Contents

1 Introduction

The testing community around JavaScript still has some ground to cover. It becomes clear when compared to other programming communities such as Ruby and Java. As illustrated by Mark Bates [1]:

> "Around the beginning of 2012, I gave a presentation for the Boston Ruby Group, in which I asked the crowd of 100 people a few questions. I began, 'Who here writes Ruby?' The entire audience raised their hands. Next I asked, 'Who tests their Ruby?' Again, everyone raised their hands. "Who writes JavaScript or CoffeeScript?' Once more, 100 hands rose. My final question: 'Who tests their JavaScript or CoffeeScript?' A hush fell over the crowd as a mere six hands rose. Of 100 people in that room, 94% wrote in those languages, but didn't test their code. That number saddened me, but it didn't surprise me."

JavaScript is a scripting language primarily used in web browsers to perform client-side actions not feasible through plain HTML and CSS. There are benefits of testing code in general, for reasons that we will come back to, but JavaScript is particularly important to test properly due to its dynamic properties (however, see section 3.1 and 3.5 for examples of static analysis), unwieldy syntax and aberrant object orientation support. Despite the wide variety of testing frameworks that exists for JavaScript, it is generally considered that few developers use them.

1.1 Motivation

Why testing, you may ask. Jack Franklin, a young JavaScript blogger from the UK, gives three reasons: it helps you to plan out your APIs, it allows you to refactor with confidence, and it helps you to discover regression bugs (i.e. when old code breaks because new code has been added). Writing tests to use a library before actually writing the library puts focus on intended usage, leading to a cleaner API. Being able to change and add code without fear of breaking something greatly accelerates productivity, especially for large applications. [3] Without tests, the ability to refactor (see section 3.5) is greatly hampered. Without refactoring, making changes becomes harder over time, the code becomes harder to understand, there are more bugs and you spend more time debugging [4, p. 47-49]. In order to avoid this, code should be tested, preferably writing tests first.

More than 90 % of today's websites use JavaScript [5] and its applications have become increasingly complex [6, question 23]. The potential risk of economic loss associated with untested code being put into production, due to undetected bugs, shortened product lifetime and increased costs in conjunction with further development and maintenance, constitutes the main motivation for this thesis.

When automatic regression tests are missing, making changes to the code is error prone. Issues related to browser compatibility or subtle dependencies between functions and events are easier to overlook in the absence of tests. Manually testing a web page with

all the targeted combination of browsers, versions and system platforms is usually not a viable option [7] so multi-platform automated testing is required.

To achieve maximal gain from testing, the tests need to be of high quality, which means that they should be maintainable and test the right thing. How to achieve this is an important part of this thesis, which is discussed in section 9.9.

Unit testing is particularly powerful when in combination with integration tests in a Continuous Integration (CI) build. This enables you to harness the power of CI, avoiding errors otherwise easily introduced as changes propagate and affect other parts of the system in an unexpected way. The integration tests will make developers aware if they are breaking previous functionality, when changing parts of the system that the JavaScript depends upon.

Testing JavaScript paves the way for test-driven development, which brings benefits in terms of the design becoming more refined, and increased maintainability. Tests can serve as documentation for the code and forcing it to be written in a testable manner, which in itself tends to mean adherence to principles such as separation of concerns, and single responsibility.

The goal with this thesis is to investigate why JavaScript testing has been performed to such a small extent, and what potential implications an increased amount of testing could provide for development and business value. Providing possible approaches to testing JavaScript under different circumstances are also part of the goal.

1.2 Background to Project

The first known testing framework JsUnit[1] was created in 2001 by Edward Hieatt [8] and since then several other test framework has appeared such as the testing framework for jQuery, which goes under the name QUnit[2], and JsUnits sequel Jasmine[3]. There are also tools for mocking (see section 3.2) such as Sinon.JS[4]. It seems as if the knowledge of how to smoothly get started, how to avoid making the tests non-deterministic and time consuming, and what to test, is rare. Setting up the structure needed to write tests is a threshold that most JavaScript programmers do not overcome [1] and thus, they lose the benefits, both short and long term, otherwise provided by testing.

In section 10.5, it is mentioned that examples are often simple and seldom provide a full picture. In contrast, the problem domain of this thesis is to focus on how to test the behaviour of JavaScript that manipulates DOM elements (see 3.4), fetches data using asynchronous calls, validates forms, communicates through APIs or manipulates the appearance of a web page. We also look at when and why you should test your JavaScript.

[1] `https://github.com/pivotal/jsunit`
[2] `http://qunitjs.com/`
[3] `http://pivotal.github.com/jasmine/`
[4] `http://sinonjs.org/`

2 Description of Work

Researching today's limited testing of JavaScript may be done from a multiple different points of view. There are soft aspects such as:

- Differences in attitudes towards testing between different communities and professional groups
- How JavaScript is typically conceived as a language and how it is used
- Knowledge about testing among JavaScript developers
- Economic viability and risk awareness

There are also more technical aspects:

- Testability of JavaScript code written without tests in mind
- Usability of testing tools and frameworks
- Reasons not to include frameworks in a project for the sole purpose of facilitating testing
- Limitations in what can be tested
- Complexity in setting up the test environment; installing frameworks, configuring build server, exposing functions to testing but not to users in production, etc.

2.1 Consequences of JavaScript testing

There are consequences (good and bad) of testing JavaScript both from a short and from a longer perspective. The development process is affected; through time spent thinking about and writing tests, shorter feedback loops, executable documentation and new ways of communicating requirements with customers. The the quality and maintainability of the end result is also likely to be positively affected. Making changes becomes easier when the application is supported by a rigorous set of tests, so ideally, the pace of development does not stagnate. The extra time required to set up the test environment and write the actual tests may or may not turn out to pay off, depending on how the application will be used and maintained.

2.2 Covering Common and Advanced Cases

Accounting for how to conveniently proceed with JavaScript testing should cover not only the simplest cases but also the most common and the hardest ones, preferably while also providing evaluation and introduction to available tools and frameworks. Many introductions and tutorials found for the testing frameworks today tends to focus on the simple cases of testing, possibly because making an impression that the framework is simple to use has been more highly prioritised than covering different edge cases of how it can be used that might not be relevant to that many anyway. To provide valuable guidance in how to set up a testing environment and how to write the tests, attention

must be paid to the varying needs of different kinds of applications. It is also important to keep in mind that the tests should be as maintainable as the system under test, to minimise maintenance costs and maximise gain.

3 Technical Background

This section gives an overview of concepts and tools relevant to understanding this thesis. Readers with significant prior knowledge about JavaScript testing may skip this section.

3.1 Lint Tools for JavaScript

Linting is a static analysis tool for detecting syntax errors, risky programming styles and failure to comply to coding conventions. There are lint tools available for JavaScript such as JSLint, JSHint, JavaScript Lint, JSure, the Closure compiler and PHP CodeSniffer. JSLint does provide some help to avoid common programming mistakes, but does not perform flow analysis [9] and type checking as a fully featured compiler would do, rendering proper testing routines the appropriate measure against programming mistakes.

3.2 Mocking and Stubbing

Mocking and stubbing involves simulation of behaviour of real objects in order to isolate the system under test from external dependencies. This is typically done in order to improve error localization and execution time, and to avoid unwanted side-effects and dependencies such as communication with databases, across networks or with a file system [10, ch. 2].

In JavaScript, tools for stubbing can be superfluous because you are able to manually replace functions with custom anonymous functions, that can have attributes for call assertion purposes. The stubbed functions can be stored in local variables in the tests and restored during teardown. This is what some refer to as VanillaJS [11, question 53]. It might come across as manual work that you can avoid by using a stubbing tool, but the benefits include fewer dependencies and sometimes more readable code, as mentioned in section 10. [11, questions 54-55]

Typical cases for using a stubbing or mocking framework rather than VanillaJS include when your assertion framework has support for it, as is the case for Jasmine, when you feel the need to do a complex call assertion, mock a large API or state expectations up front as is done with mocks. Bear in mind that this might be a symptom for that the code is in need of refactoring though, and strive for consistency by using a single method for stubbing – mixing VanillaJS with Jasmine spies and sinonJS stubs will make the tests harder to understand.

3.3 Browser Automation

Repetitive manual navigation of a web site is generally boring and time consuming. There are situations where manual testing is the right thing to do, such as when there is no need for regression testing or the functionality is too complicated to interact with for automated tests to be possible (but then the design should probably be improved). Most of the time, tasks can be automated. There are several tools available for automating a web browser: the popular open source Selenium WebDriver, the versatile but proprietary and windows specific TestComplete and Ranorex, the Ruby library Watir and its .NET counterpart WatiN, and others such as Sahi and Windmill.

