How to test your JavaScript

Emil Wall

July 3, 2013

The final version will have title page and endpaper generated from
`http://pdf.teknik.uu.se/pdf/exjobbsframsida.php` and
`http://pdf.teknik.uu.se/pdf/abstract.php`.
Hence, this page and the abstract are temporary, to be replaced in the final version.

Abstract

Abstract goes here... Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam sollicitudin varius libero ac consectetur. Nullam ornare, massa et sagittis consectetur, neque mi scelerisque arcu, in fringilla lectus risus non arcu. Suspendisse vestibulum tellus id mauris lacinia non hendrerit nibh tempor. Proin tempor interdum justo et elementum. Ut ultricies adipiscing ipsum et pharetra. Vestibulum pretium luctus est, quis egestas augue luctus et. Praesent volutpat pharetra lectus vitae elementum.

Integer fringilla ligula eu sem semper tincidunt. Nullam mi lacus, blandit non sollicitudin eget, tempor eu ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Morbi ornare sem et purus consequat ac adipiscing nunc tincidunt. Curabitur nisi ante, ornare vel adipiscing et, scelerisque vitae erat. Etiam blandit egestas magna, quis dapibus nulla euismod quis. Sed interdum interdum malesuada. Suspendisse lacinia imperdiet laoreet. Maecenas ullamcorper laoreet nunc ac egestas. Cras consequat elit eu lacus sollicitudin ut pharetra magna venenatis. Suspendisse scelerisque condimentum pulvinar. Mauris ut tellus sit amet nulla porttitor tristique. Suspendisse eleifend erat sed nisi lacinia eu lacinia metus porta. Nulla pretium, risus eget semper laoreet, dolor odio malesuada eros, at mattis enim turpis gravida felis. Aliquam adipiscing varius nibh, ac auctor eros bibendum non.

Acknowledgment

Thanks goes to my supervisor Jimmy Larsson for providing me with valuable feedback and connections, to my reviewer Roland Bol for guiding me through the process and giving useful and constructive comments on my work, to all my wonderful colleagues at Valtech which never fails to surprise me with their helpfulness and expertise, to my girlfriend Matilda Kant for her endurance and support and to my family and friends (and cats!) for all the little things that ultimately matters the most.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Background to project . . . 2

2 Description of Work **3**
- 2.1 Consequences of JavaScript testing . . . 3
- 2.2 Covering common and advanced cases . . . 4

3 Methods **4**
- 3.1 Interview Considerations . . . 5
- 3.2 Interview Questions . . . 5
  - 3.2.1 Formalities . . . 5
  - 3.2.2 The interviewee . . . 5
  - 3.2.3 JavaScript in general . . . 5
  - 3.2.4 JavaScript testing experience . . . 6
  - 3.2.5 Challenges in testing . . . 6
  - 3.2.6 Benefits of testing . . . 6
  - 3.2.7 Adding tests to existing application . . . 6

4 Previous work and Delimitations **7**

5 Analysis of Economic Impacts **7**

6 Testability **8**

7 Framework evaluation **8**

8 Adding tests to an existing project **8**

9 Draft without title **9**
- 9.1 Patterns . . . 9
- 9.2 Why don't people test their JavaScript? . . . 9
- 9.3 JsTestDriver and Jasmine integration problems . . . 9
- 9.4 Testability, TDD, exposing code to tests, counter-intuitiveness of writing tests first . . . 9
- 9.5 Manual testing, psychology, refactoring . . . 10
- 9.6 AngularJS, Jasmine and Karma . . . 10
- 9.7 Definitions . . . 10

10 Real world experiences **10**
- 10.1 Testability issues with main.js in asteroids application . . . 10
- 10.2 JsTestDriver evaluation . . . 12
- 10.3 Stubbing vs refactoring . . . 15
- 10.4 Deciding what to test . . . 15
- 10.5 Meetup open space discussion . . . 16
- 10.6 Ideas spawned when talking about this thesis . . . 16

