
48      waits(SIXTY_SECONDS);
49      runs(function(){
50        round(alias, 6, 5);
51        expect(alias.points).toEqual(1);
52        });
53    });
54  });
55
56  var round = function (game, accepts, passes){
57    for (var i=0; i < accepts; i++) {
58      game.newWord();
59      game.ok();
60      };
61    for (var i=0; i < passes; i++) {
62      game.newWord();
63      game.pass();
64      };
65  };
```

Listing B.1: Tests for Alias in Jasmine.

```
1   function startAlias() {
2     var SIXTY_SECONDS = 60 *1000;
3     var game = {};
4     game.points = 0;
5     game.inPlay = true;
6
7     game.ok = function(){
8       if (game.inPlay) {
9         game.points += 1;
10      };
11    };
12    game.pass = function(){
13      if (game.inPlay) {
14        game.points -= 1;
15      };
```

```
16    };
17
18    setTimeout(function () {game.inPlay = false;},
        SIXTY_SECONDS);
19
20    return game;
21  };
```

Listing B.2: Source code for Alias in Jasmine.

B.1.1 Abstracting away time

In the following code session, the timing function is abstracted away. The test that is skipped *'should have 1 point if one 1 word is correct: async'* is to give confidence, but the test *'should have 1 points, 6 ok, 5 pass'* gives the same confidence when testing needs to be fast.

```
1   describe("startGame", function() {
2     it("should start with 0 points", function() {
3       var alias = startAlias();
4       expect(alias.points).toEqual(0);
5     });
6   });
7
8   describe("ALIAS round timing", function() {
9       var alias;
10      var newWordToRestore;
11      beforeEach(function(){
12        alias = startAlias();
13        alias.newRound();
14        newWordToRestore = alias.newWord;
15        alias.newWord = function (){return "test"};
16        });
17      afterEach(function(){
18        alias.newWord = newWordToRestore;
19      });
```

```
20    it("should give 1 points if one word is correct", function
         () {
21      alias.ok();
22      expect(alias.points).toEqual(1);
23    });
24    it("should give no points if time is up", function() {
25      alias.ok();
26      expect(alias.points).toEqual(1);
27
28      alias.timeleft = 0;
29      alias.ok();
30      expect(alias.points).toEqual(1);
31    });
32    xit("should have 1 point if one 1 word is correct: async",
         function() {
33      runs(function(){
34        alias.ok();
35    });
36      waits(60*1000);
37      waits(500);
38      runs(function(){
39        alias.ok();
40        expect(alias.points).toEqual(1);
41        });
42    });
43
44    it("should have 1 points, 6 ok, 5 pass", function() {
45    round(alias, 6, 5);
46    expect(alias.points).toEqual(1);
47    alias.timeleft = 0;
48    round(alias, 6, 5);
49    expect(alias.points).toEqual(1);
50    });
51  });
52
```

```
53 function round(game, okays, passes){
54   for(var i = 0; i < okays; i++){
55     game.ok();
56   };
57   for(var i = 0; i < passes; i++){
58     game.pass();
59   };
60 };
```

Listing B.3: Tests for Alias with time abstracted away.

```
1  function startAlias() {
2    var game = {};
3    game.points = 0;
4    game.timeleft = 60;
5
6    game.newRound = function (){
7      game.timeleft = 60;
8      setTimeout(game.updateTime, 1000)
9    };
10
11   game.updateTime = function(){
12     if(game.timeleft > 0){
13       game.timeleft -= 1;
14       setTimeout(game.updateTime, 1000);
15     };
16   };
17
18   game.ok = function (){
19     if(game.timeleft > 0){
20     game.points += 1;}
21   };
22   game.pass = function(){
23     if(game.timeleft > 0){
24       game.points -= 1;
25     }
```

```
26   }
27
28   return game;
29 }
```

Listing B.4: Source code for Alias with time abstracted away.

Appendix C

Encapsulating timeouts

This listing shows the pseudocode on how you can create an encapsulated time block able to fake progression of time.

Note that it demands the source code to encapsulate the call to setTimeout in a method that can be stubbed by the test.