Selenium WebDriver is a collection of language specific bindings to drive a browser, which includes an implementation of the W3C WebDriver specification. It is based on Selenium RC, which is a deprecated technology for controlling browsers using a remote control server. A common way of using Selenium WebDriver is for user interface and integration testing, by instantiating a browser specific driver, using it to navigate to a page, interacting with it using element selectors, key events and clicks, and then inspecting the result through assertions. These actions can be performed in common unit testing frameworks in Java, C#, Ruby and Python through library support that uses the Selenium-Webdriver API. [12]

There is also a Firefox plugin called Selenium IDE, that allows the user to record interactions and generate code for them that can be used to repeat the procedure or as a starting point in tests. In the remaining parts of this thesis, we will mean Selenium WebDriver when we say Selenium, and refer to Selenium IDE by its full name.

3.4 The DOM

The Document Object Model (DOM) is a model for the content, structure and style of documents [13]. It can be seen as the tree structure of elements that html code consists of.

3.5 Refactoring

Refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior", or the act of applying such changes [4, p. 46]. As mentioned in the Motivation section (section 1.1), refactoring prevents program decay and helps to maintain productivity. Just like testing, refactoring is no magic bullet that solves all problems and it is important to know when to do it and when it might not be worth the effort. Common rules of thumb are to refactor when doing something similar for the third time (a pragmatic approach to the DRY principle), when it helps to understand a piece of code, when it facilitates addition of a new feature, when searching for a bug and when discussing the code with others, e.g. in code reviews [4, p. 49-51].

Using refactoring tools can be very helpful to see which parts of the code is in most need of refactoring, and to automate certain refactoring actions by facilitating renaming,

method extraction, etc. [10, ch. 5]. There is some support in IDEs such as Visual Studio (using JSLint.VS[5], ReSharper[6] and/or CodeRush[7]), WebStorm IDE[8] (or IntelliJ Idea[9] using a plugin) and NetBeans[10]. There are also standalone statistical tools such as kratko.js[11] and jsmeter[12] (not to be confused with the Microsoft Research project[13]) that helps you to identify which objects have too many methods and which methods do too many things (or have too many arguments). Relying on metrics such as lines of code is of course not always appropriate due to different coding styles, but at least it provides an overall picture [14].

3.6 Build tools

There exists some general build tools that can be used for any programming language, these are often installed on build servers and integrated with version control systems. Examples include Jenkins, which is often configured and controlled through its web interface although it also has a CLI, and GNU Make, which is typically configured using makefiles and controlled through CLI. In addition to these, there are also language specific tools: Ruby has Rake, Java has Maven, Gradle and Ant, C# has MSBuild and NAnt.

Naturally, there are build tools designed specifically for JavaScript as well, Grunt[14] being the most popular, which can be installed as a node.js package, has plugins for common tasks such as lint, testing and minification, and can be invoked through CLI. [11, question 52] Jake and Mimosa are other well known and maintained alternatives. It is also possible to use Rake, Ant or similar. Just as JsTestDriver and Mocha have adapters for Karma and Jasmine (see section 9.8), Rake has evergreen[15] that allows it to run Jasmine unit tests. [15][16, question 6]

3.7 Setting up a test suite

Many people have never set up a testing framework in a project and thinks that it is a hard thing to do. It is typically not, unless you want to do something very special such as automatic test generation or integration with a service of some sort. However, it is worth giving careful thought on what combination of tools to use, to enable a pleasant workflow and ensuring that the tests can be written in a desirable style.

Build programs (see section 3.6) and version control systems play an important role in automating testing.

[5] http://jslint.codeplex.com/
[6] http://www.jetbrains.com/resharper/
[7] http://www.devexpress.com/Products/CodeRush/
[8] http://www.jetbrains.com/webstorm/features
[9] http://www.jetbrains.com/editors/javascript_editor.jsp?ide=idea
[10] https://netbeans.org/kb/docs/ide/javascript-editor.html
[11] https://github.com/kangax/kratko.js
[12] https://code.google.com/p/jsmeter/
[13] http://research.microsoft.com/en-us/projects/jsmeter/
[14] http://gruntjs.com/
[15] https://github.com/jnicklas/evergreen

4 Methods

The methods used were first and foremost qualitative in nature, in order to prioritise insight into the problem domain above quantitatively verifying hypotheses. The chance of finding the true difficulties of JavaScript testing was expected to increase with open questions.

The work of this thesis began with an extensive literature study and an overview of technologies and frameworks. Interviews of JavaScript developers of different background were performed and analysed. There was also some hands on evaluation of tools and frameworks, assessment of testability and impact of adding tests to existing projects.

In order to describe ways of writing tests for JavaScript, the practical work involved testing an existing application, performing TDD exercises from Test-driven JavaScript Development [17] and doing some small TDD projects during the framework evaluation. There were plans to have a workshop field study, where programmers would work in pairs to solve pre-defined problems using TDD, but in the end it was decided that it would be too difficult to extract useful data from such an activity.

4.1 Literature Study

Relevant books, articles and Internet sources were identified and skimmed, some were read through thoroughly. Over time, several new titles were published and some older literature was also included as the need became apparent.

4.2 Framework Overview and Evaluation

The following testing frameworks have been evaluated: Jasmine[16] (+ Jasmine-species[17]), qUnit[18], Karma[19], Mocha[20], JsTestDriver[21], Buster.JS[22] and Sinon.JS[23]. The code written while evaluating the frameworks is publicly available as git repositories under my github account *emilwall*, together with the LaTeX code for this report.

4.3 Interviews

Semi-structured case study interviews were used rather than surveys to gather individual views on the subject. This approach allowed for harnessing unique as well as common experiences which would not be picked up in a standardised survey. The interviews were

[16] `http://pivotal.github.com/jasmine/`
[17] `http://rudylattae.github.com/jasmine-species/`
[18] `http://qunitjs.com/`
[19] `http://karma-runner.github.io/`
[20] `http://visionmedia.github.com/mocha/`
[21] `https://code.google.com/p/js-test-driver/`
[22] `http://docs.busterjs.org/`
[23] `http://sinonjs.org/`

between 20 and 60 minutes long and were conducted both in person and via Internet video calls.

4.3.1 Interview Considerations

The preparations before the interviews included specifying purpose and which subjects to include, select interviewees, preparing questions and adjust the material to fit each interviewee. The interviews took place rather late so that preliminary results and insights from the literature study could be used as basis for the discussions.

The purpose of the interviews was to investigate attitudes and to get a reality check on ideas that had emerged during previous work. Selecting the interviewees was to a large extent done based on availability, but care was also taken to include people outside of Valtech and to get opinions from people with different background (front-end, back-end, senior, junior, etc.). Enquires were made in Valtech's intranet, my supervisor asked his contacts via his Twitter account and I contacted some people via mail. This led to five interviews and one email conversation, which can all be found in the Appendix.

The interviews were performed in swedish to allow for a more fluent conversation and minimise risk of misunderstandings. They were transcribed (see Appendix), each question was given a number, and the most relevant parts were translated and included in this report with reference to the question numbers. The interviewees were asked prior to the interviews if it was ok to record the conversation and if they wanted to be anonymous, everyone was ok with being recorded and mentioned by name.

The interviewees received questions beforehand via mail which most of them answered before the interview. This allowed the interviews to focus on the vital parts rather than personal background and opinions about the JavaScript language. The questions that was sent out before the interviews were mainly about previous experience with JavaScript and testing, frameworks, attitudes towards the language, difficulties with testing and opinions and observations on benefits of testing.

4.3.2 The Interviewees

The first person I interviewed was Johannes Edelstam, an experienced Ruby and JavaScript developer, organizer of the sthlm.js meetup group, a helping hack, and a former employee of Valtech, now working at Tink. He has a positive attitude towards JavaScript as a programming language and has extensive experience of test driven development.

Next up was Patrik Stenmark, a Ruby and JavaScript developer since 2007. He is also an organizer of a helping hack and a current employee at Valtech. He considers JavaScript to be inconsistent and weird in some aspects but appreciates the fact that it is available in browsers and has developed large single page applications (SPA) in it.

The third person that was interviewed was Marcus Ahnve, a senior developer and agile coach that has been in business since 1996 working for IBM, Sun Microsystems and ThoughtWorks, and as CTO for Lecando and WeMind. He is currently working at Valtech. He is an experienced speaker and a founder of Agile Sweden, an annual conference

since 2008. He is also experienced with test driven development in Java, Ruby and JavaScript.

The next interview was with Per Rovegård, a developer with a Ph.D. in Software Engineering from Blekinge Institute of Technology. He has worked for Ericsson and is currently a consultant at factor10 where he has spent the last year developing an AngularJS application. He is the author of the programatically speaking blog and have done several talks at conferences and meetups, most recently about Angular and TDD at sthlm.js on Oct 2 this year but the interviews took place in Aug over Skype.

The last interview was with Henrik Ekelöf, a front-end developer that has seven years of professional experience with JavaScript. He has previously worked as webmaster and web developer at Statistics Sweden and SIX and is now technical consultant at Valtech. I met him in person during my introduction programme in Valtech where he had a session with us about idiomatic JavaScript, linting and optimizations, but this interview was done over Skype since he works out of town.