1 Introduction

The testing community around JavaScript still has some ground to cover. The differences in testing ambitions becomes especially 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. Due to the dynamic nature of the language, there is typically little static analysis performed on JavaScript code compared to code written in a statically typed compiled language. Granted, there are tools available such as JSLint, JavaScript Lint, JSure, the Closure compiler, JSHint and PHP CodeSniffer. JSLint is perhaps the most popular of these and does provide some help to avoid common programming mistakes, but does not perform flow analysis[2] and type checking as a fully featured compiler would do, rendering proper testing routines the appropriate measure against programming mistakes. After all, 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 and poor object orientation support. Despite the wide variety of testing frameworks that exists for JavaScript, it is generally considered that few developers use them. 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.

1.1 Motivation

JavaScript code is presumably becoming increasingly commonly used as part of business critical operations, considering that more than 90 % of today's websites use JavaScript[3] and it may be assumed to be especially prevalent in sites with a lot of content and functionality. The economic risk of having untested JavaScript is especially high when the code is connected to critical operations. For instance, application failure for a webshop may cause loss of orders and any web site that is perceived as broken can harm trademarks associated with it and change people's attitude for the worse. Moreover, 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 easily overlooked instead of being detected by tests prior to setting the site into production. Manually testing a web page with all the targeted combination of browsers,

versions and system platforms is not a viable option[4] so multi-platform automated testing is required.

High quality tests are maintainable and test the right thing. If these conditions are not met, 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[5]. 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[6, p. 599]), be based on interfaces rather than a specific implementation and have meaningful comments (if any). Structuring the code according to a testing pattern such as the Arrange-Act-Assert[7] and writing the code so that it reads like sentences can help in making the code more readable, in essence by honouring the communicate intent principle[6, p. 41]. Testing the right thing means focusing on the behaviour that provides true business value rather than trying to fulfill some coverage criteria, testing that the specification is fulfilled rather than a specific implementation and to find a balance in the amount of testing performed in relation to the size of the system under test. Typically some parts of the system will be more complex and require more rigorous testing but there should be some level of consistency in the level of ambition regarding testing across the entire application. Specifically, 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.

Unit testing is particularly powerful when run in combination with integration test in a CI build[1]. Then you are able 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. This will make developers changing parts of the system that the JavaScript depends upon aware if they are breaking previous functionality.

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 key principles such as separation of concerns, and single responsibility.

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

1.2 Background to project

Writing tests for JavaScript is nothing new, the first known testing framework JsUnit was created in 2001 by Edward Hieatt[8, 9] and since then several other test framework has appeared such as QUnit [10] and JsUnits sequel Jasmine [11], as well as tools for

[1] Continuous Integration build servers are used for automatic production launch

mocking[2] such as Sinon.JS[12]. 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 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. In contrast, the problem domain of this thesis is to focus on how to test the behaviour of JavaScript that manipulates DOM elements (Document Object Model, the elements that html code consists of), interacts with databases and fetches data using asynchronous calls, as well as when and why you should do it.

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

[2]mocking and stubbing involves simulation of behavior of real objects in order to isolate the system under test from external dependencies

about and writing tests, shorter feedback loops, executable documentation and new ways of communicating requirements with customers. The business value of the end result is also likely to be affected, as well as the quality and maintainability of the code. Ideally, the pace of development does not stagnate and making changes becomes easier when the application is supported by a rigorous set of tests. 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 Methods

The methods used are first and foremost qualitative in nature, in order to prioritise insight into the problem domain above quantitatively verifying hypotheses. The chance of finding out the true reasons to why JavaScript is tested to such a small extent increases with open questions. Specifically, aside from literature studies, the main method of this thesis work has been to perform and analyse interviews of JavaScript programmers (mainly those concerned with user interface). There has also been some hands on evaluation of tools and frameworks, and assessment of testability and impact of adding tests to existing projects. In order to describe methods of writing tests for JavaScript, the practical work involved testing an existing application, performing TDD as described in Test-driven JavaScript Development[13] and doing some small TDD projects during the framework evaluation. Another method used was a workshop field study, where programmers were allowed to work in pairs to solve pre-defined problems using TDD.