```
//In source

[...]
startGame(){
  [...]
  wait(game.endGame, SIXTY_SECONDS);
};

function wait(fun, ms){
  setTimeout(fun, ms);
};

//In test

//Setup
timer = {
  var now = 0;
```

```
18   var queue = [];
19   this.enqueue = function(a){
20     queue.add(a)
21   };
22   this.tick = function(ms) {
23     now = now + ms;
24     //sort queue in ascending order for any s > now then
         remove from array and
25       f.apply(window);
26   };
27 };
28 wait = function(fun, ms){
29   this.enqueue({f : fun, s :  ms});
30 };
```

Listing C.1: Encapsulating time.

Bibliography

[1] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, "What Do We Know about Test-Driven Development?," *IEEE SOFTWARE*, vol. 28, pp. 16–19, 2010.

[2] K. Beck, "Aim, fire [test-first coding]," *IEEE SOFTWARE*, vol. 18, p. 87, 2001.

[3] K. Beck, *Test-Driven Development By Example.* Addison-Wesley, 2003.

[4] R. Osherove, *The Art of Unit Testing: with Examples in .NET.* Manning Publications, 2009.

[5] G. Meszaros, *xUnit Test Patterns : Refactoring Test Code.* Addison-Wesley, 2007.

[6] Selenium, "Platforms Supported by Selenium." `http://seleniumhq.org/about/platforms.html#programming-languages`, 2011. [cited 3 May 2011].

[7] K. Beck, *Test-Driven Development By Example.* Addison-Wesley, 2003.

[8] D. S. Janzen and H. Saiedian, "Does Test-Driven Development Really Improve Software Design Quality?," *IEEE SOFTWARE*, vol. 25, pp. 77–84, 2008.

[9] L. Crispin, "Driving Software Quality: How Test-Driven Development Impacts Software Quality," *IEEE SOFTWARE*, vol. 23, pp. 70–71, 2006.

[10] D. S. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *IEEE SOFTWARE*, vol. 38, pp. 43–50, 2005.

[11] M. F. Aniche and M. A. Gerosa, “Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers,” in *Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010.

[12] R. C. Martin, *The Clean Coder : A Code of Conduct for Professional Programmers*. Prentice Hall, 2011.

[13] D. North, “Introducing BDD.” `http://dannorth.net/introducing-bdd/`, Mar. 2007. [cited 13 May 2011].

[14] D. North, “BddWiki : Behavior-Driven Development.” `http://behaviour-driven.org/`, Jan. 2009. [updated 2 Jan. 2009; cited 14 May 2011].

[15] cucumber / cucumber, “Cucumber.” `https://github.com/cucumber/cucumber/wiki`, May 2011. [updated 13 May 2011; cited 14 May 2011].

[16] D. Crockford, “The World’s Most Misunderstood Programming Language Has Become the World’s Most Popular Programming Language.” `http://javascript.crockford.com/popular.html`, Mar. 2008. [updated 3 Mar. 2008; cited 4 Jun. 2011].

[17] D. Crockford, “JavaScript: The World’s Most Misunderstood Programming Language.” `http://www.crockford.com/javascript/javascript.html`, 2001. [cited 8 Apr. 2011].

[18] Mozilla Foundation, “About JavaScript.” `https://developer.mozilla.org/en/About_JavaScript`, Oct. 2010. [updated 22 Oct. 2010; cited 21 Mar. 2011].

[19] R. Allen, “Self - the power of simplicity.” `http://selflanguage.org/`. [updated 8 Dec. 2010; cited 4 Jun. 2011].