4.3.3 Other People Involved

As can be seen at the end of the Appendix, there were some email conversations with people that was not interviewed as well. Among those are: Fredrik Wendt, who is a senior developer and consultant at Squeed specializing in team coaching with coding dojos, TDD and agile methodologies. David Waller, teacher at Linnéuniversitetet in a course about Rich Internet Applications with JavaScript. Marcus Bendtsen, teacher at Linköpings Universitet in a course about Web Programming and Interactivity.

5 Previous work and Delimitations

There exists academic papers on testing web applications and a few focus on JavaScript specifically. Some focus on automatically generating tests [18] and although useful for meeting code coverage criteria, these methods will not be discussed to any great length here since such tests are hard to maintain and likely to cause false positives when refactoring code. In this thesis, there will be more focus on how to employ test driven development than achieving various degrees of code coverage.

Heidegger et al. cover unit testing of JavaScript that manipulates the DOM of a web page [19] and Ocariza et al. have investigated frequency of bugs in live web pages and applications [20]. These are of more interest to this thesis since they are aimed at testing of client side JavaScript that runs as part of web sites.

The main source of reference within the field of JavaScript testing today is Test-Driven JavaScript Development [17] by Christian Johansen which deals with JavaScript testing from a TDD perspective. Johansen is the creator of Sinon.JS[24] and a contributor to a number of testing frameworks hosted in the open source community.

[24] `http://sinonjs.org/`

The scope of this thesis has been to look mainly at testing of *client side* JavaScript. This meant that framework specialised for server side code such as vows[25] and cucumis[26] are not included in the evaluation part. Testing client side code is by no means more important than the server side, but it can be argued that it is often harder and the parallels to testing in other languages are somewhat fewer since the architecture typically is different.

Frameworks that are no longer maintained such as JsUnit[27] and JSpec[28] have deliberately been left out of the evaluation. Others have been left out because of fewer users or lack of unique functionality; among these we find TestSwarm, YUI Yeti and RhinoUnit. They are still useful tools that can be considered but including them would have a negative impact on the rest of the evaluation work because of the extra time consuming activities that would be imposed.

6 Real world experiences

6.1 Description of the asteroids example

(setup, förutsättningarna, motivering av valet)

6.2 Attempts, observations

(upplåst implementation, icke-determinism)

6.3 Conclusions

tools and testability

6.4 Testability issues with main.js in asteroids application

Despite partitioning the JavaScript of the asteroids applications into separate classes, the problem of the canvas element not being available in the unit testing environment was not mitigated. The main.js file contained around 200 lines of code that could not be executed by tests without further refactoring since they were executed in a jQuery context (i.e. using `\$(function ()...)` that included the selector `$("#canvas")`. Efforts to load this code using ajax were of no gain, so the solution was instead to expose the contents of the context as a separate class and inject the canvas and other dependencies into that class. This required turning many local variables into attributes of the class to make them accessible from main.js such as the 2d-context.

[25] `http://vowsjs.org/`
[26] `https://github.com/noblesamurai/cucumis`
[27] `https://github.com/pivotal/jsunit`
[28] `https://github.com/liblime/jspec`

The problem was not solved entirely through this approach though, since some parts could not be extracted. The event handling for key-presses necessarily remained in main.js and since that code could not be executed by unit tests, changes to global variables used in the event handler does not cause any unit test to fail, even though the application will crash when executed in an integration test. The problem could be solved just as before, by extracting the event handler code into a separate class that can be tested. The problem is that the event handler modifies local variables in main.js, which still can't be tested, so there has to be some test setup code to mock these when making them global, and this affects the design in a bad way by introducing even more global state.

Same goes with the main loop, which contains logic to draw grid boundaries. When refactoring main.js the main loop was left in main.js (which can not be tested) and this introduced a bug that was reproduced only when using the particular feature of displaying the grid. The feature is not important and could be removed. The bug could also be fixed by making a couple of variables globally accessible as attributes of the rendering class, but that too would introduce a code smell. A better solution was to move it into the rendering class, as it semantically belongs there.

As a consequence of being unable to extract all code from main.js into testable classes, I started to consider using selenium tests. This could actually be argued to be a sound usage of selenium because main.js is basically the most top level part of the application and as such can be more or less directly tested with decent coverage using integration tests. The Internet sources that I could find regarding how to use selenium with JavaScript depended on node.js and mocha, which I was inexperienced with using at the time. Consequently, I spent an afternoon trying to get things to work but without any real results. Posting on stack overflow asking for help could possibly have been a way forward instead of settling with manual testing.

One has to be careful when adding code to beforeEach, setUp and similiar constructs in testing frameworks. If it fails the result is unpredictable. At least when using Jasmine with JsTestDriver, not all tests fail even when the beforeEach causes failure, and subsequent test runs may produce false negatives even though the problem has been fixed. This is likely due to optimizations in the test driver and is especially apparent when the system under test is divided into multiple files and contain globally defined objects (rather than constructors). In this case, game.js contains such a globally defined object and its tests commonly fails after some other test has failed, even after passing the other test. Restarting the test driver and emptying the cache in the captured browser usually solves this problem, but is time demanding.

6.5 JsTestDriver evaluation

When resuming from sleep on a mac the server needs to be stopped and the browsers need to be manually re-captured to the server, or else the driver hangs when trying to run the tests. This is both annoying and time consuming. However, the problem is not present on a windows machine (and it might not be reproducable on all mac machines either). In the interview with Johannes Edelstam, he agreed that this is one example of

something that deters people from testing [11].

Definitions are not cleared between test runs, meaning that some old definitions from a previous test run can remain and cause tests to pass although they should not because they are referring to objects that no longer exists or that tests can pass the first time they are run but then crash the second time although no change has been made to the code. Some of these problems indicate that the tests are bad, but it is inconvenient that the tool does not give you any indication when these problems occur, especially when there is false positives.

If there is a syntax error in a test, the JsTestDriver still reports that the tests pass. For example:

```
setting runnermode QUIET
.................................
Total 35 tests (Passed: 35; Fails: 0; Errors: 0) (23,00 ms)
  Chrome 27.0.1453.94 Windows: Run 36 tests (Passed: 35; Fails: 0;
Errors 1) (23,00 ms)
    error loading file: /test/sprites-spec/sprite-spec.js:101: Uncaught
SyntaxError: Unexpected token )
```

As a developer, you might miss the "error loading file" message and that not all 36 tests were run, because the first line seems to say that everything went fine. Sometimes Jasmine does not run any test at all when there is a syntax error, but does not report the syntax error either. It is therefore recommended that you pay close attention to the terminal output and check that the correct number of tests were run rather than just that there was no failures. This is impractical when running the tests in a CI build because the build screen will typically display success even if no tests were run. It can be of help to keep a close look on the order in which files are loaded and also to keep the console of a browser open in order to be notified of syntax errors [21].

Many of these problems can be said to stem from accidental integration tests or other errors in the tests. It should be noted however that proper stubbing of dependencies can be a daunting task, especially if dependency injection is not handled in a smooth way. In JavaScript, dependency injection can be argued to be harder than in for instance C or java because of the absence of class interfaces. The sinon.JS framework does simplify compared to manual stubbing (which on the other hand is exceptionally simple to do with JavaScript) but there is still issues of doing tradeoffs between dedicating many lines of code to stubbing, quite often having to repeat the same code in multiple places, or risk introducing bugs in the tests themselves. As a programmer you have to be very methodical, principled and meticulous not to miss some detail and write an accidental integration test. Such mistakes leave you with misleading failure result messages and sometimes the tests fail because of the order in which they are executed or similar, rather than because of an actual bug in the system under test.

Another source of problems is when global state is modified by constructors of different classes. For instance, when extracting code from main.js into rendering.js, part of that code was involved with initiating the grid which is shared between all the sprites in the application (through its prototype) and this meant the the grid was not defined unless

the rendering class had been instantiated. This imposed a required order in which to run the tests and is an example of poor maintainability due to optimization.

Deficiencies such as these are important to note because they pose potential reasons to why JavaScript developers don't test their code. If using the tools and frameworks is perceived as cumbersome and demanding, fewer will use them and those who do will consider it worth doing so in fewer cases.

When a function is defined within a constructor it is hard to stub unless you have an object created by the constructor available. In some cases you don't because the system under test creates an instance by itself and then you are (as far as I know) out of options except for stubbing the entire constructor (this produces a lot of code in the tests) or changing the system under test to increase testability, for instance by having the instance passed as an argument (which allows for dependency injection but can be odd from a semantic point of view) or defining the functions that you need to stub on the prototype of the constructor instead of in the constructor (which allows for easy stubbing but is less reliable since another class/object can modify the function as well). Often it is possible to come up with a way that increases testability without having a negative impact on readability, performance, etc. of the system under test, but not always so. Regardless, this requires a skilled programmer and effort is spent on achieving testability rather than implementing functionality which may feel unsatisfactory.

JsTestDriver is not perfect. When refreshing and clearing the cache of a captured browser, you have to wait for a couple of seconds before running your tests or else the browser will hang and you have to restart the server. This wouldn't be such a problem if it wasn't because definitions from previous test runs remain in the browser between runs. For instance, if a method is stubbed in two different tests but only restored in the one that is run first, the tests will pass the first time they are run but then fail the second time. Realizing that this is what has happened is far from trivial so as a beginner you easily get frustrated with these small issues, since you might refresh the browser quite frequently in the process of finding out.