The following testing frameworks have been evaluated: Jasmine[11] (+ Jasmine-species[14]), qUnit[10], Karma[15], Mocha[16], JsTestDriver[17], Buster.JS[18] and Sinon.JS[12]. 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.

Semi-structured 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.

3.1 Interview Considerations

The preparations before the interviews where included specifying purpose and which subjects to include, select interviewees, put together questions and other material and adjust the material to fit each interviewee. The interviews took place rather late to ensure that the interviewer could obtain a solid background and domain knowledge. Each interview was summarised in writing and the collected material was structures and analysed to increase conciseness of results.

3.2 Interview Questions

3.2.1 Formalities

The interviews took place in calm, undisturbed locations, and began with a short recap on the background and purpose of the interviews. The interviewee was informed that the purpose of the interview was to gain a better understanding of different aspects of JavaScript testing. What problems and benefits exists and how it is connected with other software engineering practices and tools.

- Is it ok if I record our conversation?
- Do you want to be anonymous?

3.2.2 The interviewee

- What kind of applications do you typically develop with JavaScript?
- What tools and frameworks have you used? What roles have they played in your development processes?
- Which are your favourites among the frameworks? Why?

3.2.3 JavaScript in general

- How productive do you feel when coding in JavaScript compared to other languages?
- How do you typically perceive JavaScript code written by others?
- What advanced features of JavaScript do you use, such as prototypal inheritance, dynamic typing and closures?
- How do you think the JavaScript syntax and features impacts maintainability?
- How would you assess the probability of making mistakes while coding in JavaScript?

3.2.4 JavaScript testing experience

- What is your experience with unit testing of JavaScript?
- What is your experience with UI testing?
- What is your experience with integration and end-to-end tests?
- Have you practiced test driven development with JavaScript? To what extent? Has this been helpful? (if not, why? what did you do instead?)
- Have you used any mocking and stubbing tools? Which, and what has been your experience with these?

3.2.5 Challenges in testing

- How do you go about determining what to test?
- What principles do you apply when writing the tests? (short test methods, avoiding control flow, code duplication)
- Have you ever set up a testing environment? If so, did you find it hard? If not, do you imagine it to be difficult?

3.2.6 Benefits of testing

- In your opinion, what are the main benefits from testing your JavaScript?
- When do you think testing JavaScript pays off?
- Have you ever had tests that impaired your productivity by being too hard to change or even understand?
- Has tests helped you in debugging and quickly finding the source of a bug?
- Has testing helped you discover bugs in the first place? Has this saved you from trouble further on?
- Has testing helped your design?
- What role has JavaScript testing played in any continuous integration you've had?
- What type of JavaScript coding do you think is best suited for TDD?

3.2.7 Adding tests to existing application

- Have you ever been given the task of adding tests to an existing (JavaScript) application?
- Was this hard?

- What changes in the application were required in order to be able to write the tests?
- Did you feel safe in changing the application or were you afraid that you'd might introduce new bugs?

4 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[19] 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[20] and Ocariza et al. have investigated frequency of bugs in live web pages and applications[21]. 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[13] by Christian Johansen which deals with JavaScript testing from a TDD perspective. Johansen is the creator of Sinon.JS[12] and a contributor to a number of testing frameworks hosted in the open source community.

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[22] and cucumis[23] 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[9] and JSpec[24] 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.

5 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

6 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)

7 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 illustrare practical use

8 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

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 (JSTD)[17] 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.

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[25, 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 Manual testing, psychology, refactoring

The fundamental problem is probably that most developers are used to manually test their JavaScript in a browser. This gives an 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 not sure that 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. There is also a risk involved in refactoring untested code[26, p. 17], since manually checking that the refactoring does not introduce bugs is time consuming and difficult to do well, although there is an exception when the refactoring is required in order to add tests. This is because 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.

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[13, 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[13, p. 471].

10 Real world experiences

10.1 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.

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.

10.2 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).

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[28].