[20] D. Crockford, “Crockford on JavaScript – Volume 1: The Early Years.” `http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-1`, Jan. 2010. [updated 26 Jan. 2010, cited 21 Mar. 2011].

[21] D. Crockford, “Crockford on JavaScript – Chapter 2: And Then There Was JavaScript.” http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-2, Feb. 2010. [updated 7 Feb. 2010; cited 21 Mar. 2011].

[22] Netscape Press Release, “INDUSTRY LEADERS TO ADVANCE STANDARDIZATION OF NETSCAPE’S JAVASCRIPT AT STANDARDS BODY MEETING : NETSCAPE TO POST JAVASCRIPT SPECIFICATION AND LICENSING INFORMATION ON INTERNET SITE.” http://web.archive.org/web/19981203070212/http://cgi.netscape.com/newsref/pr/newsrelease289.html. [updated 15 Nov. 1996; cited 4 Jun. 2011].

[23] United States Patent and Trademark Office, “Trademark Electronic Search System (TESS).” http://tess2.uspto.gov/, Dec. 1995. A search was done on the trademark ’JavaScript’ [cited 21 Mar. 2011].

[24] Oracle Press Release, “Oracle Completes Acquisition of Sun.” http://www.oracle.com/us/corporate/press/044428, Jan. 2010. [updated 27 Jan. 2010; cited 4 Jun. 2011].

[25] D. Crockford, “Crockford on JavaScript – Episode IV: The Metamorphosis of Ajax.” http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-4, Mar. 2010. [updated 3 Mar 2010; cited 21 Mar. 2011].

[26] J. J. Garrett, “Ajax: A New Approach to Web Applications.” http://www.adaptivepath.com/ideas/essays/archives/000385.php, Feb. 2005. [updated 13 Mar. 2005; cited 21 Mar. 2011].

[27] D. Crockford, *JavaScript : The Good Parts*. OÂťReilly and Yahoo! Press, 2008.

[28] IEEE, “IEEE Xplore Digital Library.” http://ieeexplore.ieee.org/, 2011.

[29] Springer, “SpringerLink.” http://springerlink.metapress.com/, 2011.

[30] ACM, “ACM Digital Library.” http://springerlink.metapress.com/, 2011.

[31] G. Benoit, "Systems Analysis." http://web.simmons.edu/~benoit/LIS486/SystemsAnalysis.html. [cited 10 Jun. 2011].

[32] Wikipedia, "Systems Analysis." http://en.wikipedia.org/wiki/Systems_analysis, Mar. 2002. [updated 26 May 2011; cited 10 Jun. 2011].

[33] J. Lu, "Introduction to Computer Science." http://www.jiahenglu.net/course/researchmethod/slides/lec9.pdf. [cited 21 Jun. 2011].

[34] G. DODIG-CRNKOVIC, "Scientific Methods in Computer Science." http://www.mrtc.mdh.se/publications/0446.pdf. [cited 21 Jun. 2011].

[35] S. Stefanov, *Object-Oriented JavaScript.* Packt Publishing, 2008.

[36] C. Johansen, *Test-Driven JavaScript Development.* Addison-Wesley Professional, Sept. 2010.

[37] E. Schurman and J. Brutlag, "The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search." http://velocityconf.com/velocity2009/public/schedule/detail/8523, June 2009. [cited 15 Jun. 2011].

[38] C. Johansen, *Test-Driven JavaScript Development*, pp. 60–69. Addison-Wesley Professional, Sept. 2010.

[39] joyent, "joyent / node." https://github.com/joyent/node/wiki, 2011. [updated 5 Apr. 2011; cited 11 Jun. 2011].

[40] Mozilla Foundation, "FireBug : Web Development Evolved." http://getfirebug.com/, 2011. [cited 26 Apr. 2011].

[41] Mozilla Foundation, "Rhino : JavaScript for Java." http://www.mozilla.org/rhino/. [cited 14 May 2011].