Having spent many hours debugging, I finally decided to do a thorough check that no test depended on another test or part of the application that is not supposed to be tested by a specific test. In short, I wanted to ensure that the tests I'd written were truly unit tests. In order to do this, I created a copy of the application repository, deleted every file in the copy except for one test and the corresponding part of the application. Then I configured a JSTD server with a browser to run only that test, and repeated the process for every test. This method does not guarantee absence of side effects or detecting tests that do not clean up after themselves, but being able to run a test multiple times without failing, in complete isolation with the part of the application that it is supposed to test, at least gives some degree of reassurance that all external dependencies have been stubbed. If any external dependency has been left unstubbed the only way for the test to pass is if the code exercising that dependency is not executed by the test, and if a test does not clean up after itself it is likely to fail the second time it runs although this too depends on how the tests are written.

6.6 Testability and other issues with adding tests to an existing application

Sometimes it can be hard to know whether or not to stub library functions such as $.isFunction or if you should trust that they behave as expected and steer the control flow via their input and the global state instead. The same applies to simple functions you have written yourself that you think are free of bugs. Not stubbing external dependencies leads to fewer lines of test setup and teardown code and usually better test coverage but can also impose a danger of the unit tests becoming more brittle and similar to integration tests.

When adding tests to an existing application, it is easy to lose track of what has and what has not been tested. Having access to a functional specification of the application can be of help but it might be unavailable, incomplete or outdated. Then you have to make a specification of your own, in order to be systematic about what tests you write. This can be done top-down by looking at user stories (if there are any), talking with the product owner and the users (if any) or identify features to test through manual testing. It can also be done bottom-up by looking at the source code that is to be tested and come up with ideas regarding what it appears like all the functions should be doing. The latter is what was done before adding tests to the asteroids application because there was no documentation available and the application was so small that a bottom-up approach seemed feasible and likely to generate better coverage than doing a top-down specification. The way this was done was by writing test plans in the form of source code comments in the spec files for each class.

Each function was analyzed with respect to what was considered to be its expected behavior, such as adding something to a data structure or performing a call with a certain argument, and then a short sentence described that behavior so that it would not be forgotten when writing the actual tests later. Since tests are typically small, one might think that it could be a good idea to write the tests directly instead of taking the detour of writing a comment first, but my experience was that a comment is a lot faster to write than a complete test, makes up for fewer lines of code and avoids getting stuck with details about how to write the test.

Another useful method for knowing what tests to write was to write tests for every bug that was detected, i.e. regression testing. This should be done before fixing the bug so you can watch the test fail, which increases the chance that the test will fail if the same bug is introduced again. Additionally, some aspects of TDD can be employed even when the code lacks tests by writing tests that document any changes you make to the application. Be careful that you do not break existing functionality though, and that the tests focus on behavior rather than implementation details. The recommended approach is writing tests for the application in its existing form before starting to change it, since this will increase understanding of how it works and reduce risk of breaking existing functionality when refactoring later. These alternatives are still worth mentioning though, because sometimes code needs to be refactored in order to make it testable.

Traditionally, coverage criteria has been a central concept in software testing and is still today in many organizations (citation needed). When doing TDD however, the need for thinking in terms of coverage is reduced as every small addition of functionality is tested

beforehand. There is no need to test specific implementation details because that will only make the system harder to change. If a certain function feels complex and likely to contain bugs, the recommended way in TDD is to take smaller steps, refactoring and testing new components separately rather than trying to achieve different kinds of graph and logic coverage for the complex function. When adding tests in retrospect it makes more sense to think about coverage, which may be done when the system is starting to feel complete in order to reduce risk of bugs. There are various tools available for ensuring that relevant parts of an application are exercised by tests and it is often relevant to design tests based on edge cases and abnormal use. As a tester, it tends to pays off having the attitude of trying to break stuff instead of just testing the so called happy flow. Different types of coverage criteria can help in formalizing this, as described in Introduction to Software Testing by Ammann and Offutt [22].

To illustrate why achieving a certain coverage criteria should not be a goal in itself, I decided to write tests for the finite state machine (FSM) in the asteroids.Game object of the asteroids application. Achieving Clause Coverage[22, p. 106] for 18 lines of production code (asteroids.Game.FSM.start) took almost 100 lines of test code, see commit 61713c of `https://github.com/emilwall/HTML5-Asteroids`. This is not that much, but it didn't provide much value either as no bug was found.

6.7 Stubbing vs refactoring

When an application is tightly coupled, stubbing becomes a daunting task. What you end up with is deciding whether you should compromise the unit tests by not stubbing everything, refactor the code to reduce the amount of calls that needs to be stubbed, or stub all dependencies. The first alternative bodes for unstable tests that might fail or cause other tests to fail for the wrong reasons. Refactoring might introduce new bugs and should probably only be done if it simplifies the design and makes the code more readable. Stubbing all dependencies might result in too much code or force you to complicate the testing configuration so that some code is run between each test. One case where this tradeoff had to be made was when writing tests for classes that depended on the Sprite class, such as the Ship class. It uses the Sprite class both for its "exhaust" attribute and for its prototype. Luckily, the Sprite constructor does not modify any global state, so in this case not stubbing the Sprite class before parsing the Ship class is acceptable. In the unit tests however, any calls to methods defined in Sprite are preferably stubbed, since they should be tested separately.

To detect improper stubbing, I ran each test isolated with just the file it was supposed to test. A problem with this was that trying to stub a function which is not defined produces an exception in order to prevent you from doing typos. This could be solved by saving the implementation in a local variable, defining the function to be an empty function, stub it with sinon.JS and then restore and re-set it to the original implementation, but this is inconvenient so instead I opted towards being careful not to miss any calls that should be stubbed. There is a point with interpreting the system under test before running any test code, since that allows for detection of typing mistakes and other integration issues.

6.8 Deciding when and what to test

Testing is especially important for large applications. It is extra valuable when new functionality is added because it helps to verify that old functionality is not broken and that the new code is structured appropriately. [2, questions 6-7]

You should focus on testing behaviour rather than appearance and implementation details. [11, question 10] Rather than testing that a certain class inherits from another, test that the methods of that class behaves as one would expect. Whether or not that is dependent on the inheritance patterns is mainly relevant for stubbing considerations - you may want to replace the prototype of an object in tests so that you can check that there are no unexpected dependencies. These are lessons learnt from working with the asteroids application, see section 6.4.

Another thing that came up during the interview with Edelstam was that when something feels like it is hard to test, it is likely that any test you write will become rather brittle as the code changes in the future. The proposed solution was to avoid testing it unless the code can be refactored so that testing becomes easier. [11, question 30] When writing the tests for the asteroids application, I deliberately chose to write tests even when it felt hard or felt like it provided little value, to see whether this made the application harder to test later and if people would remove the bad tests during the workshop.

Because unit tests are typically fast, it is common practice to prioritise decent coverage and corner cases in the unit tests rather than in integration, UI, e2e (end to end) and other types of tests. When tests can be executed fast, they are more useful when changes are made to the code. This is especially important when someone other than the person who wrote the code is making the changes, or when some time has passed since the code was written. [2, questions 22-24]

Naturally, the more important a certain functionality is in relation to business value, the more effort should be put into e2e and similar tests related to it. Inherently complex parts of the code and code that is likely to change should be tested by unit tests to allow for refactoring and increase chance of discovering bugs, whereas simple code that is not expected to change can be tested manually. However, it is typically hard to know beforehand if the code you are writing is subject to change, so a compromise by writing a few simple tests for that code may pay off. [2, questions 28-29 and 33]

6.9 Meetup open space discussion

During a talk at a meetup on python APIs (2013-05-22 at Tictail's office, an e-commerce startup based in Stockholm), the speaker mentioned that their application depended heavily on JavaScript. It turned out that they had done some testing efforts but without any lasting results. During the open space after the talks, testing became a discussion subject in a group consisting of one of the developers of the application, among others. The developer explained that they had been unable to unit test their JavaScript because the functionality was so tightly coupled that the only observable output that they could

possibly test was the appearance of the web page, via Selenium tests. He sought reassurance that they had done the right thing when deciding not to base their testing on Selenium due to instability (tests failing for the wrong reasons) and time required to run the tests. He also sought answers to how they should have proceeded.

The participants in the discussion were in agreement that testing appearance is the wrong way to go and that tests need to be fast and reliable. The experience with testing frameworks seemed to vary, some had used Jasmine and appreciated its behaviour driven approach and at least one had used Karma but under its former name Testacular. The idea that general JavaScript frameworks such as AngularJS could help in making code testable and incorporating tests as a natural part of the development was not frowned upon. The consensus seemed to be that in general, testing JavaScript is good if done right, but also difficult.

6.10 Ideas spawned when talking about this thesis

During my work on this thesis, I have explained to numerous people what it is that I'm doing. Typically, I've started out with saying something like "I'm looking at testing of JavaScript". Depending on if the person asking knows a lot about JavaScript or not, the conversation then might proceed in different directions, but the most common follow up is that I explain further that I'm looking at why people don't do it, when and how they should do it and what the problems and benefits are. Especially I'm looking at the problems.