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.

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[29].

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[29, 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.

10.3 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.

10.4 Deciding what to test

During the interview with Johannes Edelstam, one of the things that came up was that you should focus on testing behaviour rather than appearance and implementation details. This is a good excuse for not testing that a certain class inherits from another but

rather focus on 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.

Another thing that came up during the interview with Johannes 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 (TODO check this from the recording!) was to avoid testing it unless the code can be refactored so that testing becomes easier. 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.

10.5 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.

10.6 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.

References

[1] Mark Bates. *Testing Your JavaScript/CoffeeScript*. Last checked: April 9, 2013. URL: http://www.informit.com/articles/article.aspx?p=1925618.

[2] Douglas Crockford. *JSLint - The JavaScript Code Quality Tool*. Last checked: May 1, 2013. URL: http://www.jslint.com/lint.html.

[3] W3Techs - World Wide Web Technology Surveys. *Usage of JavaScript for websites*. Last checked: April 9, 2013. URL: http://w3techs.com/technologies/details/cp-javascript/all/all.

[4] John Resig. *JavaScript testing does not scale*. Last checked: April 9, 2013. URL: http://ejohn.org/blog/javascript-testing-does-not-scale/.

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

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

[7] Cunningham & Cunningham Inc. *Arrange Act Assert*. Last checked: April 24, 2013. URL: http://c2.com/cgi/wiki?ArrangeActAssert.

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

[9] Github. *pivotal/jsunit*. Last checked: April 3, 2013. URL: https://github.com/pivotal/jsunit.

[10] The jQuery Foundation. *QUnit: A JavaScript Unit Testing framework*. Last checked: April 3, 2013. URL: http://qunitjs.com/.

[11] Running documentation. *Jasmine is a behavior-driven development framework for testing JavaScript code*. Last checked: April 3, 2013. URL: http://pivotal.github.com/jasmine/.

[12] Christian Johansen. *Sinon.JS: Standalone test spies, stubs and mocks for JavaScript*. Last checked: April 5, 2013. URL: http://sinonjs.org/.

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

[14] Rudy Lattae. *Jasmine-species: Extended BDD grammar and reporting for Jasmine*. Last checked: April 5, 2013. URL: http://rudylattae.github.com/jasmine-species/.

[15] Friedel Ziegelmayer. *Spectacular Test Runner for JavaScript*. Last checked: May 29, 2013. URL: http://karma-runner.github.io/.

[16] TJ Holowaychuk. *mocha - simple, flexible, fun javascript test framework for node.js & the browser*. Last checked: April 3, 2013. URL: http://visionmedia.github.com/mocha/.

[17] Cory Smith et al. *JsTestDriver*. Last checked: April 5, 2013. URL: https://code.google.com/p/js-test-driver/.

[18] August Lilleaas and Christian Johansen. *BusterJS: A powerful suite of automated test tools for JavaScript*. Last checked: April 5, 2013. URL: http://docs.busterjs.org/.

[19] 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.

[20] 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.

[21] 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.

[22] Alexis Sellier, Charlie Robbins, and Maciej Malecki. *Vows: Asynchronous behaviour driven development for Node*. Last checked: April 5, 2013. URL: http://vowsjs.org/.

[23] Eugene Ware. *Cucumis: BDD Cucumber Style Asynchronous Testing Framework for node.js*. Last checked: April 5, 2013. URL: https://github.com/noblesamurai/cucumis.

[24] TJ Holowaychuk. *JSpec on Github no longer supported*. Last checked: April 5, 2013. URL: https://github.com/liblime/jspec.

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

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

[27] Igor Minar, Misko Hevery, and Vojta Jina. *Angular Templates*. Last checked: May 2, 2013. URL: http://docs.angularjs.org/tutorial/step_02.

[28] Mike Jansen. *Avoiding Common Errors in Your Jasmine Test Suite*. Last checked: June 12, 2013. URL: http://blog.8thlight.com/mike-jansen/2011/11/13/avoiding-common-errors-in-your-jasmine-specs.html.

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