[42] J. Resig, "Envjs." http://www.envjs.com/. [cited 14 May 2011].

[43] C. Johansen, *Test-Driven JavaScript Development*, pp. 5–14. Addison-Wesley Professional, Sept. 2010.

[44] D. Chelimsky, "rspec-1.3.1 | rspec-rails-1.3.3." http://rspec.info/documentation/, 2011. [cited 26 Apr. 2011].

[45] C. Johansen, *Test-Driven JavaScript Development*, pp. 257–263. Addison-Wesley Professional, Sept. 2010.

[46] Pivotal Labs, "Jasmine : BDD for JavaScript." http://pivotal.github.com/jasmine/, 2011. [cited 26 Apr. 2011].

[47] M. Hevery and J. Lenfant-Engelmann, "Yet Another JavaScript Testing Framework." http://googletesting.blogspot.com/2009/05/yet-another-javascript-testing.html, May 2009. [updated 22 May 2009; cited 21 Mar. 2011].

[48] M. Hevery, "Google Group on JsTestDriver." http://code.google.com/p/js-test-driver/wiki/AsyncTestCase, Jan. 2011. [updated 14 Jan. 2011; cited 3 May 2011].

[49] M. Hevery, "Google Group on JsTestDriver." http://code.google.com/p/js-test-driver/, Apr. 2011. [updated 29 Apr. 2011; cited 2 May 2011].

[50] Pivotal Labs, "JsUnit : Introduction." http://www.jsunit.net/, 2001. [cited 26 Apr. 2011].

[51] Pivotal Labs, "pivotal/jsunit." https://github.com/pivotal/jsunit, 2010. [updated 18 Feb. 2010, cited 26 Apr. 2011].

[52] C. McMahon, "caolan / nodeunit." https://github.com/caolan/nodeunit, 2011. [updated 17 Mar. 2011; cited 2 May 2011].

[53] K. Kawaguchi, "Meet Hudson : Features." http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson, 2011. [updated 15 Mar. 2011; cited 3 May 2011].

[54] The jQuery Project, "QUnit." http://docs.jquery.com/QUnit, 2010. [cited 2 May 2011].

[55] Jupiter IT, "FuncUnit." http://javascriptmvc.com/docs/FuncUnit.html#&who=FuncUnit, 2011. [cited 3 May 2011].

[56] Jupiter IT, "JavaScriptMVC." http://javascriptmvc.com/index.html, 2011. [cited 1 Jun. 2011].

[57] M. Monteleone, "mmonteleone / pavlov." https://github.com/mmonteleone/pavlov, 2011. [updated 29 Mar. 2011; cited 2 May 2011].

[58] N. Sobo, "screw-unit / README.txt," 2009. [updated 16 Feb 2009; cited 2 May 2011].

[59] N. Sobo, "nathansobo / screw-unit ." https://github.com/nathansobo/screw-unit, 2010. [updated 29 Dec. 2010; cited 2 May 2011].

[60] C. Johansen, "Sinon.JS." http://sinonjs.org/, 2010. [cited 26 Apr. 2011].

[61] jquery, "TestSwarm." https://github.com/jquery/testswarm/wiki, Sept. 2010. [updated 2 Apr. 2011; cited 5 Apr. 2011].

[62] Yahoo! Inc, "Yahoo! Developer Network : YUI Library." http://developer.yahoo.com/yui/, 2011. [cited 25 Apr. 2011].

[63] Yahoo! Inc, "Yahoo! Developer Network : YUI3: Event." http://developer.yahoo.com/yui/3/event/, 2011. [cited 25 Apr. 2011].

[64] Yahoo! Inc, "Yahoo! Developer Network : YUI 3: Test." http://developer.yahoo.com/yui/3/test/, 2011. [cited 25 Apr. 2011].

[65] J. DeWind, "Jsmock." http://jsmock.sourceforge.net/. [updated 20 May 2007; cited 1 Jun. 2011].

[66] K.-E. Rønsen, "jack." https://github.com/keronsen/jack#readme, 2009. [updated 10 Nov. 2009; cited 3 May 2011].