One not so uncommon response is that testing of JavaScript probably is so uncommon because people programming in JavaScript often have a background as web graphic designers, without that much experience of automated testing. Another common conception is that JavaScript in practise is usually not testable because it has too much to do with the front-end parts of an application, so tests are inevitably slow, unmaintainable and/or unreliable because of the environment they have to run in.

7 Problems in testing

7.1 Asynchronous events

JavaScript is often loaded asynchronously in browsers to improve performance in page loads and minimize interference with the display of the page. Asynchronous events are also commonly used, e.g. AJAX calls. Testing asynchronous events can be hard because the order in which things happen is not predetermined, so the number of possible states can be very large, thus demanding many convoluted tests to cover all the relevant cases. It can also be hard to know if the asynchronous code has finished execution or not, leading to assertions being executed at the wrong time.

Two basic approaches to testing asynchronous code are to force the execution into a certain path by faking the asynchrony, and to set up waiting conditions for when assertions should be executed. When forcing the execution path, the code is not executed

asynchronously but you can test different scenarios and thereby get a good feel of how the code will behave when run asynchronously. When using waiting conditions, timeouts need to be set for when tests should fail if waiting conditions are not met.

7.2 DOM manipulation

A key to successful testing of JavaScript with DOM manipulation is to separate the logic from the DOM manipulation as much as possible. One technique that was presented at the XP conference in Trondheim 2010[29] was to define a view layer of jQuery-related and similar code, and update the view as a separate task in the business logic. This allows for test driven development to be carried out much more effortlessly than if the logic is tightly coupled with the DOM manipulation. [16, question 4]

7.3 Form validation

Validation of forms are commonly done both server and client side so that users do not have to wait for requests to be sent to and returned from a server in order to validate input, while maintaining the security that server side validation offers. Preferably, these validations are automatically kept in sync by frameworks and ideally it should be enough to test the server side validation, but that is sometimes just as hard and to be sure you should test both.

You typically need to test for both type I and type II errors (false positives and false negatives). If forms are prevented from being submitted because of an error in the validation (type I error), the number of successful conversions (website goal achievements) is likely to decrease. Allowing invalid forms to be submitted (type II error) can either cause prolonged feedback for the user, loss of form data so the user has to re-type and loss of context due to loading an error page resulting in decreased conversion rates or, if the validation fails at the server side too, security issues.

I have a real life example of a false positive in form validation. This year when I was about to buy a domain name, my personal number was interpreted wrongly so that I got an error message saying that I was not old enough. I reported the bug on twitter and it was quickly fixed so that I could finish my registration, but I imagine most people would have chosen another provider instead. Had there been proper testing of the form validation, this would probably have been detected at an earlier stage, without losing potential customers. [23]

7.4 GUI testing

A graphical User Interface (GUI) can consist of many components and provide a large number of possible interactions. The GUI is usually the top layer of an application and

[29] Agile Processes in Software Engineering and Extreme Programming 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010, Proceedings

is therefore a common target for both integration testing and manual testing, but it can also be unit tested.

7.4.1 Web vs desktop

7.4.2 Size and sequencing problems in GUI testing

7.4.3 Regression testing of GUI

7.4.4 Automation of GUI testing

Automation vs TDD (???)

Record and replay

Model based (FSM)

7.5 Private APIs

8 Testing culture

Different people have different opinions about testing: how it should be done, when it pays off and how exciting it is. Some see testing as a tool that helps to provide structure and improve the design of code, whereas others see it as a way to detect bugs and prevent others from braking the code. Most people agree that testing is hard, but many struggle to do it anyway. Attitudes towards testing are essential to whether or not it is carried out or not, and to how worthwhile it turns out to be in the end.

8.1 State of events

It's possible that the overall quality of JavaScript code has improved over the last few years, partly as a consequence of testing methodologies entering the JavaScript community. One might argue that the new application frameworks has brought not only new capabilities (as mentioned in section **??**) but also brought people with background from backend-programming with tests, experienced with for example Ruby on Rails development, to JavaScript. In fact, most of the new frameworks have been developed by people with such background. This has influenced the way JavaScript code is written and given birth to a novel testing culture. [16, questions 12-15]

8.2 Individual Motivation and Concerns

What defines successful testing is not which tools are used, the degree of coverage that is achieved or how strictly a certain set of principles or methodologies are followed. Successful testing is defined by how easy it is to make changes, find errors and understand

the code. In order to get there, developers must feel that testing is meaningful and that they are allowed to spend time on testing as part of their professional tasks.

8.3 Project risks

Different projects have different fault tolerance. Sometimes a single bug can cause loss of millions of dollars, whereas in other projects the economic impact can be relatively small. The risk with untested JavaScript is especially high when the code supports business critical operations. For instance, there have been several cases of banks and finance systems being fined for not reporting transactions to the government [24] or giving faulty advice to investors [25] due to application failure. A webshop may lose orders and any website that is perceived as broken can harm trademarks associated with it and change people's attitudes for the worse.

9 Draft without title

9.1 Patterns

Rather than proposing best practices for JavaScript testing, the reader should be made aware that different approaches are useful under different circumstances. This applies both to choice of tools and how to organise the tests.

9.2 Why don't people test their JavaScript?

Considering all the different options in available frameworks, one is easily deceived into believing that the main reason why people don't test their JavaScript is because they are lazy or uninformed. This is not necessarily true, there are respectable obstacles for doing TDD both in the process of fitting the frameworks into your application and in writing the JavaScript code in a testable way.

9.3 JsTestDriver and Jasmine integration problems

For instance, when setting up JsTestDriver[30] (JSTD) with the Jasmine adapter there are pitfalls in which version you're using. At the time of writing, the latest version of the Jasmine JSTD adapter (1.1) is not compatible with the latest version of Jasmine (1.3.1), so in order to use it you need to find an older version of Jasmine (such as 1.0.1 or 1.1.0) or figure out how to modify the adapter to make it compatible. Moreover, the latest version of JSTD (1.3.5) does not support relative paths to parent folders when referencing script files in jsTestDriver.conf although a few older versions do (such as 1.3.3d), which is a problem if you want to place the test driver separate from the system under test rather than in a parent folder, or if you want to reference another framework such as Jasmine if it is placed in another directory.

[30] `https://code.google.com/p/js-test-driver/`

9.4 Testability, TDD, exposing code to tests, counter-intuitiveness of writing tests first

Regardless whether or not the frameworks are effortlessly installed and configured or not, there is still the issue of testability. It is common to argue that TDD forces developers to write testable code which tends to be maintainable. This is true in some respects, but one has to bear in mind that JavaScript is commonly used with many side-effects that may not be easily tested. More importantly, it is common to place all the JavaScript code in a single file and hide the implementation using some variant of the module pattern[26, p. 40], which means that only a small subset of the code is exposed as globally accessible functions, commonly functions that are called to initialize some global state such as event listeners. In order to test the functions, they need to be divided into parts, which will typically have to be more general in order to make sense as stand-alone modules. This conflicts with the eagerness of most developers to just get something that works without making it more complicated than necessary.

9.5 Refactoring and manual testing

Many JavaScript developers are used to manually test their JavaScript in a browser. For someone not experiences with testing, this gives a relatively early feedback loop and although it does not come with the benefits of design, quality and automated testing that TDD does, it tends to give a feeling of not doing any extra work and getting the job done as fast as possible. Developers do not want to spend time on mocking dependencies when they are unsure if the solution they have in mind will even work. Once an implementation idea pops up, it can be tempting to just try it out rather than writing tests. If this approach is taken, it may feel like a superfluous task to add tests afterwards since that will typically require some refactoring in order to make the code testable. If the code seems to work good enough, the developer may not be willing to introduce this extra overhead, for good reasons [2, question 43].

There is a serious risk involved in refactoring untested code[4, p. 17], since manually checking that the refactoring does not introduce bugs is time consuming and difficult to do well. However, leaving the code untested means even greater risk of bugs and the refactoring may be necessary in the future anyway, in which case it will be even harder and more error-prone. This problem can be avoided by writing tests first.

If refactoring is required in order to make the code testable, the architecture likely needs to change. Sometimes introducing a new object or changing how some dependency is handled is all that is needed. The important thing is to not continue in a direction that will cause the codebase to deteriorate over time. [2, question 34] If a lot of work goes into mocking dependencies then the tests take longer to write, require more maintenance and the need for integration tests increases. The proper remedy is usually not to reduce the isolation of the unit tests but to refactor the architecture so that each unit becomes easier to isolate. [2, question 42]

9.6 AngularJS, Jasmine and Karma

The AngularJS framework uses Jasmine and Karma in the official tutorial.

> "Since testing is such a critical part of software development, we make it easy to create tests in Angular so that developers are encouraged to write them" [27]

This is likely a large contributing factor for increasing the probability of Angular developers testing their JavaScript.

9.7 Definitions

A mock has pre-programmed expectations and built-in behaviour verification [17, p. 453].

Because JavaScript has no notion of interfaces, it is easy to accidentally use the wrong method name or argument order when stubbing a function [17, p. 471].

9.8 The frameworks

Programming languages in use today typically have frameworks to help with standard structure and other generic problems. JavaScript is certainly no exception, the number of web application frameworks that focus on JavaScript has increased a lot during the last few years alone.

A full evaluation of the most popular MVC and testing frameworks is not within the scope of this thesis, but others have done it [3][28]. Popular JavaScript testing frameworks include assertion frameworks such as Jasmine, qUnit, expect.js and chai, and drivers/test runners such as Mocha, JsTestDriver, Karma and Chutzpah[31] which may have their own assertion framework built in but are typically easy to integrate with other assertion frameworks using adapters or via built-in support.

9.9 High quality tests

As mentioned in section 1.1, tests should be maintainable and test the right thing. Otherwise, responding to changes is harder, and the tests will tend to cause frustration among the developers instead of detecting bugs and driving the understanding and development of the software [29].

The criteria for maintainability in this context are that the tests should have low complexity (typically short test methods without any control flow), consist of readable code, use informative variable names, have reasonably low level of repeated code (this can be accomplished through using Test Utility Methods [30, p. 599]), be independent of implementations and have meaningful comments (if any). Structuring the code according to a testing pattern such as the Arrange-Act-Assert [31] and writing the code so that it

[31] `http://chutzpah.codeplex.com/`

reads like sentences will help in making the code more readable, in essence by honouring the communicate intent principle [30, p. 41].

Testing the right thing means focusing on the specifications and behaviour that provides true business value rather than e.g. coverage criteria, and to avoid writing tests for things that does not really matter. Typically some parts of the system will be more complex and require more rigorous testing but there should also be consistency in the level of ambition regarding testing across the entire application. If some part of the code is hard to test it is likely to be beneficial in the long run to refactor the design to provide better testability than to leave the code untested.

10 Interview summary

It seems that in the last couple of years, the number of people testing their JavaScript has increased significantly[11, question 1]. This has been observed through asking people that do it to raise their hands during tech talks, two years ago only a few raised their hands when asked such a question whereas now almost every single one does it.

According to Edelstam, a likely reason for this change is that the tools have become better and that there are more examples of how to do it. This has caused the opinion that JavaScript is too user interface centered to recede, as people have realized how it can be done. Few people were ever against TDD or testing in general, much thanks to positive experiences from testing in Ruby on Rails projects, which actually act as great examples of that testing wide ranges of interfaces is possible and that many feel that it is necessary. Perhaps the most common reason for not testing is that the code has not been written in a testable fashion. [11, questions 2-3]

A common experience when writing tests is that you put a lot of effort into the tests and do not write that much production code in comparison, but that can be a good thing! Because then you spend more time thinking about the problem and possible abstractions, which tends to lead to elegant solutions. If you write the tests before the code, you will run into the same problems as you would have done if you wrote the code first, the difference is that you get to think about the design rather than staring at incomplete code when solving the problem. [11, question 8]

The feeling of being limited and not productive when writing tests can stem from a badly chosen level of ambition or that the focus of the tests is wrong, which in turn can be based on poor understanding of what tests are good for. Coding without tests can be much like multitasking, you get an illusion of being more productive than you actually are. One of the positive effects of TDD is that it can prevent you from losing track of direction, and helps you in making clear delimitations, since trying to be smart by allowing a function to do more than one thing means more tests. Testing will not automatically provide you with good ideas regarding where you are heading, but once you have gotten such an idea, testing tends to be easier and help you discover new aspects and scenarios which might would have been left unnoticed without tests. [11, question 8]

When deciding what to test, it pays off to focus on parts that are central to how the application is perceived, for instance pure logic and calculations might be more important than pictures and graphs. An error that propagates through the entire application is more serious than if a single picture is not displayed properly. If a test turns out to be difficult to write or frequently needs to be replaced by another test, it is usually worth considering not testing that part at all. [11, questions 9-10]

In June this year, Kent Beck wrote the following tweet[32]:

> "that was tough–it almost never takes me 6 hours to write one test. complicated domain required extensive research."

One of the responses was that many would have given up on writing that test. Beck replied that if you don't know how to write the test, you don't know how to write the code either [32]. What many probably fail to realize about why testing can be time consuming and hard, is that when writing tests you encounter problems that you would have to solve anyway. The difference is that you solve the problems by formulating tests rather than staring at production code for the same amount of time. [11, question 11]

Tools are an important part of facilitating testing, so that people are not so deterred by the initial effort required to get started. Yeoman is one example of a framework that can help you in quickly getting started with a project that is structured so that testing becomes easier. For already existing projects, the increased maturity of tools such as PhantomJS and Mocha is also truly helpful. [11, questions 11-12 and 20]

Error reports are useful feedback that tests provide. The quality of these reports vary depending on which testing frameworks you are using and how you write your tests. When using PhantomJS to run tests, some test failures require you to run the tests in a browser in order to get good error reports. [11, question 12]

An important difference between using a headless browser such as PhantomJS to run your tests compared to JsTestDriver, or other drivers that allow you to test in several browsers, is that a headless browser provides no information about how your code performs on different JavaScript implementations. Reasons why you might still decide to do so include speed, easier integration with build tools such as Jenkins, and that it yields the same results as long as the tests focus on logic rather than compatibility. [11, questions 13-15]

One could argue that JavaScript is better suited for testing than most other programming languages because of the built in features than often make stubbing frameworks redundant. The object literals can be used to define fake objects and it is easy to manually replace a function with another and then restore it. An advantage with using manually written fakes is that it tends to make the code easier to understand since knowledge about specific frameworks or DSLs (Domain Specific Language) is not required. [11, questions 20-21]

The problems that some people experience with testing frontend interfaces sometimes have to do with poor modularity. For instance, the presentation layer should not be responsible for calling large APIs. Tools such as RequireJS can be used to handle

[32] A tweet is a message sent to many using the social media `www.twitter.com`

dependencies but if each part of the application has too many or large dependencies, mocking becomes a daunting task. Typically, these kinds of problems can be solved by introducing new layers of logic and services that separate responsibilities and allows for much cleaner tests. [11, question 23]

A common case of user interface testing is form validation. Being test driven does not necessarily mean that you should test that the form exists, has a working submit button and other boilerplate things, unless the form is written in a way that is unique or new to you in some way. A typical approach would rather be to write the code for the form and then add a test that searches for a non-existing validation. The difficult part here is to strike a balance and be alert to when the code is getting so complicated that it is time to start testing, and to avoid writing tests when they are in the way and provide little value. [11, questions 24-25]

Being too religious about testing principles leads to conflicts like "writing tests take too much time from the real job". If the tests do not provide enough value for them to be worth the effort then they should be written differently or not at all. There is no value in writing tests just for the sake of it. Thinking about architecture and the end product is usually a good thing, because you need awareness of the bigger picture in order to prioritize correctly and make sure everything will fit together in the end. There is the same risk with tests as with other pieces of code, sometimes you are especially proud or fond of a certain test and unwilling to throw it away. In order to avoid that situation it is often better to think ahead and try things out than to immediately spend time writing tests. [11, question 27]

Writing implementation specific tests is a common phenomena that can stem from poor experience of writing tests, extreme testing ambitions or, perhaps most commonly, poor understanding of the system under test. It means that you write the tests based on details about the implementation that you have in mind rather than basing them on what the system should do from an interface point of view. These kind of tests should be avoided since they tend to be hard to understand and usually have to be changed whenever an improvement of the implementation is considered.

A large number of tests is not necessarily a good thing, it is harder to overview and maintain a large collection of tests. Unless the system inevitably is so complicated that extensive testing is justified, it rarely pays off to strive for 100 % coverage, since it takes too much time and the scenarios that might cause problems are at risk of being overlooked anyway. [11, question 28]

Tests can serve to prevent people from breaking code and as examples that help new people understand how an application works and how to continue development [11, questions 31-32]. Tests also help to give a clear image of how components fit together and can be a way to concretize a feature or bug.