[67] A. Kang, "JJSpec." http://jania.pe.kr/aw/moin.cgi/JSSpec/ReleaseNote?highlight=%28%28JSSpec%29%29, 2008. [updated 23 Sep. 2008; cited 3 May 2011].

[68] TheFrontside.net, "Crosscheck." http://sourceforge.net/projects/crosscheck/, 2009. [updated 17 Jul. 2009; cited 3 May 2011].

[69] A. Kent, "andykent / smoke." https://github.com/andykent/smoke, 2009. [updated 21 May 2009; cited 3 May 2011].

[70] appendTo, "appendto / jquery-mockjax." https://github.com/appendto/jquery-mockjax. [updated 26 Mar. 2011; cited 1 Jun. 2011].

[71] M. James, "Silk Icons." http://www.famfamfam.com/lab/icons/silk/. [cited 27 Jun. 2011].

[72] Open Source Initiative, "The MIT License (MIT)." http://www.opensource.org/licenses/mit-license.php. [cited 11 Jun. 2011].

[73] Open Source Initiative, "The BSD 2-Clause License." http://www.opensource.org/licenses/bsd-license.php. [cited 11 Jun. 2011].

[74] The Apache Software Foundation, "Apache License, Version 2.0." http://www.apache.org/licenses/LICENSE-2.0, Jan. 2004. [cited 11 Jun. 2011].

[75] Free Software Foundation, Inc., "GNU GENERAL PUBLIC LICENSE." http://www.opensource.org/licenses/mit-license.php, June 2007. [updated 29 Jun. 2007; cited 11 Jun. 2011].

[76] The jQuery Project, "jQuery project : Licence." http://jquery.org/license/, 2010. [cited 2 May 2011].

[77] Pivotal Labs, "pivotal / jasmine." https://github.com/pivotal/jasmine, 2011. [updated 9 Mar. 2011, cited 26 Apr. 2011].

[78] visionmedia, "visionmedia / jspec." https://github.com/visionmedia/jspec, 2010. [updated 30 Sep. 2010; cited 2 May 2011].

[79] jquery, "jquery / qunit." https://github.com/jquery/qunit, 2011. [updated 20 Apr. 2011; cited 2 May 2011].

[80] jupiterjs, "jupiterjs / funcunit." https://github.com/jupiterjs/funcunit, 2011. [updated 2 May 2011; cited 2 May 2011].

[81] Yahoo! Inc, "yui / yuitest." https://github.com/yui/yuitest, 2011. [updated 18 Apr. 2011; cited 2 May 2011].

[82] Wikipedia, "Kata (programming)." http://en.wikipedia.org/wiki/Code_Kata, Mar. 2002. [updated 26 May 2011; cited 10 Jun. 2011].

[83] P.-P. Koch, "W3C DOM Compatibility - HTML." http://www.quirksmode.org/dom/w3c_html.html, 2011. [updated 4 Apr. 2011; cited 4 Jun. 2011].

[84] J. Newbery, "froots / jasmine-sinon." https://github.com/froots/jasmine-sinon. [updated 11 Feb. 2011; cited 29 May 2011].

[85] C. Johansen, "Jstdutil - A Ruby wrapper over JsTestDriver." http://cjohansen.no/en/javascript/jstdutil_a_ruby_wrapper_over_jstestdriver. [updated 4 Nov. 2009; cited 1 Jun. 2011].

[86] M. Hevery, "Google Group on JsTestDriver." http://code.google.com/p/js-test-driver/wiki/GettingStarted, Jan. 2010. [updated 21 Oct. 2010; cited 21 Mar. 2011].

[87] W3Schools, "AJAX Tutorial." http://www.w3schools.com/Ajax/Default.Asp. [cited 20 Jun. 2011].

[88] Macromates Ltd., "textmate the missing editor." http://lincoln.gsfc.nasa.gov/trl/Nolte2003.pdf. [cited 20 Jun. 2011].

[89] Pivotal Labs, "Class jasmine.Spy." http://pivotal.github.com/jasmine/jsdoc/symbols/jasmine.Spy.html, Feb. 2011. [updated 27 Feb. 2011; cited 20 Jun. 2011].

[90] D. Crockford, *JavaScript : The Good Parts*, ch. 4-5. OÂťReilly and Yahoo! Press, 2008.