10.1 Testing private APIs

Most problems with APIs, both external and private, are similar to problems with any external dependencies that you do not have full control over. The tests need to be kept up

to date and in order to do so, integration tests are needed. Sometimes, testing towards a true implementation is problematic because it takes too long time, manipulates data that should not be touched or because the owner of the API does not accept large amounts of traffic or requires payment for it. Then you typically run the integration tests less often and stick to mocked APIs with fake objects to compensate. [2, questions 19-20]

Testing private APIs differs from testing calls to an external API in that the latter are often well documented, versioned and rarely changes. When using an external API, the main concerns are making sure that the correct version is used and that the API is called in a correct way with respect to that version. A private API on the other hand may change frequently which means that the fake objects representing requests and responses of the API in other tests need to change as well. Private APIs need to be tested because endpoints frequently change and may be used in several places. It is then important to have integration tests that can detect when the fakes need to change. [11, questions 34, 36]

Introducing versioning for private APIs is not always feasible, for at least two reasons. There can be difficulties in keeping track of the different versions and there can be a risk of hampering development since the parts that use the API are dependent on the new version of the API to be released before they can be updated. However, there exists testing techniques that can be employed regardless if versioning is in place or not. One such technique is to write tests for what characterizes an object that the API serves as an interface for, to be able to determine if an object is a valid response or request. These tests can then be used on both real objects and on the fake objects that are used in other tests, which means that when the API changes the tests can be updated accordingly and then the tests involving the fake will fail. Changing the fakes so that the tests passes will hopefully result in that any incompatibility is discovered since the fakes are used in other parts of the testing suite as well. [11, question 34]

Testing a private API is not the same thing as testing a private method. In JavaScript there is formally no private methods, only functions not exposed to the current scope, nevertheless methods that are not intended to be called from the outside can be considered private. It is a common principle that you do not test a private method, if you feel the need to, then that method is probably too complex to be private and the functionality that it is meant to provide should be moved into a new object or similar. Private methods represent implementation details, tests should be focused on behaviour and functionality, not implementation. [11, questions 62-63]

10.2 Culture, Collaboration and Consensus

Testing is not about eliminating the risk of bugs or ensuring that every line of code is executed exactly the way it was originally meant to be executed. In order to better understand the main benefits of testing, it is recommended to read books such as Clean Code, and to work together with experienced programmers and testers. Without this understanding there is a risk of frustration, inability to decide which tools to use and difficulties in motivating choices related to testing. Collaboration problems may occur when members of a team have different opinions about the purpose of testing. [11,

question 38]

To get as much benefit as possible from testing, every developer should be involved in the testing effort and preferably the product owner and other stakeholders too. The work of specifying stories can involve agreeing on testing scenarios, thinking about what the desired behavior is and coming up with examples that can be used as input and expected output in end-to-end tests. [11, questions 39-40][2, question 30]

The attitude towards and experience with testing varies heavily, so as a developer you can be forced to not test as much as you would otherwise want to, in order to avoid frustration and dips in productivity by testing other people's code. You should advocate methods that you believe in, especially in the beginning of a new project, but in the end adjusting to the culture is also essential since most benefits of testing appear only after a long term commitment by the whole team. [2, questions 31-32]

10.3 Frameworks and tools

There are a number of BDD frameworks available that are meant to bring testing closer to the specification and story work, as proposed in section 10.2. They aim to mimic the way stories are written to create a ubiquitous natural language that can be understood by both programmers and non-programmers. The way this is done is by building sentences with a Given, Then, When (GTW) form. Cucumber[33] is a popular framework that was designed for this in the Ruby programming language and there is a JavaScript version called Cucumber.js[34]. Yadda,[35] Kyuri,[36] Cucumis[37] and Jasmine-given[38] are other examples of JavaScript GTW-frameworks. [33, section 8.4]

Since different assertion frameworks have different syntax, they influence the readability of tests. This is presumably one of the main reasons why BDD frameworks such as Jasmine and Buster have become popular. [16, question 7]

Apart from testing frameworks, your ability to write tests is also influenced by what application framework you are using [16, question 8]. According to Edelstam, if you are able to choose, it is often preferable to have small rather than "magic bullet" frameworks for large projects because they tend to be less obtrusive. Exceptions may include when the framework is very well known to the developers or when there is a perfect fit between what you want to achieve and what the framework is designed for, but one should bear in mind that requirements tend to evolve. [11, questions 48-50] Stenmark had another view on things and instead recommended small frameworks only for small projects, since the value of bindings and other features tend to outweigh adjustment problems. [2, questions 12-14]

Using a suitable application framework and doing test driven development can be mutually beneficial. The framework helps to create structure that simplifies testing and the

[33] http://cukes.info/
[34] https://github.com/cucumber/cucumber-js
[35] https://github.com/acuminous/yadda
[36] https://github.com/nodejitsu/kyuri
[37] https://github.com/noblesamurai/cucumis
[38] https://github.com/searls/jasmine-given

testing process further improves the quality of the code, allowing you to make better use of the framework and providing incentive for even better structure. [2, question 15]

10.4 Testability and Selenium

As mentioned in section 3.3, Selenium is currently the most popular tool for browser automation. Sometimes it is the only way to test the JavaScript of an application without rewriting the code[2, question 43] (as mentioned in section6.9) but then tests tend to be brittle and provide little value. In general, since Selenium tests take such time to run they should only cover the most basic functionality in a smoke test fashion. [2, questions 16-17] Testing all possible interaction sequences is rarely feasible and should primarily be considered if it can be done fast, such as in a single page application (SPA) where sequences can be tested without reloading the page. [11, question 44]

With Selenium IDE (see section 3.3), there is a possibility of recording a certain interaction and generate code for it. This could potentially be used to reproduce bugs by letting the user that encountered the bug record the actions that led to it, and communicate with a developer so that proper assertions are added at the end of the sequence. This has some interesting implications, but experience shows that it is hard to do it in practice since it requires the users to be trained in how to use the plugin and to invest extra time whenever an error occurs. [11, questions 42-43]

10.5 Patterns, Examples and Documentation

In guides on how to use different JavaScript testing frameworks, examples are often decoupled from the typical use of JavaScript - the Web. They tend to merely illustrate testing of functions without side effects and dependencies. Under these circumstances, the testing is trivial and most JavaScript programmers would certainly be able to put up a test environment for such simple code.

Examples are useful when learning idiomatic ways of solving problems. Code tends to end up being more complicated than examples you read because it is hard to come up with useful abstractions that make sense. Those who write examples to illustrate a concept always have to find a tradeoff between simplicity, generality and usefulness, and tend to go for simplicity [11, questions 56-57]. This can for example be observed in [4, p. 13-45] where the tests and their setup are omitted despite their claimed importance.

Combining different concepts may help to achieve code with good separation, that can be tested by simple tests. Today blog posts and books about patterns in JavaScript are in abundance and you can often find examples in the documentation of frameworks. When it comes to examples of testing in general there are several classics to refer to [34][30]. For examples of JavaScript testing specifically the alternatives have been scarce until recently [17][33][35][36].

10.6 Spikes in TDD

Although writing tests first is a recommended approach in most situations, there is a technique for when you need to try something out before you write tests for it, without compromising testability. Dan North, the originator of BDD, came up with a name for this technique: spiking [37]. The idea is that you create a new branch in your version control repository, and hack away. Add anything you think might solve your problem, don't care about maintainability, testability or anything of the sort. If you find yourself not knowing how to proceed, discard all changes in the branch and start over. As soon as you get an idea about how a solution could look like you switch back to the previous branch and start coding in a test first fashion. [11, question 59]

There is an ongoing discussion about whether or not you have to start over after a spike. Liz Keogh, a well known consultant and core member of the BDD community, has published posts about the subject in her blog, in which she argues that an experienced developer can benefit from trying things out without tests (spiking) and then stabilizing (refactoring and adding tests) once sufficient feedback has been obtained to reduce the uncertainty that led to the need for spiking [38]. She argues that this allows her to get faster feedback and be more agile without compromising the end result in any noticeable way. In another post, she emphasizes that this approach is only suitable for developers who are really good at TDD, while at the same time claiming that it is more important to be "able to tidy up the code" than "getting it right in the first place" [39]. It may seem like an elitist point of view and a sacrilege towards TDD principles but in the end, whatever makes you most productive and produces the most valuable software has raison d'être.

Counterintuitive it may seem, throwing away a prototype and starting from scratch to test drive the same feature can improve efficiency in the long run. The hard part of coding is not typing, it is learning and problem solving. A spike should be short and incomplete, its main purpose is to help you get your mind set on what tests you can write and what the main points in a solution would be. [11, question 60]

10.7 The DRY principle in testing

The point of the Don't Repeat Yourself (DRY) principle is *not* that the same lines of code cannot occur twice in a project, but that the same *functionality* must not occur twice. Two bits of code may seem to do the same thing while in reality they don't. This applies to eliminating duplication both in production code and tests, do not overdo it since that will harm maintainability rather than improve it. Sometimes it is beneficial to allow for some duplication up front and then refactor once you have a clear picture about how alike the two scenarios actually are, if necessary. [11, questions 69-70]

10.8 Impact of Node.js on testing

Aside from the increased awareness and knowledge of testing that Node.js has brought to the JavaScript community from other communities such as Ruby on Rails, the runtime

itself facilitates testing since it enables different testing tools to be distributed through npn and run in the terminal. Previously, you had to load a html page in a browser in order to run your tests. [2, question 9]

10.9 The JavaScript Community

In order to understand the problems with testability that can be seen in JavaScript applications, it is important to understand how many were first introduced to the language. When jQuery was released in 2006, a coding style evolved based on handlers with DOM manipulation through selectors and asynchronous callbacks. Since tutorials were written in a quick-and-dirty fashion, even experienced developers failed to apply principles for writing testable and maintainable code. Others lacked the background of software engineering altogether. [2, question 10]

10.10 Time required to run tests

Tests that depend on the DOM (see section 3.4) tend to be relatively slow, so it would be impractical to have hundreds of unit tests with DOM manipulation if you want to be able to run them often [2, questions 21-22]. Selenium tests are also known to be slow, the same principle applies there. Only running tests that are connected with changes that have been made is one way to get around this, there is support for this in many testing tools. An alternative is to separate the fast tests from the slow, and configure Autotest[39] or some tool based on Guard[40] such as guard-jasmine[41] to run the fast tests each time a file changes, and only run the slow tests before sending the code to a central source control repository or deploying the application.

10.11 High quality tests

There is a lot of previous work on testing patterns and how to write concise, useful and maintainable tests [30, part III][33, ch. 3-5][17, p. 461-474][35, p. 86-87][36, p. 13-14]. In todays BDD frameworks there is often a possibility to separate the basic functionality from the special cases by organizing the specs in nested describes. This can provide an overview of what an application does just by looking at the spec output and is commonly seen in open source projects [2, question 42].

10.12 Learning TDD

In order to get started with testing JavaScript, you need to understand why you should do it [11, question 38] and then practise with katas and begin to use testing in your projects.

[39] `http://autotest.github.io/`
[40] `https://github.com/guard/guard/`
[41] `https://github.com/netzpirat/guard-jasmine/`

11 Analysis of Economic Impacts

- Identify cases where companies and authorities have suffered economic loss due to bugs in JavaScript code vital to business critical functions of web sites
- Assess risks and undocumented cases of manifested bugs
- Estimate maintenance costs of code that lack tests compared to well-tested code
- Draw conclusions about how test-driven development can either shorten or prolong the time required to develop a product, depending on programmer experience and the size and type of the application being developed
- Costs associated with acquiring developers with the skills necessary to write tests

12 Testability

- Search for JavaScript code to analyse, do representative selection for different areas of application
- Analyse testability of selected code segments by looking at how the applications are partitioned, how well single-purpose principles are followed, and what parts of the code is exposed and accessible by tests
- Discuss validity factors, whether the selection is fair and really representative, how open source affects quality, etc. (self-criticism)

13 Framework evaluation

- Perform and document complexity of installation process
- Try different "Getting Started" instructions
- Compare syntax, functionality and dependencies
- Discuss suitable areas of application
- Create example implementations of different types of tests to illustrate practical use

14 Adding tests to an existing project

The current content of this section could be rewritten and placed in the method section, while replaced by actual results.

- Identify what parts of the project are currently testable
- Identify what functionality should be tested

- Decide on testing framework and motivate choice based on circumstances and previous analysis
- Set up the testing environment so the tests can run automatically on a build server
- Write the actual tests and continually refactor the code, while documenting decisions in this report
- Analyse impact on code quality and number of bugs found

15 Future

(new books, new attitudes, old problems, tool convergence and revenue/open source)

16 Lessons learned

Used differently, not just another programming language

Difficult environment for testing because of many dependencies and events

Full coverage not compatible with BDD and not being implementation specific

References

[1] Mark Bates. *Testing Your JavaScript/CoffeeScript*. URL: `http://www.informit.com/articles/article.aspx?p=1925618` (visited on 04/09/2013).

[2] Patrik Stenmark. *Interview*. Aug. 7, 2013.

[3] Jack Franklin. *.NET Magazine Essential JavaScript: the top five testing libraries*. Oct. 8, 2012. URL: `http://www.netmagazine.com/features/essential-javascript-top-five-testing-libraries` (visited on 08/12/2013).

[4] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. ADDISON-WESLEY, 1999. ISBN: 0201485672.

[5] W3Techs - World Wide Web Technology Surveys. *Usage of JavaScript for websites*. URL: `http://w3techs.com/technologies/details/cp-javascript/all/all` (visited on 04/09/2013).

[6] Henrik Ekelöf. *Interview*. Aug. 28, 2013.

[7] John Resig. *JavaScript testing does not scale*. URL: `http://ejohn.org/blog/javascript-testing-does-not-scale/` (visited on 04/09/2013).

[8] Edward Hieatt and Robert Mee. "Going Faster: Testing The Web Application". In: *IEEE Software* (Mar. 2002), p. 63.

[9] Douglas Crockford. *JSLint - The JavaScript Code Quality Tool*. URL: `http://www.jslint.com/lint.html` (visited on 05/01/2013).

[10] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, 2004. ISBN: 0131177052.

[11] Johannes Edelstam. *Interview*. June 27, 2013.

[12] SeleniumHQ. *Platforms Supported by Selenium*. URL: http://docs.seleniumhq.org/about/platforms.jsp (visited on 08/09/2013).

[13] W3C DOM IG. *Document Object Model (DOM)*. Jan. 19, 2005. URL: http://www.w3.org/DOM (visited on 09/09/2013).

[14] Juriy Zaytsev. *Refactoring Javascript with kratko.js*. URL: http://perfectionkills.com/refactoring-javascript-with-kratko-js/ (visited on 09/11/2013).

[15] Philip Rose. *Stack Overflow - Build process tools for JavaScript*. URL: http://stackoverflow.com/questions/7719221/build-process-tools-for-javascript (visited on 08/21/2013).

[16] Marcus Ahnve. *Interview*. Aug. 12, 2013.

[17] Christian Johansen. *Test-Driven JavaScript Development*. ADDISON-WESLEY, 2010. ISBN: 9780321683915.

[18] Shay Artzi et al. "A Framework for Automated Testing of Javascript Web Applications". In: *International Conference on Software Engineering '11* (May 2011), pp. 571–580.

[19] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. "DOM Transactions for Testing JavaScript". In: *Proceeding TAIC PART'10 Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques* (Sept. 2010), pp. 211–214.

[20] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. "JavaScript Errors in the Wild: An Empirical Study". In: *IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)* (Nov. 2011), pp. 100–109.

[21] Mike Jansen. *Avoiding Common Errors in Your Jasmine Test Suite*. URL: http://blog.8thlight.com/mike-jansen/2011/11/13/avoiding-common-errors-in-your-jasmine-specs.html (visited on 06/12/2013).

[22] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge university press, 2008. ISBN: 9780521880381.

[23] Emil Wall. *Twitter - I get a validation error for my personal number, "Not old enough to buy .SE domain. Min age: 18" I'm born 890101...* URL: https://twitter.com/erif89/status/378536014337171456 (visited on 09/13/2013).

[24] Madelene Hellström. *Bugg kostade SEB 2.5 miljoner*. May 25, 2010. URL: http://computersweden.idg.se/2.2683/1.322567/bugg-kostade-seb-25-miljoner (visited on 08/30/2013).

[25] Computer Sweden. *Bugg kostade 1.5 miljarder*. Sept. 23, 2011. URL: http://computersweden.idg.se/2.2683/1.405796/bugg-kostade-15-miljarder (visited on 08/30/2013).

[26] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN: 9780596517748.

[27] Igor Minar, Misko Hevery, and Vojta Jina. *Angular Templates*. URL: http://docs.angularjs.org/tutorial/step_02 (visited on 02/02/2013).

[28] Sebastian Porto. *Sebastian's blog: A Comparison of Angular, Backbone, CanJS and Ember*. Apr. 12, 2013. URL: http://sporto.github.io/blog/2013/04/12/comparison-angular-backbone-can-ember/ (visited on 08/13/2013).

[29] Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. PRENTICE HALL, 2009. ISBN: 9780132350884.

[30] Gerard Meszaros. *xUnit Test Patterns - refactoring test code*. ADDISON-WESLEY, 2007. ISBN: 9780131495050.

[31] Cunningham & Cunningham Inc. *Arrange Act Assert*. URL: `http://c2.com/cgi/wiki?ArrangeActAssert` (visited on 04/24/2013).

[32] Kent Beck. *Twitter - It almost never takes me 6 hours to write one test*. URL: `https://twitter.com/KentBeck/status/350039069646004224` (visited on 07/26/2013).

[33] Marco Emrich. *Behaviour Driven Development with JavaScript*. Developer.Press, 2013. ISBN: 9781909264113.

[34] Kent Beck. *Test-Driven Development By Example*. ADDISON-WESLEY, 2002. ISBN: 9780321146533.

[35] Mark Ethan Trostler. *Testable JavaScript*. O'Reilly Media, 2013. ISBN: 9781449323394.

[36] Evan Hahn. *JavaScript Testing with Jasmine*. O?Reilly Media, 2013. ISBN: 9781449356378.

[37] Dan North. *Twitter - I think I was the first to name and describe the strategy of Spike and Stabilize but there were definitely others already doing it*. URL: `https://twitter.com/tastapod/statuses/371352069812527105` (visited on 08/26/2013).

[38] Liz Keogh. *Beyond Test Driven Development*. June 24, 2012. URL: `http://lizkeogh.com/2012/06/24/beyond-test-driven-development/` (visited on 08/24/2013).

[39] Liz Keogh. *If you can't write tests first, at least write tests second*. Apr. 24, 2013. URL: `http://lizkeogh.com/2013/04/24/if-you-cant-write-tests-first-at-least-write-tests-second/` (visited on 08/24/2013).